异步图书
www.epubit.com.cn

Pearson

- 经典著作全面更新
- Python 程序员案头必备
- 涵盖 Python 2 和 Python 3 共有特性

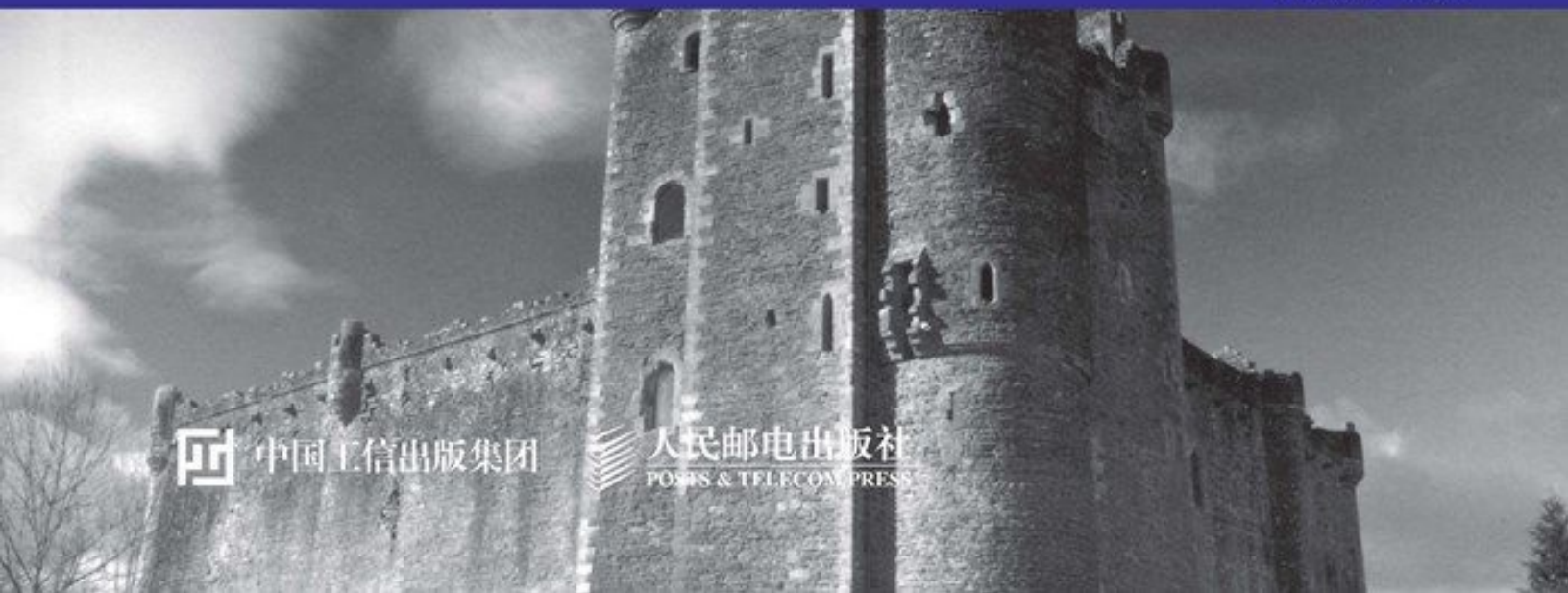
Python参考手册

（第4版·修订版）

Python

Essential Reference Fourth Edition

[美] David M.Beazley 著
谢俊 杨越 高伟 译
宋秉金 审校



中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

目 录

版权信息

书名：Python参考手册（第4版•修订版）

ISBN：978-7-115-39439-2

- 著　　[美] David M.Beazley

  译　　谢　俊　杨　越　高　伟

  审　　校　宋秉金

  责任编辑　杨海玲

- 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

  邮编　100164　电子邮件　315@ptpress.com.cn

  网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

  反盗版热线：(010)81055315

内容提要

本书是Python编程语言的权威参考手册，书中详尽讲解了Python核心和Python库中最重要的部分。全书分为三个部分，第一部分涉及类型与对象，运算符与表达式，程序结构与控制流，函数与函数式编程，类与面向对象编程，模块、包与分发，输入与输出，测试、调试、探查与调优等与Python语言相关的内容；第二部分涉及内置函数和异常、运行时服务、数学运算、数据结构、算法与代码优化，字符串和文本处理，数据库访问，文件和目录处理，操作系统服务，线程与并发，网络编程与套接字，网络应用程序编程，Web编程，网络数据处理和编码、其他库模块等与Python库相关的内容；第三部分涉及扩展和嵌入Python等内容。此外，书中还包括一些Python官方文档或其他参考资料中未提及的高级主题。

本书适合Python程序员以及具备其他编程语言经验的开发人员阅读和参考。

版权声明

谨将本书献给Paula、Thomas和他即将出生的兄弟。

前言

本书是Python编程语言的一份简明参考。尽管有经验的程序员也可以通过本书学习Python语言，但本书并不是一本讲述如何编程的全面教程或专著。相反，本书的目标是准确而简练地介绍Python语言核心以及Python库中最关键的部分。本书假定读者以前拥有Python或其他语言（如C或Java）的编程经验。另外，对系统编程（例如基本的操作系统概念和网络编程）有大致的了解对理解Python库参考中的部分章节可能会有一定的帮助。

Python可以从http://www.python.org免费下载。几乎每一种操作系统，包括UNIX、Windows和Macintosh，都有相应的版本。另外，Python网站还包含文档、指南和各种第三方软件的链接。

本书的这一版是在Python发展的关键时刻面世的。Python 2.6和Python 3.0几乎是同时发布的，但Python 3与以前的版本并不兼容。作为一名作者和程序员，我面临着一个两难问题：是直接跳到Python 3.0，还是使用大多数程序员更为熟悉的Python 2.x版本？

多年前，在应该使用哪些语言特性的问题上，身为C程序员的我经常把某些书当作终极权威。例如，使用K&R书中未曾提到过的某些特性很可能导致程序无法移植，因此得格外谨慎。作为程序员，这种方法对我非常有效，因此我决定在本书的这个版本中沿用这一做法。也就是说，这一版删除了Python 3中已经不再支持的Python 2特性。类似地，我也没有关注Python 3中不能向后兼容的特性（尽管附录中仍然会介绍这些特性）。最终，我希望，无论使用的是哪个版本的Python，本书都能够成为Python程序员案头必备的参考书。

这一版还包含自第1版出版至今10余年来最激动人心的一些变化。在过去的数年间，Python的发展重点体现在新的语言特性上——特别是与函数式编程和元编程相关的特性。因此，讲述函数式编程和面向对象编程的章节得到了极大的扩充，包括的主题有生成器、迭代器、协程、装饰器和元类。讲述库的章节则将重点转移到更加现代的模块上。整本书中的示例和代码片段都做了更新。我认为大多数程序员会对扩充之后的内容感到满意。

最后值得一提的是，Python本身已经有数千页有用的文档。本书的内容在很大程度上是基于这些文档的，但又有很多关键的区别。首先，这本书提供信息的方式更加紧凑，使用不同的示例，并对很多主题提供了不同的描述。其次，Python库参考部分的大量主题都做了扩充，包含很多外部的参考资料。底层系统和网络模块尤其如此，因为对这些模块的有效使用往往依赖于手册和外部参考中列出的种种选项。另外，为了使参考手册更加简明，这一版还删掉了很多已经废弃和相对晦涩的库模块。

我的目标是写出一本真正包含使用Python及其众多模块所需的一切内容的参考手册。尽管这并不是一本介绍Python语言的百科全书，但我希望本书能成为你未来数年里一本有用的必备编程参考书。十分欢迎读者对本书提出意见和建议。

David Beazley

2009年**6**月

于伊利诺伊州芝加哥

致谢

本书能与读者见面，要感谢很多人的大力支持。首先要感谢Noah Gift参与这个项目，并提出了许多建设性意见。Kurt Grandis也对很多章节发表了中肯的见解。我还要感谢前几版的技术审稿人Timothy Boronczyk、Paul DuBois、Mats Wichmann、David Ascher和Tim Bell，他们的精彩意见和建议促成了过去几版的成功。在1999年那个炎热的夏天，Guido van Rossum、Jeremy Hylton、Fred Drake、Roger Masse和Barry Warsaw招待了我好几个星期，同时对第1版提供了极大的帮助。最后，同样重要的是，本书的问世离不开读者的热情反馈。要感谢的人实在太多，这里无法一一列出他们的名字，我已尽力采纳大家的建议来让本书变得更好。我还要感谢Addison-Wesley和Pearson Education的工作人员，他们对这个项目给予了持续的支持与帮助。Mark Taber、Michael Thurston、Seth Kerney和Lisa Thibault都对本书的顺利出版倾注了大量的心血。还要特别感谢Robin Drake，他在第3版出版的过程中做了大量的编辑工作。最后，我要感谢我了不起的妻子和好搭档Paula Kamen，感谢她给予我的所有鼓励、幽默和爱。

第一部分　**Python**语言

本部分内容

第1章　Python简介

本章将快速介绍Python这门语言，目标是在阐明Python的大部分基本特性的同时，又不会太过纠缠于特殊的规则或细节。为此，本章简要讲述一些基本概念，如变量、表达式、控制流、函数、生成器、类和输入/输出。本章不追求大而全，但有经验的程序员应该能够把本章中的资料推而广之，创建出更加高级的程序。鼓励初学者多尝试一些示例，找到对这门语言的感觉。如果你对Python不熟悉也没有使用过Python 3，可以使用Python 2.6来学习本章内容。本章介绍的几乎所有主要概念同时适用于这两个版本，但在Python 3中有少数关键语法变化（其中大多数与打印和I/O有关），可能会使本章中介绍的许多示例无法运行。请参考附录A，以了解详细信息。

1.1　运行Python

Python程序是由解释器来执行的。通常，只要在命令shell中输入`python` 即可启动解释器。然而，解释器和Python开发环境存在多种实现（如Jython、IronPython、IDLE、ActivePython、Wing IDE、pydev等），因此需要参考相应文档中的启动说明。解释器启动后将出现一个提示符，在此可以开始输入程序，进入简单的读入-求值循环。例如，在下面的输出中，解释器显示了版权消息和>>>提示符，用户可以在提示符后输入熟悉的打印“Hello World”命令：

```
Python 2.6rc2 (r26rc2:66504, Sep 19 2008, 08:50:24)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World"


Hello World
>>>
```

注意

如果在尝试前述例子时出现了语法错误，你使用的很可能就是Python 3。遇到这种情况并不妨碍你继续学习本章的内容，但要注意`print` 语句在Python 3中已经变为一个函数。在下面的例子中，只要在要打印的内容两边加上括号即可正常运行，例如：

```
>>> print("Hello World")
Hello World
>>>
```

如果要打印的内容只有一项，在要打印内容两边放置括号的方法在Python 2中同样有效。然而，这种语法在现有的Python代码中并不常见。在后面的章节中，这种语法有时会用在与打印无直接关系的展示特性的例子中，但这些例子应该同时适用于Python 2和3。

Python的交互模式是它最有用的功能之一。在交互式shell中，可以输入任意合法的语句或语句序列，然后立即查看结果。很多人甚至使用交互式Python作为桌面计算器，作者本人也是如此。例如：

```
>>> 6000 + 4523.50 + 134.12


10657.620000000001
>>> _ + 8192.32


18849.940000000002
>>>
```

以交互模式使用Python时，特殊变量_ 保存最后一次运算的结果。如果要在后续语句中保存或使用最后一次运算的结果，使用此变量十分方便。但要强调一点，此变量只有在以交互模式编程时才会被定义。

如果要创建可以重复运行的程序，可将语句放到一个文件中：

```
# helloworld.py
print "Hello World"
```

Python源文件是普通的文本文件，后缀通常是.py 。# 字符表示该行直至行尾的内容都是注释。

要执行helloworld.py 文件，可通过如下方式将文件名提供给解释器：

```
% python helloworld.py


Hello World
%
```

在Windows中，双击一个.py 文件或者在Windows开始菜单的“运行”命令中输入程序名称，均可启动Python程序。这会启动解释器，并在控制台窗口中运行程序。但要注意，当程序执行完成后，控制台窗口将立即消失（通常你来不及看清楚输出）。要进行调试，最好是在像IDLE这样的Python开发工具中运行程序。

在UNIX中，可以在程序的首行中使用#!，如下所示：

```
#!/usr/bin/env python
print "Hello World"
```

解释器不断运行语句，直到到达输入文件的结尾。如果是以交互模式运行，有两种方法可以退出解释器，一种是输入EOF（end of file，文件结束）字符，另一种是从Python IDE的下拉菜单中选择Exit。在UNIX中，EOF是Ctrl+D，而在Windows中则是Ctrl+Z。程序可以通过抛出SystemExit 异常来请求退出。

```
>>> raise SystemExit
```

1.2 变量和算术表达式

程序清单1-1中的程序通过执行一次简单的复利计算，说明变量和表达式的用法。

程序清单1-1　简单的复利计算

```
principal = 1000           # 初始金额
rate = 0.05                # 利率
numyears = 5               # 年数
year = 1
while year <= numyears:
    principal = principal * (1 + rate)
    print year, principal  # 注意：在Python 3中应写成print(year, principal)
    year += 1
```

此程序的输出如下所示：

```
1 1050.0
2 1102.5
3 1157.625
4 1215.50625
5 1276.2815625
```

Python是一种动态类型的语言，在程序执行过程中，变量名称会被绑定到不同的值，而且这些值可以属于不同的类型。赋值运算符的作用仅仅是在名称和值之间创建一种关联。尽管每个值都有一个相关类型，如integer 或string ，但变量名称是无类型的，在执行过程中可以引用任意类型的数据。这与C语言不同，例如，在C语言中，名称代表了用于保存值的固定类型、大小和内存位置。Python的动态行为可以从程序清单1-1的principal 变量看出来。最初给它分配的是一个integer 值，但程序稍后给它重新赋了值，如下所示：

```
principal = principal * (1 + rate)
```

这条语句对表达式求值，并把名称principal 重新与结果关联。principal 的原始值是整数类型的1000 ，但现在的新值是浮点数（rate 被定义为浮点数，因此上述表达式的值也是浮点数）。因此在程序中，principal 看上去的“类型”就从integer 动态变为了float 。然而准确地说，不是principal 的类型变了，而是principal 名称引用的值的类型变了。

换行代表一条语句的结束。然而，也可以在同一行上使用分号来隔开多条语句，如下所示：

```
principal = 1000; rate = 0.05; numyears = 5;
```

while 语句对随后的条件表达式进行检验。如果被检验的语句为真，while 语句的主体就会执行。然后再次检验条件，再执行主体，直到条件为假。因为循环主体是由缩进表示的，每次循环时都会执行程序清单1-1中while 之后的3条语句。Python不会指定所需缩进的量，只要在一个代码块中保持一致即可。然而，每个缩进层次使用4个空格是最常见的情况，而且通常也建议这么做。

程序清单1-1中的程序有一个问题，即输出不是很美观。为了改进这一点，可以让各列右对齐，并将principal 的精度限制为两位。实现这种格式有几种方法。最常用的方法是使用字符串格式化运算符% ，如下所示：

```
print "%3d %0.2f" % (year, principal)
print("%3d %0.2f" % (year, principal))   # Python 3
```

现在程序的输出如下：

```
1 1050.00
2 1102.50
3 1157.63
4 1215.51
5 1276.28
```

格式化字符串包含普通文本和特殊的格式化字符序列，如"%d" 、"%s" 和"%f" 。这些序列分别用于指定特定类型数据的格式，如整数、字符串或浮点数。特殊字符序列还可以包含用于指定宽度和精度的修饰符。例如，"%3d" 将一个整数格式化为在一个宽度为3的列中右对齐，而"%0.2f" 将一个浮点数格式化为在小数点后只出现两位数字。格式化字符串的行为与C语言中的printf() 函数几乎完全相同，第4章将对此进行详细说明。

更新潮的字符串格式化的方法是使用format() 函数单独格式化每个部分。例如：

```
print format(year,"3d"),format(principal,"0.2f")
print(format(year,"3d"),format(principal,"0.2f")) # Python 3
```

format() 函数使用的格式说明符类似于传统字符串格式化运算符（% ）使用的格式说明符。例如，"3d" 将一个整数格式化为在一个宽度为3的列中右对齐，而"%0.2f" 将一个浮点数格式化为两位精度。字符串也有一个format() 方法，可用于一次性格式化很多值。例如：

```
print "{0:3d} {1:0.2f}".format(year,principal)
print("{0:3d} {1:0.2f}".format(year,principal)) # Python 3
```

在这个例子中，"{0:3d}" 和"{1:0.2f}" 中冒号前的数字代表传递给format() 方法的相关参数，而冒号后的部分则是格式说明符。

1.3 条件语句

if 与else 语句可执行简单的检验，如下所示：

```
if a < b:
    print "Computer says Yes"
else:
    print "Computer says No"
```

if 和else 子句的主体是用缩进表示的。else 子句是可选的。

要创建一条空子句，可以使用pass 语句，如下所示：

```
if a < b:
    pass      # 什么也不执行
else:
    print "Computer says No"
```

使用or 、and 和not 关键字可以组成布尔表达式：

```
if product == "game" and type == "pirate memory" \
                    and not (age < 4 or age > 8):
    print "I'll take it!"
```

> 注意
>
> 编写复杂的检验条件通常需要编写很长的代码行，看起来令人生厌。为了提高代码的可读性，可以像上面一样在一行的结尾使用反斜杠（\），然后就可以在下一行继续书写上一条语句的内容。如果这样做，正常的缩进规则将不被应用于下一行，因此可以随意设置后续行的格式。

Python没有专门的switch 或case 语句用于检测多个值。要处理多个检验条件，可以使用elif 语句，如下所示：

```
if suffix == ".htm":
    content = "text/html"
elif suffix == ".jpg":
    content = "image/jpeg"
elif suffix == ".png":
    content = "image/png"
else:
    raise RuntimeError("Unknown content type")
```

要表示真值，可以使用布尔值True 和False ，例如：

```
if 'spam' in s:
    has_spam = True
else:
    has_spam = False
```

所有关系运算符（如< 和> ）的结果都返回True 或False 。本例中使用的in 运算符通常用于检查某个值是否包含在另一个对象（如字符串、列表或字典）中。它也返回True 或False ，因此前一个例子可以缩短为：

```
has_spam = 'spam' in s
```

1.4 文件输入和输出

以下程序可打开一个文件并逐行读取该文件的内容：

```
f = open("foo.txt")          # 返回一个文件对象
line = f.readline()          # 调用文件的readline()方法
while line:
    print line,              # 后面跟','将忽略换行符
    # print(line,end='')     # 在Python 3中使用
    line = f.readline()
f.close()
```

open() 函数返回一个新的文件对象。调用该对象的方法可以执行各种文件操作。readline() 方法读取一行内容，包括结尾的换行符在内。读至文件结尾时将返回空字符串。

在这个例子中，程序只是循环读取了文件foo.txt 中的所有行。如果程序在像这样的数据集（如输入中的行、数字、字符串等）上进行循环，那么这通常就称为迭代 。因为迭代是很常见的一种操作，所以Python为其提供了一条专用语句for ，用于迭代内容项。例如，同样的程序可以写成下面这种更简洁的形式：

```
for line in open("foo.txt"):
    print line,
```

要将程序的输出写入一个文件中，需要在print 语句后面使用>> 指定一个文件，如下所示：

```
f = open("out","w")     # 打开文件以便写入
while year <= numyears:
    principal = principal * (1 + rate)
    print >>f,"%3d %0.2f" % (year,principal)
    year += 1
f.close()
```

>> 语法只能用在Python 2中。如果使用Python 3，可将print 语句改为以下内容：

```
print("%3d %0.2f" % (year,principal),file=f)
```

另外，文件对象支持使用write() 方法写入原始数据。例如，前一例子中的print

语句也可以写成下面这样：

```
f.write("%3d %0.2f\n" % (year,principal))
```

尽管这些例子处理的都是文件，但同样的技巧也适用于标准的解释器输出流和输入流。例如，如果想交互式地读取用户输入，可以从文件`sys.stdin` 中读取。如果要将数据输出到屏幕上，可以写入文件`sys.stdout` 中，这与在输出`print` 语句所生成数据时所用的文件是同一个文件。例如：

```
import sys
sys.stdout.write("Enter your name :")
name = sys.stdin.readline()
```

在Python 2中，这段代码还可以简化为：

```
name = raw_input("Enter your name :")
```

在Python 3中，`raw_input()` 函数叫做`input()`，但它们的工作方式完全相同。

1.5 字符串

要创建一个字符串字面量，将字符串放在单引号、双引号或三引号中即可，如下所示：

```
a = "Hello World"
b = 'Python is groovy'
c = """Computer says 'No'"""
```

字符串前后使用的引号必须是对应匹配的。两个三引号之间出现的所有文本都视为字符串的内容，而使用单引号和双引号指定的字符串必须在一个逻辑行上。当字符串字面量的内容需放在多个文本行上时，三引号字符串就很有用，如下所示：

```
print '''Content-type: text/html

<h1> Hello World </h1>
Click <a href="http://www.python.org">here.
'''
```

字符串存储在一个字符序列中，这个字符序列使用整数作为索引，索引从0开始。要提取其中的一个字符，可以使用索引运算符 *`s [i ]`*，如下所示：

```
a = "Hello World"
b = a[4]                # b = 'o'
```

要提取一个子字符串，可以使用切片运算符 *s* [*i* :*j*]。这会提取字符串 *s* 中索引位置 *k* 处的所有字符，其中索引 *k* 的范围是i<=*k* <*j* 。如果省略 *i* ，则假定使用字符串的起始位置，如果省略 *j* ，则假定使用字符串的结尾位置：

```
c = a[:5]              # c = "Hello"
d = a[6:]              # d = "World"
e = a[3:8]             # e = "lo Wo"
```

可以使用加（+ ）运算符连接两个字符串：

```
g = a + " This is a test"
```

Python不会把字符串的内容隐式地解释为数值数据（Perl或PHP等语言中会这样解释）。例如，+ 运算符始终会连接字符串：

```
x = "37"
y = "42"
z = x + y    # z = "3742"（字符串连接）
```

要执行数学计算，首先要使用int() 或float() 等函数将字符串转换为数值，例如：

```
z = int(x) + int(y)   # z = 79（整数求和）
```

使用str() 、repr() 或format() 函数可将非字符串值转换为字符串表示形式，例如：

```
s = "The value of x is " + str(x)
s = "The value of x is " + repr(x)
s = "The value of x is " + format(x,"4d")
```

尽管str() 和repr() 都可以创建字符串，但它们的输出通常存在细微的差别。str() 生成的输出与使用print 语句得到的输出相同，而用repr() 创建的字符串可表示程序中某个对象的精确值，例如：

```
>>> x = 3.4


>>> str(x)


'3.4'
>>> repr(x)
```

```
'3.3999999999999999'
>>>
```

上例中3.4 的不精确表示并非是Python中的一个bug。这是双精度浮点数的一个特点，因为从设计上说，底层计算机硬件无法精确地表示十进制小数。

format() 函数可将值转换成特定格式等字符串，例如：

```
>>> format(x,"0.5f")


'3.40000'
>>>
```

1.6 列表

列表 是任意对象组成的序列。把值放入方括号中就可以创建列表，如下所示：

```
names = [ "Dave", "Mark", "Ann", "Phil" ]
```

列表使用从0开始的整数索引，使用索引运算符可以访问并修改列表中的项：

```
a = names[2]              # 返回列表的第3项"Ann"
names[0] = "Jeff"         # 将第1项改为"Jeff"
```

要将新项追加到列表末尾，可使用append() 方法：

```
names.append("Paula")
```

要将一项插入到列表中，可使用insert() 方法：

```
names.insert(2, "Thomas")
```

使用切片运算符可以提取一个子列表或对子列表重新赋值：

```
b = names[0:2]                      # 返回[ "Jeff", "Mark" ]
c = names[2:]                       # 返回[ "Thomas", "Ann", "Phil", "Paula" ]
names[1] = 'Jeff'                   # 将names中的第2项替换为'Jeff'
names[0:2] = ['Dave','Mark','Jeff']  # 将列表的头两项替换为右边的列表
```

使用加号（+ ）可以连接列表：

```
a = [1,2,3] + [4,5]    # 结果是[1,2,3,4,5]
```

创建一个空列表有两种方式：

```
names = []        # 一个空列表
names = list()    # 一个空列表
```

列表可以包含任意种类的Python对象，包括其他列表在内，如下例所示：

```
a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]
```

嵌套列表中包含的项需要使用多次索引操作才能访问到，例如：

```
a[1]             # 返回 "Dave"
a[3][2]          # 返回 9
a[3][3][1]       # 返回 101
```

程序清单1-2中的程序展示了列表的一些高级特性，该程序会读取在命令行上指定的一个文件中的数值列表，然后输出其中的最大值和最小值。

程序清单1-2　列表的高级特性

```
import sys                        # 加载sys模块
if len(sys.argv) != 2             # 检查命令行参数的数量：
     print "Please supply a filename"
     raise SystemExit(1)
f = open(sys.argv[1])             # 命令行上的文件名
lines = f.readlines()             # 将所有行读取到一个列表中
f.close()

# 将所有输入值从字符串转换为浮点数
fvalues = [float(line) for line in lines]

# 打印最小值和最大值
print "The minimum value is ", min(fvalues)
print "The maximum value is ", max(fvalues)
```

该程序的第一行使用`import` 语句从Python库加载`sys` 模块。加载该模块的目的是获得命令行参数。

`open()` 函数使用了一个文件名，该文件名是以命令行选项的形式提供的并保存在列表`sys.argv` 中。`readline()` 方法将所有输入行读取到一个字符串列表中。

表达式`[float(line) for line in line]` 通过对列表`lines` 中的所有字符串进行循环，并对每个元素应用函数`float()`，从而构造一个新列表。这种功能特别强大的列表构造方法叫做列表推导 （list comprehension）。因为你还可以使用`for` 循环来读取文件

中的行，所以可以将上面程序中转换值的代码简化为一条语句：

```
fvalues = [float(line) for line in open(sys.argv[1])]
```

将输入行转换成一个浮点数列表后，再使用内置函数min() 和max() 计算出最大值和最小值即可。

1.7 元组

要创建简单的数据结构，可以使用元组 将一组值打包到一个对象中。在圆括号中放入一组值即可创建元组，例如：

```
stock = ('GOOG', 100, 490.10)
address = ('www.python.org', 80)
person = (first_name, last_name, phone)
```

即使没有圆括号，Python通常也能识别出元组：

```
stock = 'GOOG', 100, 490.10
address = 'www.python.org',80
person = first_name, last_name, phone
```

为了完整起见，也可以定义0个和1个元素的元组，但语法较为特殊：

```
a = ()          # 0元组 （空元组）
b = (item,)     # 1元组 （注意随后的逗号）
c = item,       # 1元组 （注意随后的逗号）
```

和列表一样，也可以使用数字索引来提取元组中的值。然而，更常见的做法是将元组解包为一组变量，例如：

```
name, shares, price = stock
host, port = address
first_name, last_name, phone = person
```

尽管元组支持的大部分操作与列表的相同（如索引、切片和连接），但创建元组后不能修改它的内容（也就是说无法替换、删除现有元组中的元素，或者向现有元组中添加新元素）。这说明最好把元组看成一个由多个部分组成的单一对象，而不是可在其中插入或删除项的不同对象的集合。

因为元组与列表之间存在诸多相似之处，所以有些程序员往往完全忽略了元组，而只使用列表，因为后者看似更灵活。尽管这并无不可，但如果程序创建了大量的小列表（即

包含的项少于十来个），则会造成内存浪费。这是因为系统会为列表分配稍微多一些内存，以优化添加新项的操作的性能。而由于元组是不可变的，所以它们的展现更为紧凑，不会占据额外的内存空间。

表示数据时，经常同时使用元组和列表。例如，下面的程序显示了如何读取包含不同数据列，且各数据列由逗号隔开的文件：

```
# 文件中各行的格式为"name,shares,price

"
filename = "portfolio.csv"
portfolio = []
for line in open(filename):
    fields = line.split(",")       # 将每行划分为一个列表
    name   = fields[0]             # 提取并转换每个字段
    shares = int(fields[1])
    price  = float(fields[2])
    stock  = (name,shares,price)   # 创建一个元组(name, shares, price)
    portfolio.append(stock)        # 将记录追加到列表中
```

字符串的split()方法会按照指定的分隔符将一个字符串划分为一个字段列表。该程序最后创建的portfolio数据结构类似一个二维的行列数组，每行由一个元组表示，并可通过如下方式访问：

```
>>> portfolio[0]


('GOOG', 100, 490.10)
>>> portfolio[1]


('MSFT', 50, 54.23)
>>>
```

每个数据项可以通过如下方式访问：

```
>>> portfolio[1][1]


50
>>> portfolio[1][2]


54.23
>>>
```

下面给出了一种循环访问所有记录并将字段展开到一组变量中的简单方法：

```
total = 0.0
for name, shares, price in portfolio:
    total += shares * price
```

1.8 集合

集合 用于包含一组无序的对象。要创建集合，可使用set()函数并像下面这样提供一系列的项：

```
s = set([3,5,9,10])       # 创建一个数值集合
t = set("Hello")          # 创建一个唯一字符的集合
```

与列表和元组不同，集合是无序的，也无法通过数字进行索引。此外，集合中的元素不能重复。例如，在检查前面代码中t集合的值时，会得到这样的结果：

```
>>> t


set(['H', 'e', 'l', 'o'])
```

注意，只出现了一个'l'。

集合支持一系列标准操作，包括并集、交集、差集和对称差集，例如：

```
a = t | s          # t和s的并集
b = t & s          # t和s的交集
c = t - s          # 差集（项在t中，但不在s中）
d = t ^ s          # 对称差集（项在t或s中，但不会同时出现在二者中）
```

使用add()或update()可以在集合中添加新项：

```
t.add('x')            # 添加一项
s.update([10,37,42])  # 在s中添加多项
```

使用remove()可以删除一项：

```
t.remove('H')
```

1.9 字典

字典 就是一个关联数组或散列表，其中包含通过键（key）索引的对象。在大括号（{ }）中放入值即可创建字典，如下所示：

```
stock = {
        "name"   : "GOOG",
        "shares" : 100,
```

```
        "price"  : 490.10
    }
```

要访问字典成员，可使用键索引运算符，如下所示：

```
name  = stock["name"]
value = stock["shares"] * shares["price"]
```

插入或修改对象的方法是：

```
stock["shares"] = 75
stock["date"]   = "June 7, 2007"
```

尽管字符串是最常用的键类型，还可以使用其他的Python对象，包括数值和元组。但包括列表和字典在内的一些对象不能用作键，因为它们的内容可以发生变化。

如前所述，在定义一个可包含多个命名字段的对象时，字典是一种很有用的方式。然而，字典也可用作快速查找无序数据的一个容器。例如，下面是一个股票价格的字典：

```
prices = {
    "GOOG" : 490.10,
    "AAPL" : 123.50,
    "IBM"  : 91.50,
    "MSFT" : 52.13
}
```

创建一个空字典有两种方式：

```
prices = {}      # 一个空字典
prices = dict()  # 一个空字典
```

使用`in`运算符可以检验某个内容项是不是字典成员，如下所示：

```
if "SCOX" in prices:
    p = prices["SCOX"]
else:
    p = 0.0
```

这个特殊的步骤序列还可以写成更简洁的形式，如下所示：

```
p = prices.get("SCOX",0.0)
```

要获得一个字典关键字的列表，将字典转换为列表即可：

```
syms = list(prices)       # syms = ["AAPL", "MSFT", "IBM", "GOOG"]
```

使用del 语句可以删除字典的元素：

```
del prices["MSFT"]
```

字典可能是Python解释器中最完善的数据类型。因此，如果只是要在程序中存储和处理数据，使用字典比使用一些自定义数据结构要好得多。

1.10 迭代与循环

最常用的循环结构是for 语句，它可以用来对容器成员进行迭代操作。迭代是Python中内涵最丰富的功能之一。但最常见的迭代形式是简单循环访问一个序列（如字符串、列表或元组）的所有成员，例如：

```
for n in [1,2,3,4,5,6,7,8,9]:
    print "2 to the %d power is %d" % (n, 2**n)
```

在这个例子中，每次迭代都会将列表[1,2,3,4,...,9] 中的下一个值赋给变量n 。因为在整数范围内执行循环十分常见，为此经常会使用下面的快捷方法：

```
for n in range(1,10):
    print "2 to the %d power is %d" % (n, 2**n)
```

range(*i* , *j* , [,步长]) 函数创建的对象表示值在 *i* 到 *j* -1 之间的整数范围。如果起始值i 被省略，则认为是0。第三个参数是可选的步长值。例如：

```
a = range(5)         # a = 0,1,2,3,4
b = range(1,8)       # b = 1,2,3,4,5,6,7
c = range(0,14,3)    # c = 0,3,6,9,12
d = range(8,1,-1)    # d = 8,7,6,5,4,3,2
```

在使用range() 函数时请注意，在Python 2中，它创建的值是已经用整数值完全填满的列表。当范围非常大时，这可能会在不经意间耗掉所有可用内存。因此，在老式的Python代码中，可能会看到程序员使用另一个函数xrange() 。例如：

```
for i in xrange(100000000):    # i = 0,1,2,...,99999999
    statements
```

进行查找时，xrange() 函数创建的对象会在收到查询请求时根据需要计算它所表示的值。因此，它成为了表示极大范围整数值的首选方式。在Python 3中，xrange() 函数

已经更名为range()，并且已删除了老式range()函数的功能。

for语句并不仅限于处理整数序列，还可用于迭代多种对象，包括字符串、列表、字典和文件，例如：

```
a = "Hello World"
# 打印出a中的每个字符
for c in a:
    print c

b = ["Dave","Mark","Ann","Phil"]
# 打印出一个列表的成员
for name in b:
    print name

c = { 'GOOG' : 490.10, 'IBM' : 91.50, 'AAPL' : 123.15 }
# 打印出一个字典的所有成员
for key in c:
    print key, c[key]

# 打印一个文件中的所有行
f = open("foo.txt")
for line in f:
    print line,
```

for循环是Python最强大的语言特性之一，因为你可以创建自定义的迭代器对象和生成器函数，为它提供值序列。本章稍后和第6章将会讲述有关迭代器和生成器的更多内容。

1.11 函数

使用def语句可以创建函数，如下例所示：

```
def remainder(a,b):
    q = a // b      # //是截断除法运算符
    r = a - q*b
    return r
```

要调用函数，只要使用函数名加上用圆括号括起来的参数即可，如result = remainder(37, 15)。如果要让函数返回多个值，可以使用元组，如下所示：

```
def divide(a,b):
    q = a // b      # 如果a和b是整数，q就是整数
    r = a - q*b
    return (q,r)
```

使用元组返回多个值时，可以很容易地将结果解包到单独的变量中，例如：

```
quotient, remainder = divide(1456,33)
```

要给函数参数提供一个默认值，可使用以下赋值方式：

```
def connect(hostname,port,timeout=300):
     # 函数体
```

在函数定义中给一个参数提供默认值以后，调用此函数时就可以省略该参数，此时该参数将使用默认值，如下所示：

```
connect('www.python.org', 80)
```

还可以使用关键字参数调用函数，此时可以按任意顺序提供参数，但这需要你知道函数定义中的参数名称，如下所示：

```
connect(port=80,hostname="www.python.org")
```

在函数中创建变量或给变量赋值时，该变量的作用域是局部的。也就是说，该变量只定义在函数体内部，而且当函数返回值后会立即销毁该变量。要在函数内部修改某个全局变量的值，可以使用global 语句，如下所示：

```
count = 0
...
def foo():
    global count
    count += 1           # 更改全局变量count
```

1.12 生成器

如果使用yield 语句，可以让函数生成一个结果序列，而不仅仅是一个值，例如：

```
def countdown(n):
    print "Counting down!"
    while n > 0:
         yield n       # 生成一个值（n）
         n -= 1
```

任何使用yield 的函数都称为生成器 。调用生成器函数将创建一个对象，该对象通过连续调用next() 方法（在Python 3中是__next__() ）生成一系列的结果，例如：

```
>>> c = countdown(5)
```

```
>>> c.next()


Counting down!
5
>>> c.next()


4
>>> c.next()


3
>>>
```

next() 调用使生成器函数一直运行，到下一条yield 语句为止。此时next() 将返回传递给yield 的值，而且函数将暂时中止执行。再次调用next() 时，函数将继续执行yield 之后的语句。此过程持续到函数返回为止。

通常不会像上面这样手动调用next()，而是会使用一个for 循环，例如：

```
>>> for i in countdown(5):


...     print i,


Counting down!
5 4 3 2 1
>>>
```

生成器是编写基于处理管道、流或数据流程序的一种极其强大的方式。例如，下面的生成器函数模拟了常用于监控日志文件的UNIX tail -f 命令的行为：

```
# tail一个文件（如tail -f）
import time
def tail(f):
    f.seek(0,2)     # 移动到EOF
    while True:
        line = f.readline()    # 尝试读取一个新的文本行
        if not line:           # 如果没有内容，暂时休眠并再次尝试
             time.sleep(0.1)
             continue
        yield line
```

下面的生成器用于在很多行中查找特定的子字符串：

```
def grep(lines, searchtext):
    for line in lines:
        if searchtext in line: yield line
```

下面的例子将以上两个生成器合并在一起，创建了一个简单的处理管道：

```
# UNIX "tail -f | grep python"命令的python实现
wwwlog = tail(open("access-log"))
pylines = grep(wwwlog,"python")
for line in pylines:
    print line,
```

生成器的微妙之处在于，它经常和其他可迭代的对象（如列表或文件）混合在一起。特别是在编写如for item in s 这样的语句时，s 可以代表一个列表、文件的各行、生成器函数的结果，或者支持迭代的其他任何对象。能够在s 中插入不同对象，为创建可扩展的程序提供了一个强大的工具。

1.13 协程

通常，函数运行时要使用单一的一组输入参数。但是，函数也可以编写成一个任务程序， 用来处理发送给它的一系列输入。这类函数被称为协程 ，它是通过将yield 语句作为表达式(yield) 的形式创建的，如下所示：

```
def print_matches(matchtext):
    print "Looking for",matchtext
    while True:
         line = (yield)    # 获得一行文本
         if matchtext in  line:
             print line
```

要使用这个函数，首先要调用它，向前执行到第一条(yield) 语句，然后使用send() 给它发送数据，例如：

```
>>> matcher = print_matches("python")


>>> matcher.next()

# 向前执行到第一条(yield)语句
Looking for python
>>> matcher.send("Hello World")


>>> matcher.send("python is cool")


python is cool
>>> matcher.send("yow!")


>>> matcher.close()

    # matcher函数调用结束
```

```
>>>
```

使用send()为协程发送某个值之前，协程会暂时中止。发送值之后，协程中的(yield)表达式将返回这个值，而接下来的语句将会处理它。处理直到遇到下一个(yield)表达式才会结束，这时函数将暂时中止。正如上一个例子所示，这个过程将会继续下去，直到协程函数返回或者调用它的close()方法为止。

基于生产者—消费者模型（即程序的一部分生成的数据会被程序的另一部分使用）编写并发程序时，协程十分有用。在这种模型中，协程代表了数据的一个消费者。下面给出了一起使用生成器和协程的一个例子：

```
# 一组匹配器协程
matchers = [
    print_matches("python"),
    print_matches("guido"),
    print_matches("jython")
]
# 通过调用next()准备所有的匹配器
for m in matchers: m.next()

# 将一个活跃的日志文件传递给所有的匹配器。注意，为保证运行正常，
    # 必须有一台Web服务器持续将数据写入日志
wwwlog = tail(open("access-log"))
for line in wwwlog:
    for m in matchers:
        m.send(line)        # 将数据发送到每个匹配器协程中
```

第6章将会进一步介绍协程。

1.14 对象与类

程序中使用的所有值都是对象。对象 由内部数据和各种方法组成，这些方法会执行与这些数据相关的各种操作。前面在处理像字符串和列表这样的内置类型时，就已经用到了对象和方法。例如：

```
items = [37, 42]    # 创建一个列表对象
items.append(73)    # 调用append()方法
```

dir()函数可以列出对象上的可用方法，是进行交互式实验的有用工具，例如：

```
>>> items = [37, 42]


>>> dir(items)


['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
...
```

```
 'append', 'count', 'extend', 'index', 'insert', 'pop',
 'remove', 'reverse', 'sort']
>>>
```

查看对象时，会看到诸如append() 和insert() 等很熟悉的方法。但也可以看到固定以双下划线开始和结束的特殊方法。这些方法用于实现各种语言运算。例如，__add__() 方法实现了+运算符的功能：

```
>>> items.

 __add__([73,101])


[37, 42, 73, 101]
>>>
```

class 语句用于定义新的对象类型，实现面向对象编程。例如，下面的类定义了一个支持push() 、pop() 和length() 操作的简单栈：

```
class Stack(object):
    def __init__(self):           # 初始化栈
        self.stack = [ ]
    def push(self,object):
        self.stack.append(object)
    def pop(self):
        return self.stack.pop()
    def length(self):
        return len(self.stack)
```

在类定义的第一行中，语句class Stack(object) 将Stack 声明为一个object 。使用圆括号是Python指定继承的方式——在这个例子中，Stack 继承自object ，object 也是所有Python类型的根类型。类定义中使用def 语句定义了方法。每个方法中的第一个参数始终指向对象本身。根据约定，该参数的名称为self 。涉及对象属性的所有操作都必须显式引用self 变量。以双下划线开始和结束的方法是特殊的方法。例如，__init__ 用于在创建对象后初始化该对象。

要想使用类，可编写如下所示的代码：

```
s = Stack()           # 创建一个栈
s.push("Dave")        # 在栈中放入一些内容
s.push(42)
s.push([3,4,5])
x = s.pop()         # x的值为[3,4,5]
y = s.pop()         # y的值为42
del s               # 删除s
```

这个例子创建了一个全新的对象来实现栈。但是，栈与内置的列表对象几乎完全相同。因此，继承list 然后添加一个额外的方法也是可行的：

```
class Stack(list):
    # 为栈接口添加push()方法
    # 注意：列表已经提供了一个pop()方法
    def push(self,object):
        self.append(object)
```

通常，类中定义的所有方法只适用于该类的实例（即创建的对象）。但是，也可以定义不同种类的方法，如C++和Java程序员所熟知的静态方法，例如：

```
class EventHanlder(object):
    @staticmethod
    def dispatcherThread():
        while (1):
            # 等待请求
            ...

EventHandler.dispatcherTread()       # 像函数一样调用方法
```

在这个例子中，@staticmethod 将方法声明为静态方法。@staticmethod 是使用装饰器 （decorator）的一个例子，我们将在第6章中进一步介绍装饰器。

1.15 异常

如果程序中出现错误，就会引发异常，并显示类似下面的回溯消息：

```
Traceback (most recent call last):
 File "foo.py", line 12, in <module>
IOError: [Errno 2] No such file or directory: 'file.txt'
```

该回溯消息指出了所发生的错误类型及位置。通常情况下，错误会导致程序终止。但是可以使用try 和except 语句捕捉并处理异常，如下所示：

```
try:
    f = open("file.txt","r")
except IOError as e:
    print e
```

如果出现IOError ，引发错误的详细信息将被放在对象e 中，然后控制权被传递给except 代码块中的代码。如果出现其他类型的异常，对象e 将被传递给用于处理这些异常的代码块（如果有的话）。如果没有出现错误，except 代码块中的代码将被忽略。处理完异常后，程序将继续执行紧跟在最后一个except 代码块后面的语句。程序不会返回到发生异常的位置。

raise 语句用于手工引发异常。引发异常时，可以使用任意一个内置异常，如下所示：

```
raise RuntimeError("Computer says no")
```

你也可以创建自己的异常，这将在5.4.2节中详细介绍。

在进行异常处理时，如何正确地管理系统资源（如锁、文件和网络连接）通常是一个棘手的问题。为了简化此类编程，可以对某些种类的对象使用with 语句。下面的例子给出了使用互斥锁的代码：

```
import threading
message_lock = threading.Lock()
...
with message_lock:
    messages.add(newmessage)
```

在这个例子中，with 语句执行时会自动获取message_lock 对象。当执行离开with 代码块上下文后，锁将被自动释放。不管with 代码块内部发生了什么，都会出现这种管理行为。例如，如果出现一个异常，当控制离开代码块环境时锁也将被释放。

with 语句通常只适用于与系统资源或执行环境相关的对象，如文件、连接和锁。但是，用户定义的对象也可以定义自己的自定义处理机制。这一点将在3.9.11节中详细介绍。

1.16 模块

随着程序变得越来越大，为了便于维护，需要把它分为多个文件。为此，Python允许把定义放入一个文件中，然后在其他程序和脚本中将其作为模块导入。要创建模块，可将相关的语句和定义放入与模块同名的文件中（注意，该文件的后缀必须是.py ）。例如：

```
# 文件:    div.py
def divide(a,b):
    q = a/b       # 如果a和b是整数，则q也是整数
    r = a - q*b
    return (q,r)
```

要在其他程序中使用该模块，可以使用import 语句：

```
import div
a, b = div.divide(2305, 29)
```

import 语句创建了一个新的命名空间，并在该命名空间中执行与.py 文件相关的所有语句。要在导入后访问命名空间的内容，只要使用该模块的名称作为前缀，正如上面例子中的div.divide() 一样。

如果要使用不同的名称导入模块，可以给import 语句加上可选的as 限定符，如下所示：

```
import div as foo
a,b = foo.divide(2305,29)
```

要将具体的定义导入到当前的命名空间中，可使用from 语句：

```
from div import divide
a,b = divide(2305,29)      # 不再使用div前缀
```

要把模块的所有内容加载到当前的命名空间中，还可以使用以下语句：

```
from div import *
```

与对象一样，dir() 函数可以列出模块的内容，是进行交互式实验的有用工具：

```
>>> import string


>>> dir(string)


['__builtins__', '__doc__', '__file__', '__name__', '_idmap',


'_idmapL', '_lower', '_swapcase', '_upper', 'atof', 'atof_error',


'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize',
 'capwords', 'center', 'count', 'digits', 'expandtabs', 'find',
...
>>>
```

1.17 获得帮助

使用Python时，有几个快速获取可用信息的来源。首先，以交互模式运行Python时，可以使用help() 命令获得有关内置模块和Python其他方面的信息。单独输入help() 将获得一般信息，而输入help('模块名') 则可获得具体模块的信息。如果提供函数名称，help() 命令还可以返回该函数的详细信息。

大多数Python函数都有描述该函数用途的文档字符串。要打印这个文档字符串，只要打印__doc__ 属性即可。例如：

```
>>> print issubclass.__doc__


issubclass(C, B) -> bool

Return whether class C is a subclass (i.e., a derived class) of class B.
```

```
When using a tuple as the second argument issubclass(X, (A, B, ...)),
is a shortcut for issubclass(X, A) or issubclass(X, B) or ... (etc.).
>>>
```

最后但也很重要的一点是，大多数Python安装还包括了命令pydoc 。该命令用于返回有关Python模块的文档。只需在系统命令提示符后输入“pydoc 主题 ”即可。

第2章　词法约定和语法

本章介绍Python程序的词法和语法约定。本章涉及的主题包括行结构、语句分组、保留字、字面量、运算符、标记和源代码编码。

2.1　行结构和缩进

程序中的每条语句都以换行符结束。使用续行符（\）可将长语句分为几行，如下所示：

```
a = math.cos(3 * (x - n)) + \
    math.sin(3 * (y - n))
```

三引号字符串、列表、元组或字典的定义如果跨越多行，是可以不使用续行符的。一般来说，包含在圆括号(...)、方括号[...]、大括号{...}或三引号中的任意程序部分都可以放在多行上，而不需要使用续行符，因为它们清晰地表示出了定义的开始和结束。

缩进用于表示不同的代码块，如函数、条件语句、循环和类的主体。代码块中首条语句的缩进量可以是任意的，但整个代码块中的缩进必须保持一致，例如：

```
if a:
    statement1

    # 缩进一致，正确
    statement2


else:
    statement3


      statement4

  # 缩进不一致，错误
```

如果函数、条件语句、循环或类的主体较短，只包含一条语句，就可以将其放在同一行上，例如：

```
if a:  statement1
else:  statement2
```

要表示一个空的主体或代码块，可使用pass 语句，例如：

```
if a:
    pass
```

```
else:
    statements
```

尽管可以用制表符进行缩进，但并不鼓励这样做。Python编程社区的首选是用空格（而且也鼓励你这样做）。遇到制表符时，系统会将其转换为移到下一个为8倍数的列所需的空格量（例如，如果一个制表符出现在第11列中，那么系统会填充足够多的空格以移到第16行）。如果运行Python时使用`-t` 选项，那么在同一程序代码块中发现有制表符和空格混用的情况时，就会显示警告信息。使用`-tt` 选项可将这些警告信息转换为`TabError` 异常。

要在一行上放置多条语句，可以使用分号（`;`）隔开各条语句。如果一行上只有一条语句，也可以使用分号结尾，但这是不必要的。

`#` 字符表示此行的内容都是注释，但出现在引号字符串中的`#` 号无此作用。

最后，除非是在交互模式下运行，否则解释器将忽略所有空白行。在交互模式下，如果输入一条多行语句，空白行即表示输入结束。

2.2 标识符和保留字

标识符是用来识别变量、函数、类、模块和其他对象的名称。标识符可以包含字母、数字和下划线（`_`），但必须以非数字字符开始。字母目前只允许使用ISO-Latin字符集中的字符A～Z和a～z。由于标识符是区分大小写的，所以`FOO` 和`foo` 是两个不同的标识符。诸如$、%和@等的特殊符号不允许出现在标识符中。另外，像`if` 、`else` 和`for` 这样的单词是保留字，也不能用作标识符名称。下面的列表显示了所有保留字：

```
and          del          from         nonlocal          try
as           elif         global       not               while
assert       else         if           or                with
break        except       import       pass              yield
class        exec         in           print
continue     finally      is           raise
def          for          lambda       return
```

以下划线开始或结束的标识符通常具有特殊含义。例如，以一个下划线开始的标识符（如`_foo` ）不能使用`from module import *` 语句导入。前后均带有双下划线的标识符（如`__init__` ）是为特殊方法保留的，而只有前面带有双下划线的标识符（如`__bar` ）则用于实现私有的类成员，这一点将在第7章介绍。一般用途的标识符应避免使用以上几种格式。

2.3 数字字面量

内置的数字字面量分为4种类型：

- 布尔值
- 整数

- 浮点数
- 复数

标识符True 和False 会被解释为布尔值，其整数值分别是1和0。像1234 这样的数字会被解释为十进制整数。要使用八进制、十六进制或二进制指定整数，可以在值的前面分别加上0 、0x 或0b （如0644 、0x100fea8 或0b11101010 ）。

在Python中，整数的位数是任意的，所以，如果要指定一个非常大的整数，只需写出所有位数，如1234568901234567890 。但是在检查值和查看过去的Python代码时，可能会看到后面加上字母l（小写的L）或L 字符的大数字，如12345678901234567890L 。这个结尾处的L 表示Python会根据值的大小，选择将整数内部表示为固定精度的机器整数或任意精度的长整数类型。在老版本的Python中，可以显式地选择使用任一种类型，并且可以在数字结尾加上字母L 表示这是长类型。如今，这种区分已经没有必要，不鼓励使用。因此，表示大整数值时不必加上L 。

像123.34 和1.2334e+02 这样的数字会被解释为浮点数。整数或浮点数后面加上j 或J 就构成了虚数，如12.34J 。用一个实数加上一个虚数就构成了复数，方法是将实数和虚数加起来，如1.2+12.34J 。

2.4 字符串字面量

字符串字面量用于指定一个字符序列，其定义方法是把文本放入单引号（' ）、双引号（" ）或三引号（''' 或""" ）中。这三种引号形式在语义上没有差别，但要求在字符串开始和结尾使用的引号类型必须相同。置于单引号和双引号中的字符串必须定义在一行上，而三引号的字符串可以分布在多行上，并且会将所有格式符号（即换行符、制表符、空格等）包含在内。像"hello" 'world' 这样的相邻字符串（由空格、换行符或续行符隔开）将被连接起来，形成一个字符串"helloworld"。

在字符串字面量中，反斜杠（\ ）字符用于转义特殊字符，如换行符、反斜杠本身、引号和非打印字符。表2-1列出了可识别的一些转义码。无法识别的转义符序列将保持原样，包括最前面的反斜杠在内。

表**2-1** 标准的字符转义码

字　　符	描　　述
\	续行符
\\	反斜杠
\'	单引号
\"	双引号

\a	Bell（音箱发出提示音）
\b	退格符
\e	Escape
\0	Null（空值）
\n	换行符
\v	垂直制表符
\t	水平制表符
\r	回车符
\f	换页符
\ooo	八进制值（\000～\377 ）
\uxxxx	Unicode字符（\u0000～\uffff ）
\Uxxxxxxxx	Unicode字符（\U00000000～\Uffffffff ）
\N{字符名称}	Unicode字符名称
\xhh	十六进制值（x00～xff ）

转义码\OOO 和\x 用于将字符嵌入到很难输入的字符串字面量（如控制码、非打印字符、符号、国际字符等）中。对于这些转义码，你必须指定对应于字符值的整数值。例如，若要输入单词Jalapeño的字符串字面量，可以输入"Jalape\xf1o"，其中的\xf1 就是ñ的字符代码。

在Python 2中，字符串字面量对应于8位字符或面向字节的数据。这种字符串有一个很严重的缺陷，即它们无法完全支持国际字符集和Unicode。为了解决这个问题，Python 2对Unicode数据使用了单独的字符串类型。要输入Unicode字符串字面量，应在第一个引号前加上前缀"u"，例如：

```
s = u"Jalape\u00f1o"
```

在Python 3中不必加这个前缀字符（而且如果加上会算作语法错误），因为所有字符串已经使用了Unicode编码。如果使用-U 选项运行解释器，Python 2将会模拟这种行为。（即所有字符串字面量将被作为Unicode字符对待，u 前缀可以省略。）

无论你使用哪个Python版本，表2-1中的\u 、\U 和\N 转义码都可用于在Unicode字面量中插入任意字符。每个Unicode字符都有一个指定的码点（code point），在Unicode字符集中一般表示为U+*XXXX* ，其中 *XXXX* 是由4个或更多个十六进制数字表示的序列。（注意，这种表示法并非Python语法，但作者们在描述Unicode字符时经常使用它。）例如，字符ñ的码点是U+00F1 。\u 转义码用于插入码点范围在U+0000 和U+FFFF 之间的Unicode字符（如\u00f1 ）。\U 转义码用于插入码点范围在U+10000 及以上的字符（如\U00012345 ）。使用\U 转义码时请注意，码点在U+10000 以上的Unicode字符通常被分解为一对字符，称为代理编码对（surrogate pair）。这与Unicode字符串的内部表示有关，第3章将会对此进行更详细的介绍。

Unicode字符还有一个描述性名称。如果知道名称，就可以使用\N{字符名称} 转义序列，例如：

```
s = u"Jalape\N{LATIN SMALL LETTER N WITH TILDE}o"
```

关于码点和字符名称的权威性参考，请参阅http://www.unicode.org/charts 。

另外，可以在字符串字面量前面加上r 或R ，如r'\d' 。这些字符串称为原始字符串，因为其中所有的转义字符都会原封不动地保留，也就是说，这种字符串包含的文本只表示其字面上的含义，包括反斜杠在内。原始字符串的主要用途是指定其中反斜杠字符是有实际含义的字面量。例如，指定配合re 模块一起使用的正则表达式，或者Windows计算机上的一个文件名（如r'c:\newdata \tests' ）。

原始字符串不能以单个反斜杠结尾，如r"\" 。在原始字符串中，如果前面\ 字符的数量是奇数个，\u*XXXX* 转义序列仍然会解释为Unicode字符。例如，ur"\u1234" 定义的是包含单个字符U+1234的原始Unicode字符串，而ur"\\u1234" 定义的则是包含7个字符的字符串，其中前两个字符是反斜杠，余下5个字符是字面量"u1234" 。此外，如上所示，在Python 2.2中，r 必须出现在原始Unicode字符串中的u 之后。在Python 3.0中，u 前缀是可选的。

不能使用对应于UTF-8或UTF-16等数据编码的原始字节序列来定义字符串字面量。例如，直接输入像'Jalape\xc3\xblo' 这样的原始UTF-8编码字符串，将会产生一个9个字符的字符串U+004A、U+0061、U+006C、U+0061、U+0070、U+0065、U+00C3、U+00B1、U+006F，这可能不是你想要的结果。因为在UTF-8中，多字节序列\xc3\xb1 代表一个字符U+00F1，而不是两个字符U+00C3和U+00B1。要将一个已编码的字节字符串指定为字面量，在第一个引号前加上"b" ，即b"Jalape\xc3\xblo" 。这样才能从字面上创建一个单字节的字符串。我们可以使用decode() 方法解码字节字面量的值，将这种表示法的字节字面量转换为一个普通的字符串。关于这方面的更多细节，将在第3章和第4章中介绍。

字节字面量在大多数程序中都极少使用，因为这种语法直到Python 2.6才出现，而且在该版本中，字节字面量和普通字符串之间没有差别。但在Python 3中，字节字面量变成

了与普通字符串不同的新的bytes 数据类型（参见附录A）。

2.5 容器

将一些值放在方括号[...]、圆括号(...)和花括号{...}内，可分别表示一个列表、元组和字典中包含的对象集合，如下所示：

```
a = [ 1, 3.4, 'hello' ]        # 一个列表
b = ( 10, 20, 30 )             # 一个元组
c = { 'a': 3, 'b': 42 }        # 一个字典
```

列表、元组和字典字面量可以在不使用续行符（\）的情况下分布在多行上。另外，最后一项后面允许跟一个逗号，例如：

```
a = [ 1,
      3.4,
      'hello',
    ]
```

2.6 运算符、分隔符及特殊符号

Python可以识别的运算符有：

```
+    -   *   **   /   //   %    <<   >>     &     |
^    ~    <   >    <=  >=   ==  !=   <>     +=
-=   *=  /=  //=  %=  **=  &=  |=   ^=     >>=  <<=
```

以下标记可以用作表达式、列表、字典和语句不同部分的分隔符：

```
( ) [ ] { }  ,  :  .  `  =  ;
```

例如，等号（=）在赋值语句中可用作名称和值之间的分隔符，而逗号（,）则可用于分隔函数的各个参数，列表和元组中的各个元素等。小数点（.）可用在浮点数中以及在扩展切片操作中组成省略号（...）。

最后，语句中也会用到下面这些特殊符号：

```
'   "   #   \   @
```

字符$ 和? 在Python中没有任何意义，不能出现在程序中，但可以出现在带引号的字符串字面量中。

2.7 文档字符串

如果模块、类或函数定义的第一条语句是一个字符串，该字符串就成为了相关对象的文档字符串，如下所示：

```
def fact(n):
    "This function computes a factorial"
    if (n <= 1): return 1
    else: return n * fact(n - 1)
```

代码浏览工具和文档生成工具有时会用到文档字符串。通过对象的__doc__ 属性可以访问文档字符串，如下所示：

```
>>> print fact.__doc__


This function computes a factorial
>>>
```

文档字符串的缩进必须与定义中的所有其他语句保持一致。另外，文档字符串不能通过表达式进行计算或者通过变量进行赋值。文档字符串必须是包含在引号中的字符串字面量。

2.8 装饰器

函数、方法或类定义的前面可以使用一个特殊的符号，称为装饰器，其目的是修改后面定义的行为。装饰器使用@符号表示，必须单独放在对应的函数、方法或类之前的那行上，例如：

```
class Foo(object):
   @staticmethod
   def bar():
       pass
```

可以使用多个装饰器，但每个装饰器必须各占一行，例如：

```
@foo
@bar
def spam():
    pass
```

第6章和第7章中将介绍有关装饰器的更多内容。

2.9 源代码编码

编写Python源程序时一般使用标准的7位ASCII码。但是，在Unicode环境中工作的用户可能会发现这很别扭——特别是当他们必须使用国际字符编写大量字符串字面量时。

只需在Python程序的第1行或第2行中包含一个特殊的编码注释，就可以使用不同的编码编写Python源代码：

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

s = "Jalapeño"   # 引号中的字符串直接使用UTF-8编码。
```

程序中包含特殊的`coding:`注释语句时，可以直接使用支持Unicode的编辑器输入字符串字面量。但是，Python的其他元素（包括标识符名称和保留字）还是只能使用ASCII字符。

第3章　类型与对象

Python程序中保存的所有数据都是围绕对象 这个概念构建的 。对象包括基本的数据类型，如数字、字符串、列表和字典。但也可以通过类的形式创建用户定义的对象。另外，与程序结构和解释器内部操作相关的大多数对象也被暴露出来。本章将介绍Python对象模型的内部工作原理，并概述内置的数据类型。第4章将进一步介绍运算符和表达式。第7章将介绍如何创建用户定义的对象。

3.1　术语

程序中存储的所有数据都是对象。每个对象都有一个标识、一个类型（也叫作它的类）和一个值。例如，执行`a = 42`这行代码时，就会创建一个值为42的整数对象。对象的身份 可以看作是指向它在内存中所处位置的指针，而a就是引用这个具体位置的名称。

对象的类型 也称作对象的类 ，描述了对象的内部表示及它支持的方法与操作。创建特定类型的对象时，有时也将该对象称为该类型的实例 。实例被创建之后，它的标识和类型就不可改变。如果对象的值是可以修改的，则称为可变对象 （mutable）。如果对象的值不可以修改，则称为不可变对象 （immutable）。如果某个对象包含对其他对象的引用，则将其称为容器 （container）或集合 （collection）。

大多数对象的特征是拥有大量数据属性和方法。属性 （attribute）就是与对象相关的值。方法 （method）就是被调用时将在对象上执行某些操作的函数。使用点（`.`）运算符可以访问属性和方法，如下所示：

```
a = 3 + 4j        # 创建一个复数
r = a.real        # 获得实部（属性之一）

b = [1, 2, 3]     # 创建一个列表
b.append(7)       # 使用append方法添加一个新元素
```

3.2　对象标识与类型

内置函数`id()`可返回一个对象的标识，返回值为整数。这个整数通常对应该对象在内存中的位置，尽管这与Python的具体实现有关，而且不应该对标识作如此的解释。`is`运算符用于比较两个对象的标识。内置函数`type()`则返回一个对象的类型。下面的例子说明了比较两个对象的不同方式：

```
# 比较两个对象
def compare(a,b):
    if a is b:
        # a和b是同一个对象
        statements


    if a == b:
```

```
        # a和b具有相同的值
        statements


    if type(a) is type(b):
        # a和b具有相同类型
        statements
```

对象的类型本身也是一个对象，称为对象的类。该对象的定义是唯一的，而且对于某个类型的所有实例都是相同的。因此，类型之间可以使用`is` 运算符进行比较。所有类型对象都有一个指定的名称，可用于执行类型检查。这些名称中大部分都是内置的，如`list`、`dict` 和`file`，例如：

```
if type(s) is list:
    s.append(item)

if type(d) is dict:
    d.update(t)
```

因为可以通过定义类对类型进行特殊化，所以检查类型的更佳方式是用内置函数`isinstance (`*`object , type`*` )`，例如：

```
if isinstance(s,list):
    s.append(item)

if isinstance(d,dict):
    d.update(t)
```

因为`isinstance()` 函数能够辨别继承，所以它是检查所有Python对象类型的首选方式。

尽管在程序中可以增加类型检查，但类型检查通常不像你想象得那么有用。首先，过多的检查会严重影响性能。其次，程序不可能始终定义完全符合继承层次结构的对象。例如，如果前面的`isinstance(s, list)` 语句的用途是测试`s` 是否属于类似`list` 类型，则它对于拥有与`list` 相同编程接口但并不是从内置`list` 类型直接继承而来的对象不起作用。给程序添加类型检查的另一个方法是定义抽象基类（abstract base class）。这一点将在第7章中介绍。

3.3 引用计数与垃圾回收

所有对象都有引用计数。给一个对象分配一个新名称，或是将其放入一个容器（如列表、元组或字典），都会增加该对象的引用计数，例如：

```
a = 37        # 创建一个值为37的对象
b = a         # 增加37的引用计数
```

```
c = []
c.append(b)   # 增加37的引用计数
```

这个例子创建了一个包含值37的对象，a 只是引用这个新创建的对象的一个名称。将a 赋值给b 时，b就成了同一对象的新名称，该对象的引用计数因此增加。类似地，将b 放到一个列表中时，该对象的引用计数将再次增加。在这个例子中，至始至终只有一个包含37的对象，所有其他操作都只是创建了对该对象的新引用。

使用del 语句或者引用超出作用域（或者被重新赋值）时，对象的引用计数就会减少，例如：

```
del a       # 减少37的引用计数
b = 42      # 减少37的引用计数
c[0] = 2.0  # 减少37的引用计数
```

使用sys.getrefcount() 函数可以获得对象的当前引用计数，例如：

```
>>> a = 37


>>> import sys


>>> sys.getrefcount(a)


7
>>>
```

多数情况下，引用计数比你猜测得要大得多。对于不可变数据（如数字和字符串），解释器会主动在程序的不同部分共享对象，以便节约内存。

当一个对象的引用计数归零时，它将被垃圾回收机制处理掉。但在某些情况下，在很多已不再使用的对象间可能存在循环依赖关系，例如：

```
a = { }
b = { }
a['b'] = b   # a包含对b的引用
b['a'] = a   # b包含对a的引用
del a
del b
```

在这个例子中，del 语句将会减少a 和b 的引用计数，并销毁用于引用底层对象的名称。然而，因为每个对象都包含一个对其他对象的引用，所以引用计数不会归零，对象也不会被销毁（从而导致内存泄漏）。为了解决这个问题，解释器会定期执行一个周期检测器，搜索不可访问的对象周期并删除它们。解释器在执行过程中会被分配越来越多的内存，在此过程中，会定期运行周期检测算法。使用gc 模块中的函数可以准确调整和控制

该算法的行为（参见第13章）。

3.4 引用与复制

在程序进行像a = b 这样的赋值时，就会创建一个对b 的新引用。对于像数字和字符串这样的不可变对象，这种赋值实际上创建了b 的一个副本。然而，对可变对象（如列表和字典）来说，这样赋值的效果大不一样，例如：

```
>>> a = [1,2,3,4]


>>> b = a

             # b是对a的引用
>>> b is a


True
>>> b[2] = -100

       # 修改b中的一个元素
>>> a

                  # 注意a也已经改变
[1, 2, -100, 4]
>>>
```

因为在这个例子中，a 和b 引用的是同一个对象，修改其中任意一个变量都会影响到另一个。为了避免这种情况，必须创建对象的副本而不是创建新引用。

对于像列表和字典这样的容器对象，可以使用两种复制操作：浅复制和深复制。浅复制 将创建一个新对象，但它包含的是对原始对象中包含的项的引用，例如：

```
>>> a = [ 1, 2, [3,4] ]


>>> b = list(a)

         # 创建a的一个浅复制
>>> b is a


False
>>> b.append(100)

       # 给b追加一个元素
>>> b


[1, 2, [3, 4], 100]
>>> a
```

```
                    # 注意a没有变化
[1, 2, [3, 4]]
>>> b[2][0] = -100

       # 修改b中的一个元素
>>> b


[1, 2, [-100, 4], 100]
>>> a

                    # 注意a中的变化
[1, 2, [-100, 4]]
>>>
```

在这个例子中，a 和b 是单独的列表对象，但它们包含的元素是共享的。因此，修改a 的一个元素也会修改b 中的对应元素。

深复制 将创建一个新对象，并且递归地复制它包含的所有对象。Python中没有内置操作可创建对象的深复制，但可以使用标准库中的copy.deepcopy() 函数完成该工作，如下例所示：

```
>>> import copy


>>> a = [1, 2, [3, 4]]


>>> b = copy.deepcopy(a)


>>> b[2][0] = -100


>>> b


[1, 2, [-100, 4]]
>>> a

                    # 注意a没有变化
[1, 2, [3, 4]]
>>>
```

3.5 第一类对象

Python中的所有对象都享受着“头等”（first class）待遇，称作第一类对象。这意味着所有能用标识符命名的对象都具有平等的身份。这还意味着所有能被命名的对象都可以当作数据处理。例如，下面给出了一个包含两个值的简单字典：

```
items = {
    'number' : 42
    'text' : "Hello World"
}
```

给这个字典添加一些特殊的项，便可看到对象的第一类本质，例如：

```
items["func"]  = abs              # 添加abs()函数
import math
items["mod"]   = math             # 添加一个模块
items["error"] = ValueError      # 添加一个异常类型
nums = [1,2,3,4]
items["append"] = nums.append    # 添加另一个对象的一个方法
```

在这个例子中，items 字典包含一个函数、一个模块、一个异常和另一个对象的一个方法。如果愿意，你可以使用items 的字典查询代替原始名称，代码依然有效，例如：

```
>>> items["func"](-45)

        # 执行abs(-45)
45
>>> items["mod"].sqrt(4)

# 执行math.sqrt(4)
2.0
>>> try:


...     x = int("a lot")


... except items["error"] as e:

# e等同于except ValueError
...     print("Couldn't convert")


...
Couldn't convert
>>> items["append"](100)

      # 执行nums.append(100)
>>> nums


[1, 2, 3, 4, 100]
>>>
```

Python中的一切都是第一类的，新手一般不太能完全意识到这一点。但利用这个特性可以写出非常紧凑灵活的代码。例如，假定要通过正确的类型转换把一行文本GOOG, 100, 490.10 转换为一个字段列表，聪明的做法是创建一个类型列表（类型属于第一类

对象），然后执行一些简单的列表处理操作：

```
>>> line = "GOOG,100,490.10"

>>> field_types = [str, int, float]

>>> raw_fields = line.split(',')

>>> fields = [ty(val) for ty,val in zip(field_types,raw_fields)]

>>> fields

['GOOG', 100, 490.10000000000002]
>>>
```

3.6 表示数据的内置类型

Python中大约有十几种内置数据类型，可用于表示程序中用到的大多数数据。表3-1显示了这些类型的主要分类。对于表中“类型名称”列内所列出的名称或表达式，可以使用`isinstance()`和其他类型相关函数检查该类型。某些类型只在Python 2中可用，表中已经清楚地指出了这一点。（在Python 3中，它们已经被废弃或合并到其他类型中。）

表**3-1** 表示数据的内置类型

类型分类	类型名称	描述
None	type(None)	null对象None
数字	int	整数
	long	任意精度的整数（仅在Python 2中使用）
	float	浮点数
	complex	复数
	bool	布尔值（True 或False ）
序列	str	字符串
	unicode	Unicode字符串（仅在Python 2中使用）
	list	列表
	tuple	元组
	xrange	xrange() 函数创建的整数范围（在Python 3中称为range ）
映射	dict	字典
集合	set	可变集合
	frozenset	不可变集合

3.6.1 None 类型

None 类型表示一个null对象（没有值的对象）。Python只提供了一个null对象，在程序中表示为None 。如果一个函数没有显式地返回值，则返回该对象。None 经常用作可选参数的默认值，以便让函数检测调用者是否为该参数实际传递了值。None 没有任何属性，在布尔表达式中求值时为False 。

3.6.2 数值类型

Python使用5种数值类型：布尔型、整数、长整数、浮点数以及复数。除了布尔值之外，所有数字对象都有正负。所有数值类型都是不可变的。

布尔值包括True 和False 两个值，分别映射为数值1和0。

整数表示范围在-2 147 483 648和2 147 483 647（在某些机器上这个范围可能更大）之间的所有整数。长整数表示任意范围的整数，只受可用内存大小的限制。虽然存在两种整数类型，但Python尽量让这种区别不明显（事实上在Python 3中，这两种类型已经统一为一种整数类型）。因此，尽管在现有Python代码中有时会看到对长整数的引用，这基本上是可以忽略的实现细节问题——只要对所有整数运算都使用整数类型即可。一个例外是在对整数值进行显式类型检查的代码中。在Python 2中，如果整数x 被升级为长整数，表达式isinstance(*x* , int) 将返回False 。

浮点数是用机器上浮点数的原生双精度（64 位）表示的。通常该精度符合IEEE 754的要求，提供大约17位数的精度和范围从-308到308的指数。这与C语言中的double 类型相同。Python不支持32位的单精度浮点数。如果程序需要精确控制空间和数字精度，可以考虑使用numpy扩展库（下载地址为http://numpy.sourceforge.net ）。

复数使用一对浮点数表示，复数 *z* 的实部和虚部分别使用 *z* .real 和 *z* .imag 访问。方法 *z* .conjugate() 用于计算 *z* 的复共轭（ *a* +*b* j 的共轭是 *a* -*b* j ）。

数值类型拥有许多的属性和方法，旨在简化涉及混合算术的运算。为了更好地与有理数（在fractions 模块中可以找到）兼容，整数使用了属性 *x* .numerator 和 *x* .denominator 。为了兼容复数，整数或浮点数*y* 拥有属性 *y* .real 和 *y* .imag ，以及方法 *y* .conjugate() 。使用 *y* .as_interger_ratio() 可将浮点数y 转换为表示分数的一对整数。方法 *y* .is_integer() 用于测试浮点数y 是否表示整数值。通过方法 *y* .hex() 和 *y* .fromhex() 可用低级二进制形式使用浮点数。

库模块中还定义了另外几种数值类型。decimal 模块支持更加泛化的十进制算术。fractions 模块增加了一个有理数类型。第14章将介绍这些模块。

3.6.3 序列类型

序列 表示索引为非负整数的有序对象集合，包括字符串、列表和元组。字符串是字符的序列，而列表和元组是任意Python对象的序列。字符串和元组是不可变的，而列表则支持插入、删除和替换元素。所有序列都支持迭代。

1．所有序列通用的操作

表3-2列出了所有序列类型都可以使用的运算符和方法。使用索引运算符 *s* [*i*] 可

以访问序列 *s* 的元素i，而使用切片运算符 *s* [*i* :*j*]或扩展切片运算符 *s* [*i* :*j* :*stride*]（这些操作将在第4章中介绍）可选中一个子序列。使用内置的len(*s*)函数可以返回任意序列的长度。使用内置的min(*s*)和max(*s*)函数可以返回一个序列的最小值和最大值。但是，这些函数只能用于可对其中元素排序的序列（一般为数字和字符串）。sum(*s*)可用于对 *s* 中的各项求和，但只限于数字数据。

表**3-2** 适用于所有序列的操作和方法

项　　目	描　　述
s [*i*]	返回一个序列的元素 *i*
s [*i* : *j*]	返回一个切片
s [*i* : *j* : *stride*]	返回一个扩展切片
lens(*s*)	*s* 中的元素数
min(*s*)	*s* 中的最小值
Max(*s*)	*s* 中的最大值
sum(*s* [, *initial*])	*s* 中各项的和
all(*s*)	检查 *s* 中的所有项是否为True
any(*s*)	检查 *s* 中的任意项是否为True

表3-3列举了可应用于可变序列（如列表）的其他运算符。

表**3-3** 适用于可变序列的操作

项　　目	描　　述
s [*i*] = *v*	序列项赋值
s [*i* : *j*] = *t*	切片赋值
s [*i* : *j* : *stride*] = *t*	扩展切片赋值

del *s* [*i*]	序列项删除
del *s* [*i* : *j*]	切片删除
del *s* [*i* : *j* : *stride*]	扩展切片删除

2. 列表

列表支持的方法如表3-4所示。内置函数list(*s*)可将任意可迭代类型转换为列表。如果 *s* 已经是一个列表，则该函数构造的新列表是 *s* 的一个浅复制。 *s* .append(*x*)方法可将新元素x 追加到列表结尾。 *s* .index(*x*)方法用于搜索列表中首次出现的 *x* 。如果没找到任何元素，就会引发一个ValueError 异常。同样， *s* .remove(*x*)方法用于删除列表中首次出现的 *x* ，如果该项不存在，就会引发一个ValueError 异常。 *s* .extend(*t*)方法通过将序列t 中的元素追加到列表 *s* 中，对 *s* 进行扩展。

表**3-4** 列表方法

方　法	描　述
list(*s*)	将 *s* 转换为一个列表
s .append(*x*)	将一个新元素 *x* 追加到 *s* 末尾
s .extend(*t*)	将一个新列表 *t* 追加到 *s* 末尾
s .count(*x*)	计算*s* 中*x* 的出现次数
s .index(*x* , [, *start* [, *stop*]])	当s[*i*] = = *x* .*start* 时返回最小的 *i* ，可选参数 *stop* 用于指定搜索的起始和结束索引
s .insert(*i* , *x*)	在索引 *i* 处插入 *x*
s .pop([*i*])	返回元素 *i* 并从列表中移除它。如果省略 *i* ，则返回列表中最后一个元素
s .remove(*x*)	搜索 *x* 并从 *s* 中移除它
s .reverse()	颠倒 *s* 中的所有元素的顺序
s .sort([*key* [, *reverse*]])	对 *s* 中的所有元素进行排序。 *key* 是一个钥匙函数。 *reverse* 是一个标志，表明以倒序对列表进行排序。 *key* 和 *reverse* 应该始终以关键字参数的形式指定

s .sort() 方法用于对一个列表中的元素进行排序，可以接受一个键函数和反转标志，这二者都必须作为关键字参数指定。键函数是在排序过程中，进行比较之前应用于每个元素的函数。如果指定了键函数，该函数应该使用一个元素作为输入，它的返回值将用于在排序时执行比较。需要执行特殊的排序操作时，例如，对一个字符串列表进行不区分大小写的排序时，指定键函数十分有用。使用 *s* .reverse() 方法可以颠倒列表中所有元素的顺序。sort() 和reverse() 方法都直接操作列表元素，并返回None 。

3．字符串

Python 2提供两种字符串对象类型。字节字符串是字节（包含8位数据）的序列。它们可以包含二进制数据和嵌入的NULL字节。Unicode字符串是未编码的Unicode字符序列，其内部表示是16位整数，因而最多有65 536个不同的字符值。Unicode标准最多支持100万个不同的字符值，但Python默认不支持这些额外的字符。相反，它们被编码为特殊的双字符（4 字节）序列，称为代理对 （surrogate pair）——其解释由应用程序决定。Python可以被编译为使用32位整数保存Unicode字符，但这项特性是可选的。如果启用这项特性，Python可以表示范围从U+000000到U+110000的所有Unicode值，此时所有Unicode相关的函数都将做相应调整。

字符串支持的方法如表3-5所示。尽管这些方法操作的都是字符串实例，但它们实际上不会修改底层的字符串数据。因此，诸如 *s* .capitalize()、 *s* .center() 和 *s* .expandtabs() 这样的方法始终返回一个新的字符串，而不会修改字符串 *s* 。如果字符串s 中的所有字符均满足测试条件，像 *s* .isalnum() 和 *s* .isupper() 这样的方法就会返回True 或False 。此外，如果字符串的长度为0，这些测试始终返回False 。

s .find()、 *s* .index()、 *s* .rfind() 和 *s* .rindex() 方法用于在 *s* 中搜索一个子字符串。所有这些函数都返回一个整数，其值代表该子字符串在 *s* 中的索引。另外，如果未找到子字符串，find() 方法将返回-1 ，而index() 方法会引发一个ValueError 异常。 *s* .replace() 方法可使用相应文本替换一个子字符串。要重点强调的是，所有这些方法都只能处理简单的子字符串。正则表达式模式匹配和搜索是使用re 库模块中的函数处理的。

s .split() 和 *s* .rsplit() 方法可通过分隔符将一个字符串划分为一个字段列表。 *s* .partition() 和 *s* .rpartition() 方法用于搜索一个分隔符子字符串，并将 *s* 分为3个部分：分隔符之前的文本，分隔符本身，以及分隔符之后的文本。

很多字符串方法都接受可选的 *start* 和 *end* 参数，其值为整数，用于指定 *s* 中起始和结束位置的索引。大多数情况下，这些值可以为负值，表示索引是从字符串结尾处开始计算的。

s .translate() 方法用于执行高级的字符替换操作，例如快速将所有控制字符抽离出字符串。它接受一个转换表作为参数，该转换表中包含原始字符串中的字符与结果字符之间的一对一映射。8位字符串的转换表是一个256个字符的字符串，而Unicode字符串的转换表可以是任意序列对象 *s* ，其中 *s* [*n*] 返回一个整数字符码，或对应于整数值 *n* 的Unicode字符。

s .encode() 和 *s* .decode() 方法用于在字符串数据与指定字符编码之间来回转换。它们接受一个编码名称作为输入，如'ascii' 、'utf-8' 或'utf-16' 。这些方法常用于将Unicode字符串转换为适合I/O操作的数据编码，第9章中会进一步介绍该内容。请

注意，在Python 3中，encode() 方法只能用于字符串，而decode() 方法只能用于字节数据类型。

s .format() 方法用于执行字符串格式化操作。它接受任意方式组合的位置参数与关键字参数。 *s* 中使用{*item* }表示的占位符将被适当的参数所替代。位置参数可以使用像{0} 和{1} 这样的占位符进行引用，而关键字参数可以使用{*name* } 这样的占位符进行引用，例如：

```
>>> a = "Your name is {0} and your age is {age}"


>>> a.format("Mike", age=40)


'Your name is Mike and your age is 40'
>>>
```

在特殊的格式化字符串中，{*item* } 占位符还可以包括简单的索引和属性查找。例如，占位符｛*item* [*n*]｝（ *n* 为数字）会对 *item* 进行一次序列查找。占位符｛*item* [*key*]｝（ *key* 为非数字字符串）会执行一次字典查找 *item* ["*key* "]。占位符{*item* .*attr* } 引用的是 *item* 的属性 *attr* 。在4.3节中将介绍fromat() 方法的更多细节。

表**3-5**　字符串方法

方　法	描　述
s .captitalize()	首字符变大写
s .center(*width* [, *pad*])	在长度为 *width* 的字段内将字符串居中。 *pad* 是填充字符
s .count(*sub* [, *start* [, *end*]])	计算指定子字符串 *sub* 的出现次数
s .decode([*encoding* [, *errors*]])	解码一个字符串并返回一个Unicode字符串（只能用于字节字符串）
s .encode([*encoding* [, *errors*]])	返回字符串的编码版本（只能用于Unicode字符串）
s .endswith(*suffix* [, *start* [, *end*]])	检查字符串是否以 *suffix* 结尾
s .expandtabs([*tabsize*])	用空格替换制表符

s .find(*sub* [, *start* [, *end*]])	找到指定子字符串 *sub* 首次出现的位置，否则返回-1
s .format(*args, **kwargs)	格式化 *s*
s .index(*sub* [, *start* [, *end*]])	找到指定子字符串 *sub* 首次出现的位置，否则报错
s .isalnum()	检查所有字符是否都为字母或数字
s .isalpha()	检查所有字符是否都为字母
s .isdigit()	检查所有字符是否都为数字
s .islower()	检查所有字符是否都为小写
s .isspace()	检查所有字符是否都为空白
s .istitle()	检查字符串是否为标题字符串（每个单词的首字母大写）
s .isupper()	检查所有字符是否都为大写
s .join(*t*)	用 *s* 作为分隔符连接序列t 中的字符串
s .ljust(width [, *fill*])	在长度为 *width* 的字符串内左对齐 *s*
s .lower()	转换为小写形式
s .lstrip([*chrs*])	删掉 *chrs* 前面的空白或字符
s .partition(*sep*)	用分隔符字符串 *sep* 划分字符串。返回一个元组(*head* , *sep* , *tail*)，如果未找到 *sep* ， 则返回(*s* , "","")
s . *replace* (*old* , *new* [, *maxreplace*])	替换一个子字符串
s .rfind(*sub* [, *start* [, *end*]])	找到一个子字符串最后一次出现的位置
s .rindex(*sub* [, *start* [, *end*]])	找到一个子字符串最后一次出现的位置，否则报错

s .rjust(*width* [, *fill*])	在长度为 *width* 的字符串内右对齐 *s*
s .rpartition(*sep*)	用分隔符字符串 *sep* 划分字符串 *s* ，但是从字符串的结尾处开始搜索
s .rsplit([*sep* [, *maxsplit*]])	用 *sep* 作为分隔符对字符串从后往前进行划分。maxsplit 是划分的最大次数。如果省略 *maxsplit* ，结果与split() 方法完全相同
s .rstrip([*chrs*])	删掉 *chrs* 尾部的空白或字符
s .split([*sep* [, *maxsplit*]])	用 *sep* 作为分隔符对字符串进行划分。 *maxsplit* 是划分的最大次数
s .splitlines([*keepends*])	将字符串分为一个行列表。如果 *keepends* 为1，则保留各行最后的换行符
s .startswith(*prefix* [, *start* [, *end*]])	检查一个字符串是否以 *prefix* 开头
s .strip([*chrs*])	删掉 *chrs* 开头和结尾的空白或字符
s .swapcase()	将大写转换为小写，小写转换成大写
s .title()	将字符串转换为标题格式
s .translate(table [, *deletechars*])	用一个字符转换表 *table* 转换字符串，删除 *deletechars* 中的字符
s .upper()	将一个字符串转换为大写形式
s .zfill(*width*)	在字符串的左边填充0，直至其宽度为 *width*

4. xrange() 对象

内置函数xrange([*i* ,] *j* [, *stride*]) 创建的对象表示一系列整数 *k* ， *k* 的范围是 *i* <= *k* < *j* 。第一个索引 *i* 和 *stride* 是可选的，默认值分别为0和1。访问xrange 对象时就会计算它的值，尽管xrange 对象很像序列，但实际上还是存在一些限制。例如，它不支持所有标准切片操作。这限制了xrange 只能用于少数场景，如简单循环的迭代。

应该指出，在Python 3中xrange() 已经更名为range() 。但是其操作方式与以上描述的完全一致。

3.6.4 映射类型

映射类型 表示一个任意对象的集合，这些对象通过另一个几乎是任意键值的集合进行索引。和序列不同，映射对象是无序的，可以通过数字、字符串和其他对象进行索引。映射是可变的。

字典是唯一内置的映射类型，可以看作是Python中的散列表或关联数组。任何不可变对象都可以用作字典的键值，如字符串、数字、元组等。列表、字典和元组包含可变对象，不能被用作键，因为字典类型要求键值是固定值。

要选择映射对象中的一项，可以使用键索引运算符 *m* [*k*]，其中 *k* 是一个键值。如果找不到键，就会引发KeyError 异常。len(*m*) 函数用于返回一个映射对象中包含数据项的数量。表3-6列出了字典的方法和操作。

表**3-6** 字典的方法和操作

项　　目	描　　述
len(*m*)	返回 *m* 中项的数量
m [*k*]	返回 *m* 中键 *k* 的项
m [*k*]= *x*	将 *m* [*k*] 的值设为 *x*
del *m* [*k*]	从 *m* 中删除 *m* [*k*]
k in *m*	如果 *k* 是 *m* 中的键，则返回True
m .clear()	删除 *m* 中的所有项
m .copy()	返回 *m* 的一个副本
m .fromkeys(*s* [, *value*])	创建一个新字典并将序列 *s* 中的所有元素作为新字典的键，而且这些键的值均为value
m .get(*k* [, *v*])	返回 *m* [*k*]，如果找不到m[*k*]，则返回 *v*
m .has_key(*k*)	如果 *m* 中存在键k，则返回True，否则返回False （已废弃，使用in 运算符替代。只适用于Python 2）
m .items()	返回由（ *key* , *value* ）对组成的一个序列

m .keys()	返回键值组成的一个序列
m .pop(*k* [, *default*])	如果找到 *m* [*k*]，则返回 *m* [*k*] 并从 *m* 中删除它；否则，如果提供了defult 的值，则返回这个值，如果没有提供，则引发 *KeyError* 异常
m .popitem()	从 *m* 中删除一个随机的（ *key* , *value* ）对，并把它返回为一个元组
m .setdefault(k [, v])	如果找到 *m* [*k*]，则返回 *m* [*k*]；否则返回 *v* ，并将 *m* [*k*] 的值设为 *v*
m .update(b)	将 *b* 中的所有对象添加到 *m* 中
m .values()	返回 *m* 中所有值的一个序列

表3-6中的大多数方法用于操作或获取字典的内容。 *m* .clear() 方法用于删除m 中的所有项。 *m* .update(b) 方法插入在映射对象 *b* 中找到的所有(*key* , *value*) 对，从而更新当前映射对象。 *m* .get(*k* [,*v*]) 方法用于获取一个对象，但如果这个对象不存在，则返回一个可选的默认值 *v* 。 *m* .setdefault(*k* [,*v*]) 方法与 *m* .get() 方法相似，只是如果对象不存在，在返回 *v* 的同时还会设置 *m* [*k*]= *v* 。如果省略v ，它的默认值为None 。 *m* .pop() 方法返回字典中的一项，同时将其删除。 *m* .popitem() 方法用于迭代删除一个字典的内容。

m .copy() 方法将为映射对象中包含的项目制作一份浅副本，然后把它们放到一个新的映射对象中。 *m* .fromkeys(*s* [, *value*]) 方法会用序列 *s* 中的项作为键创建一个新的映射对象。生成的映射对象与 *m* 属于同一类型。与所有这些键相关的值都被设为None ，除非用可选的 *value* 参数提供了另一个值。fromkeys() 方法被定义为一个类方法，因此使用类名也可以调用它，如dict.fromkeys() 。

m .items() 方法返回一个包含(*key* , *value*) 对的序列。 *m* .*keys* () 方法返回由所有键值组成的一个序列，而 *m* .values() 返回由所有值组成的一个序列。使用这些方法的结果时，应该假定唯一可执行的安全操作就是迭代。在Python 2中，结果是一个列表，但在Python 3中结果是一个迭代当前映射对象内容的迭代器。如果在代码中假定它是迭代器，那么代码一般可以兼容这两个Python版本。如果需要将这些方法的结果作为数据保存，可以把它们存储到一个列表中，从而制作一个副本，如items = list(*m* .items()) 。如果只是想要所有键的列表，可使用keys = list(*m*) 。

3.6.5 集合类型

集合 是不同元素的无序集合。与序列不同，集合不提供索引或切片操作。它们和字典也有所区别，即对象不存在相关的键值。放入集合的项必须是不可变的。集合分为两种类型：set 是可变的集合，而frozenset 是不可变的集合。这两类集合是用一对内置函数创建的：

```
s = set([1,5,10,15])
f = frozenset(['a',37,'hello'])
```

set() 和frozenset() 方法都通过迭代所提供参数的方式来填充集合。这两类集合提供的方法如表3-7所示。

表**3-7** 集合类型的方法和操作

项　　目	描　　述
len(*s*)	返回 *s* 中项的数量
s .copy()	制作 *s* 的一份副本
s .difference(*t*)	求差集。返回所有在 *s* 中，但不在 *t* 中的项
s .intersection(*t*)	求交集。返回所有同时在 *s* 和 *t* 中的项
s .isdisjoint(*t*)	如果 *s* 和 *t* 没有相同项，则返回True
s .issubset(*t*)	如果 *s* 是 *t* 的一个子集，则返回True
s .issuperset(*t*)	如果 *s* 是 *t* 的一个超集，则返回Ture
s .symmetric_difference(*t*)	求对称差集。返回所有在 *s* 或 *t* 中，但又不同时在这两个集合中的项
s .union(*t*)	求并集。返回所有在 *s* 或 *t* 中的项

s .difference(*t*)、 *s* .intersection(*t*)、 *s* .symmetric_difference(*t*)和 *s* .union(*t*)方法提供了对集合的标准数学操作。返回值和s 的类型相同（set 或frozenset） 。参数t 可以是支持迭代的任意Python对象，包括集合、列表、元组和字符串。这些集合操作也可以通过数学运算符实现，这一点在第4章中将会进一步讲述。

可变集合（set ）还另外提供了一些方法，如表3-8所示。

表**3-8** 可变集合类型的方法

项　　目	描　　述
s .add(*item*)	将 *item* 添加到 *s* 中。如果 *item* 已经在 *s* 中，则无任何效果

s .clear()	删除 *s* 中的所有项
s .difference_update(*t*)	从 *s* 中删除同时也在 *t* 中的所有项
s .discard(*item*)	从 *s* 中删除 *item* 。如果 *item* 不是 *s* 的成员，则无任何效果
s .intersection_update(*t*)	计算 *s* 与t 的交集，并将结果放入 *s*
s .pop()	返回一个任意的集合元素，并将其从s 中删除
s .remove(*item*)	从 *s* 中删除 *item* 。如果 *item* 不是 *s* 的成员，则引发KeyError 异常
s .symmetric_difference_update(*t*)	计算 *s* 与 *t* 的对称差集，并将结果放入 *s*
s .update(*t*)	将 *t* 中的所有项添加到 *s* 中。 *t* 可以是另一个集合、一个序列或者支持迭代的任意对象

所有这些操作都可以直接修改集合 *s* 。参数 *t* 可以是支持迭代的任意对象。

3.7 表示程序结构的内置类型

在Python中，函数、类和模块都可以当做数据操作的对象。表3-9列出了用于表示程序本身各种元素的类型。

表**3-9** 表示程序结构的内置**Python**类型

类型分类	类型名称	描　　述
可调用类型	types.BuiltinFunctionType	内置函数或方法
	type	内置类型和类的类型
	object	所有类型和类的祖先
	types.FunctionType	用户定义的函数
	types.MethodType	类方法
模块	types.ModuleType	模块
类	object	所有类型和类的祖先
类型	type	内置类型和类的类型

注意，在表3-9中object 和type 出现了两次，因为它们可作为函数进行调用。

3.7.1 可调用类型

可调用类型表示支持函数调用操作的对象。具有这种属性的对象有几种，包括用户定义的函数、内置函数、实例方法和类。

1．用户定义的函数

用户定义的函数 是指用`def` 语句或`lambda` 运算符在模块级别上创建的可调用对象，例如：

```
def foo(x,y):
    return x + y

bar = lambda x,y: x + y
```

用户定义的函数`f` 具有以下属性。

属　　性	描　　述
f . __doc__	文档字符串
f . __name__	函数名称
f . __dict__	包含函数属性的字典
f . __code__	字节编译的代码
f . __defaults__	包含默认参数的元组
f . __globals__	定义全局命名空间的字典
f . __closure__	包含与嵌套作用域相关数据的元组

在Python 2的老版本中，以上很多属性的名称都是`func_code` 、`func_defaults` ，诸如此类。表中列出的属性名称与Python 2.6和Python 3兼容。

2．方法

方法 是在类定义中定义的函数。有3种常见的方法——实例方法、类方法和静态方法：

```
class Foo(object):
    def instance_method(self,arg):
        statements
```

```
    @classmethod
    def class_method(cls,arg):
        statements


    @staticmethod
    def static_method(arg):
        statements
```

实例方法 是操作指定类的实例的方法。实例作为第一个参数传递给方法，根据约定该参数一般称为self 。类方法 把类本身当作一个对象进行操作。通过第一个参数cls 将类对象传递给类方法。静态方法 就是打包在类中的函数，它不能使用实例或类对象作为第一个参数。

实例方法和类方法都是由一个类型为types.MethodType 的特殊对象表示的。然而，理解这种特殊类型需要深入了解对象属性查找（.）的工作方式。在一个对象上查找内容的过程（.）与进行函数调用始终是两个独立的操作。调用方法时，这两种操作都会发生，但作为不同的步骤。下面这个例子说明了在Foo 的实例上调用f.instance_method(arg) 的过程：

```
f = Foo()                    # 创建一个实例
meth = f.instance_method   # 查找方法，注意这里没有()
meth(37)                     # 现在调用方法
```

在这个例子中，meth 称为绑定方法 （bound method）。绑定方法是可调用对象，它封装了函数（方法）和一个相关实例。调用绑定方法时，实例就会作为第一个参数（self）传递给方法。因此，可以把上述例子中的meth 看作一个整装待发的方法，只是尚未使用函数调用运算符() 进行调用。

方法查找也适用于类本身，例如：

```
umeth = Foo.instance_method    # 查找Foo上的方法instance_method
umeth(f,37)                    # 调用它，但要显式地提供self参数
```

在这个例子中，umeth 称为非绑定方法 （unbound method）。非绑定方法是封装了方法函数的可调用对象，但需要传递一个正确类型的实例作为第一个参数。在上述例子中，我们传递了Foo 的一个实例f 作为第一个参数。如果传递的对象类型错误，就会引发TypeError 异常，例如：

```
>>> umeth("hello",5)


Traceback (most recent call last):
  File "", line 1, in
TypeError: descriptor 'instance_method' requires a 'Foo' object but received a
```

```
'str'
>>>
```

在用户定义的类中，绑定方法和非绑定方法代表的对象类型都是types.MethodType，它只不过是普通函数对象外围的一个浅层包装器。为方法对象定义的属性如下所示。

属　　性	描　　述
m . __doc__	文档字符串
m __name__	方法名称
m . __class__	定义该方法的类
m . __func__	实现方法的函数对象
m . __self__	与方法相关的实例（如果是非绑定方法，则为None ）

Python 3的一项不太为人所知的特性是，非绑定方法不再由types.MethodType 封装。如果像上面的例子一样访问Foo.instance_method，得到的只是实现该方法的原始函数对象。此外，Python 3对于self 参数不再进行任何类型检查。

3. 内置函数与方法

对象types.BuiltinFunctionType 用于表示用C和C++实现的函数和方法。内置方法具有的属性如下所示。

属　　性	描　　述
b . __doc__	文档字符串
b . __name__	函数/方法名称
b . __self__	与方法相关的实例（如果是绑定方法）

对于像len() 这样的内置函数，__self__ 被置为None，表明函数没有绑定到任何特定对象。对于像x.append （x 是一个列表对象）这样的内置方法，__self__ 被置为x。

4. 可调用的类与实例

类对象和实例也可以当作可调用对象进行操作。类对象使用class 语句创建，并作为函数调用，以创建新实例。在这种情况下，函数的参数被传递给类的__inti()__ 方法，以便初始化新创建的实例。如果实例定义了一个特殊方法__call__()，它就能够模拟函数的行为。如果某个实例 *x* 中定义了这个方法，使用 *x* (*args*) 语句等同于调用方法 *x* .__call__(*args*)。

3.7.2 类、类型与实例

定义类时，类定义通常会生成一个type 类型的对象。例如：

```
>>> class Foo(object):

...    pass

...
>>> type(Foo)
```

下面列出了一个类型对象 *t* 的常用属性。

属　　性	描　　述
t . __doc__	文档字符串
t . __name__	类名称
t . __bases__	由基类构成的元组
t . __dict__	保存类方法和变量的字典
t . __module__	定义类的模块名称
t . __abstractmethods__	抽象方法名称的集合（如果不存在抽象方法，就是未定义）

创建一个对象实例时，实例的类型就是定义它的类，例如：

```
>>> f = Foo()
```

```
>>> type(f)

<class '__main__.Foo'>
```

下面列出了一个实例i 的特殊属性。

属　　性	描　　述
i . __class__	实例所属的类
i . __dict__	保存实例数据的字典

__dict__ 属性通常用于存储与一个实例相关的所有数据。进行像 *i* .attr = value 这样的赋值时，值就保存在这里。但如果用户定义的类使用__slots__ ，就会使用一种更加有效的内部表示，而实例也不会有__dict__ 属性。第7章将更加详细地介绍对象和Python对象系统的组织方式。

3.7.3 模块

模块 类型是一个容器，可保存使用import 语句加载的对象。例如，程序中出现语句 import foo 时，就会把名称foo 赋给相应的模块对象。模块定义了一个使用字典实现的命名空间，使用属性__dict__ 可以访问该字典。只要引用模块的属性（使用. 运算符），就会将其转换为一次字典查找。例如， *m* .*x* 等价于 *m* .__dict__["*x* "] 。类似地，像 *m* .*x* = *y* 这样给属性赋值等价于 *m* .__dict__["*x* "] = *y* 。模块的可用属性如下所示。

属　　性	描　　述
m . __dict__	与模块相关联的字典
m . __doc__	模块文档字符串
m . __name__	模块名称
m . __file__	用于加载模块的文件
m . __path__	完全限定包名，只在模块对象引用包时定义

3.8 解释器内部使用的内置类型

解释器内部使用的很多对象都已暴露给用户使用，包括跟踪对象、代码对象、帧对象、生成器对象、切片对象和省略号（Ellipsis），如表3-10所示。程序很少会直接操作这些对象，但它们在工具构建人员和框架设计人员那里可以派上用场。

表**3-10** 解释器内部的内置**Python**类型

类型名称	描　　述
types.CodeType	字节编译代码
types.FrameType	执行帧
types.GeneratorType	生成器对象
types.TracebackType	异常的栈跟踪
slice	由扩展切片生成
Ellipsis	在扩展切片中使用

3.8.1 代码对象

代码对象表示原始的、字节编译过的可执行代码，或者叫做字节码，通常由内置的compile()函数返回。代码对象类似于函数，只是它们不包含与定义代码的命名空间相关的任何上下文，代码对象也不保存关于默认参数值的信息。代码对象c 拥有的只读属性如下所示。

属　　性	描　　述
c .co_name	函数名称
c .co_argcount	位置参数个数（包括默认值）
c .co_nlocals	函数使用的局部变量个数
c .co_varnames	包含局部变量名称的元组

c .co_cellvars	包含嵌套函数所引用的变量名称的元组
c .co_freevars	包含嵌套函数所使用的自由变量名称的元组
c .co_code	表示原始字节码的字符串
c .co_consts	包含字节码所用字面量的元组
c .co_names	包含字节码所用名称的元组
c .co_filename	被编译代码所在文件的名称
c .co_firstlineno	函数的首行行号
c .co_lnotab	字符串编码字节码相对于行号的偏移
c .co_stacksize	所需的栈大小（包括局部变量）
c .co_flags	包含解释器标志的整数。如果函数使用"*args"语法来接受任意数量的位置参数，就置位2。如果函数支持使用"**kwargs"语法的任意关键字参数，就置位3。所有其他位均保留

3.8.2 帧对象

帧对象用于表示执行帧，多出现在跟踪对象中（下面即将介绍）。帧对象f 的只读属性如下所示。

属　　性	描　　述
f .f_back	上一个栈帧（对当前调用者而言）
f .f_code	正在执行的代码对象
f .f_locals	局部变量的字典
f .f_globals	全局变量的字典
f .f_builtins	内置名称的字典

f .f_lineno	行号
f .f_lasti	当前指令。这是f_code 字节码字符串的索引

以下属性可以修改（调试器和其他工具会用到它们）。

属　　性	描　　述
f .f_trace	在每行源代码起始处调用的函数
f .f_exc_type	最近出现的异常类型（只能用于Python 2）
f .f_exc_value	最近出现的异常值（只能用于Python 2）
f .f_exc_traceback	最近出现的异常跟踪（只能用于Python 2）

3.8.3 跟踪对象

出现异常时就会创建跟踪对象，它包含栈跟踪信息。进入异常处理程序后，可以使用sys.exc_info() 函数来获取栈跟踪信息。跟踪对象可用的只读属性如下所示。

属　　性	描　　述
t .tb_next	栈跟踪的下一级（朝着发生异常的执行帧方向深入）
t .tb_frame	当前级别的执行帧对象
t .tb_lineno	出现异常的行号
t .tb_lasti	当前级别中正在执行的指令

3.8.4 生成器对象

调用生成器函数（参见第6章）时就会创建生成器对象。只要有函数使用特殊的yield 关键字，就算定义了一个生成器函数。生成器对象身兼两个用途，一是迭代器，二是容器，其中包含生成器函数自身的相关信息。生成器函数可用的属性和方法如下所示。

属　性	描　述
g .gi_code	生成器函数的代码对象
g .gi_frame	生成器函数的执行帧
g .gi_running	显示生成器函数目前是否正在运行的整数
g .next()	执行函数，直到遇到下一条yield语句为止，并返回值（该方法在Python 3中叫做__next__ ）
g .send(value)	向生成器发送一个值。被传递的值是由生成器中的yield 语句返回的，该生成器将一直执行，直至遇到下一条yield 语句。send() 返回传递给该表达式中的yield 的值
g .close()	通过在生成器函数中引发GeneratorExit 异常来终止生成器。当生成器被垃圾回收机制回收时此方法会自动执行
g.throw(*exc* [, *exc_value* [, *exc_tb*]])	在生成器的当前yield 语句处引发一个异常。 *exc* 是异常类型， *exc_value* 是异常值，而 *exc_tb* 是一个可选的跟踪。如果结果表达式被捕捉到并进行处理，则返回已传递给下一条yield 语句的值

3.8.5 切片对象

切片对象用于表示在扩展切片语法中指定的切片，如 *a* [*i* :*j* :*stride*]、 *a* [*i* :*j* , *n* :*m*] 或 *a* [..., *i* :*j*]。使用内置的slice([*i* ,] *j* [,*stride*]) 函数也可以创建切片对象。切片对象的只读属性如下所示。

属　性	描　述
s . *start*	切片的下边界；如果省略则为None
s . *stop*	切片的上边界；如果省略则为None
s . *step*	切片的步长；如果省略则为None

切片对象还提供一个方法s.indices(*length*)。该函数接受一个长度参数，并返回一个元组(*start* , *stop* , *stride*)，用于表明如何将切片应用到该长度的一个序列，例如：

```
s = slice(10,20)    # 切片对象表示[10:20]
s.indices(100)      # 返回(10,20,1) –> [10:20]
```

```
s.indices(15)       # 返回(10,15,1) –> [10:15]
```

3.8.6 Ellipsis 对象

Ellipsis 对象用于表示索引查找[] 中省略号(...) 是否存在。通过内置名称 Ellipsis 可以访问这种类型的对象。它没有任何属性，其值True 。Python中没有任何内置类型使用了Ellipsis ，但如果要在自己创建的对象上的索引运算符[] 中构造高级功能，可能就要用到它。下面的代码说明了如何创建Ellipsis 对象并将其传递到索引运算符中：

```
class Example(object):
    def__getitem__ (self,index):
        print(index)
e = Example()
e[3, ..., 4]       # 调用e.__getitem__((3, Ellipsis, 4))
```

3.9 对象行为与特殊方法

Python中的对象通常根据它们的行为和实现的功能进行分类。例如，所有序列类型都分在一组，如字符串、列表和元组，就是因为它们都支持一组相同的序列操作，如 *s* [*n*]、len(*s*) 等。所有基本的解释器操作都是通过特殊的对象方法实现的。这些特殊方法的名称前后始终带有双下划线（__ ）。当程序执行时，这些方法都由解释器自动触发。例如，操作 *x* + *y* 被映射为内部方法 *x* .__add__(*y*)，而索引操作 *x* [*k*] 被映射为 *x* .__getitem__(*k*)。每种数据类型的行为完全取决于它实现的一组特殊方法。

用户定义的类可以定义行为类似于内置类型的新对象，只要提供本节所述特殊方法的恰当子集即可。另外，像列表和字典这样的内置类型可以通过重新定义一些特殊方法来进行自定义（通过继承）。

接下来的几节将介绍与各种解释器特性相关的特殊方法。

3.9.1 对象的创建与销毁

表3-11中的方法分别用于创建、初始化和销毁实例。调用__new__() 类方法可以创建实例。__init__() 方法用于初始化对象的属性，在创建新对象后将立即调用。需要销毁对象时调用__del__() 方法，只有该对象不再被使用时才会调用该方法。需要注意的是，语句del *x* 只会减少一个对象的引用计数，而不一定会导致该函数被调用。第7章详细介绍了关于这些方法的更多信息。

表**3-11** 创建和销毁对象的特殊方法

方　法	描　述
__new__(*cls* [,* *args* [,** *kwargs*]])	创建新实例时调用的类方法

__init__(*self* [,* *args* [,** *kwargs*]])	初始化新实例时调用
__del__(*self*)	销毁实例时调用

__new__() 和__init__() 方法用于创建和初始化新实例。调用A(args) 创建对象时，会将其转换为以下步骤：

```
x = A.__new__(A,args

)
is isinstance(x,A): x.__init__(args

)
```

在用户定义的对象中，很少定义__new__() 或__del__() 方法。__new__() 方法通常只定义在元类或继承自不可变类型之一（整数、字符串、元组等）的用户定义对象中。__del__() 方法只有存在某种关键资源管理问题的情况下才会定义，如释放锁定或关闭连接时。

3.9.2 对象字符串表示

表3-12中的方法用于创建一个对象的各种字符串表示。

表**3-12** 对象表示的特殊方法

方　　法	描　　述
__ format__(*self* , *format* _ *spec*)	创建格式化后的表示
__ repr__(*self*)	创建对象的字符串表示
__ str__(*self*)	创建简单的字符串表示

__repr__() 和__str__() 方法用于创建对象的简单字符串表示。__repr__() 方法通常返回一个表达式字符串，可对该字符串求值以重新创建对象。该方法还负责创建在交互式解释器中检查变量时看到的输出值，其调用者是内置的repr() 函数。下面给出了一个使用repr() 和eval() 的例子：

```
a = [2,3,4,5]     # 创建一个列表
s = repr(a)       # s = '[2, 3, 4, 5]'
b = eval(s)       # 将s变为一个列表
```

如果无法创建字符串表达式，习惯做法是让__repr__() 方法返回一个<…message…>形式的字符串，例如：

```
f = open("foo")
a = repr(f)        # a = "<open file 'foo', mode 'r' at dc030>"
```

__str__() 方法的调用者是内置str() 函数和与打印相关的函数。它与__repr__() 方法的区别在于，它返回的字符串更加简明易懂。如果该方法未定义，就会调用__repr__() 方法。

__format__() 方法的调用者是format() 函数或字符串的format() 方法。format_spec 参数是包含了格式规范的字符串。该字符串与format() 方法的format_spce 参数相同，例如：

```
format(x,"spec

")           # 调用x.__format__("spec

")
"x is {0:spec

}".format(x)  # 调用x.__format__("spec

")
```

格式规范的语法是任意的，可以根据对象进行自定义。不过也存在标准语法，第4章将做介绍。

3.9.3 对象比较与排序

表3-13中的方法可执行对象的简单测试。__bool__() 方法用于真值测试，返回值应该为True 或False 。如果该方法未定义，Python将调用__len__() 方法来确定对象的真值。__hash__() 方法定义在希望用作字典中键的对象上。如果两个对象比较后相等，作为返回值的整数应该完全相同。此外，可变对象不应定义该方法，因为对象的任何改动都将改变散列值，导致在后续的字典查找中无法定位对象。

表**3-13** 对象测试与散列的特殊方法

方　法	描　述
__ bool__(*self*)	为真值测试返回False 或True
__ hash__(*self*)	计算整数的散列索引

对象可以实现一个或多个关系运算符（`<`、`>`、`<=`、`>=` 和`!=`）。这些方法均接受两个参数，而且支持返回任意类型的对象，包括布尔值、列表或任意其他Python类型。例如，有数学包可能使用该方法对两个矩阵的元素进行比较，并返回矩阵结果。如果无法进行比较，这些函数也会引发异常。表3-14中列出了用于比较运算符的特殊方法。

表**3-14**　用于比较的方法

方　　法	结　　果
`__lt__(` *self* `,` *other* `)`	*self* `<` *other*
`__le__(` *self* `,` *other* `)`	*self* `<=` *other*
`__gt__(` *self* `,` *other* `)`	*self* `>` *other*
`__ge__(` *self* `,` *other* `)`	*self* `>=` *other*
`__eq__(` *self* `,` *other* `)`	*self* `==` *other*
`__ne__(` *self* `,` *other* `)`	*self* `!=` *other*

对象不必实现表3-14中的所有操作。然而，如果要使用==比较对象或者使用对象作为字典键，应该定义`__eq__()` 方法。如果要为对象排序或者使用诸如`min()` 或`max()` 之类的函数，必须要至少定义`__lt__()` 方法。

3.9.4　类型检查

表3-15中的方法可用于重新定义类型检查函数`isinstance()` 和`issubclass()` 的行为。这些方法最常见的应用是定义抽象的基类和接口，第7章对此有介绍。

表**3-15**　类型检查的方法

方　　法	结　　果
`__instancecheck__(` *cls* `,` *object* `)`	`isinstance(` *object* `,` *cls* `)`
`__subclasscheck__(` *cls* `,` *sub* `)`	`issubclass(` *sub* `,` *cls* `)`

3.9.5　属性访问

表3-16中的方法分别使用点（`.`）运算符和`del` 运算符读、写和删除对象的属性。

表**3-16**　针对属性访问的特殊方法

方　　法	描　　述
__ getattribute__(*self* , *name*)	返回属性 *self* . *name*
__getattr__(*self* , *name*)	如果通过常规属性查找未找到属性，返回属性 *self* .*name* ，无法计算属性则引发AttributeError 异常
__setattr__(*self* , *name* , *value*)	设置属性 *self* .*name* = *value* ，覆盖默认值
__delattr__(*self* , *name*)	删除属性 *self* .*name*

访问属性时始终会调用__getattribute__() 方法。如果找到属性则返回之，否则调用__getattr__() 方法。__getattr__() 方法的默认行为是引发AttributeError 异常。设置属性时始终会调用__setattr__() 方法，而删除属性时始终会调用__delattr__() 方法。

3.9.6　属性包装与描述符

属性操作有时候使用一个额外逻辑层来包装对象的属性，同时该逻辑层可以同3.9.5节中提过的获取、设置和删除操作进行交互，这一点很精妙。实现此类包装的方法是创建一个描述符 对象来实现表3-17中的一个或多个方法。记住，描述符是可选的，极少情况下才需要定义。

表**3-17**　描述符对象的特殊方法

方　　法	描　　述
__get__(*self* , *instance* , *cls*)	返回一个属性值，否则引发AttributeError 异常
__set__(*self* , *instance* , *value*)	将属性设为 *value*
__delete__(*self* , *instance*)	删除属性

描述符的__get__() 、__set__() 和__delete__() 方法用于与类和类型的__getattribute__() 、__setattr__() 和__delattr__() 方法进行交互。如果在用户自定义类的主体中放入一个描述符对象的实例，这种交互就会发生。在这种情况下，对于描述符属性的所有访问都将显式地调用描述符对象本身的相应方法。描述符一般用于实现对象系统的底层功能，包括绑定和非绑定方法、类方法、静态方法和特性。第7章中给出

了一些更加深入的例子。

3.9.7 序列与映射方法

如果对象要模拟序列和映射对象的行为，就要用到表3-18中的方法。

表**3-18** 序列与映射的方法

方　　法	描　　述
__len__(*self*)	返回 *self* 的长度
__getitem__(*self* , *key*)	返回 *self* [*key*]
__setitem__(*self* , *key* , *value*)	设置 *self* [*key*] = *value*
__delitem__(*self* , *key*)	删除 *self* [*key*]
__contains__(*self* , *obj*)	如果 *obj* 在 *self* 中，则返回True，否则返回False

例如：

```
a = [1,2,3,4,5,6]
len(a)              # a.__len__()
x = a[2]            # x = a.__getitem__(2)
a[1] = 7            # a.__setitem__(1,7)
del a[2]            # a.__delitem__(2)
5 in a              # a.__contains__(5)
```

内置的len() 函数调用__len__() 方法，返回一个非负的长度值。该函数还用于确定真值，除非已经定义了__bool__() 方法。

为了操作单个项，__getitem__() 方法可根据键返回项。这里的键可以是任意Python对象，但对于序列而言通常为整数。__setitem__() 方法用于给元素赋值。__delitem__() 方法在对单个元素进行del 操作时调用。__contains__() 方法用于实现in 运算符。

切片运算（如x = *s* [*i* :*j*]）也使用__getitem__()、__setitem__() 和__delitem__() 方法来实现。但给切片传递的键是一个特殊的 *slice* 对象。该对象拥有可描述所请求切片范围的属性，例如：

```
a = [1,2,3,4,5,6]
x = a[1:5]          # x = a.__getitem__(slice(1,5,None))
a[1:3] = [10,11,12] # a.__setitem__(slice(1,3,None), [10,11,12])
del a[1:4]          # a.__delitem__(slice(1,4,None))
```

Python的切片功能实际上比很多程序员认为的更强大。例如，它支持以下扩展切片的变体，在处理矩阵和数组这样的多维数据结构时可能非常有用：

```
a = m[0:100:10]           # 带步进的切片(步进值=10)
b = m[1:10, 3:20]         # 多维切片
c = m[0:100:10, 50:75:5]  # 带步长的多维切片
m[0:5, 5:10] = n          # 扩展切片分配
del m[:10, 15:]           # 扩展切片删除
```

扩展切片每个维度的一般格式是 *i* :*j* [:*stride*]， *stride* 是可选的。和普通切片一样，可以省略切片每个部分的开始或结束值。另外，省略号（写为...）可用于表示扩展切片中结束或开始的任意维数：

```
a = m[..., 10:20]  # 使用Ellipsis对象访问扩展切片
m[10:20, ...] = n
```

使用扩展切片时，__getitem__()、__setitem__()和__delitem__()方法分别用于实现访问、修改和删除操作。然而，传递给这些方法的值是一个包含slice或Ellipsis对象组合的元组，而非整数，例如：

```
a = m[0:10, 0:100:5, ...]
```

调用__getitem__()方法的方式如下：

```
a = m.__getitem__((slice(0,10,None), slice(0,100,5), Ellipsis))
```

Python字符串、元组和列表目前在一定程度上支持扩展切片，这一点将在第4章中介绍。特殊用途的Python扩展，特别是与科学相关的扩展，可能会提供新的类型和对象，从而为扩展切片操作提供高级支持。

3.9.8 迭代

如果对象 *obj* 支持迭代，它必须提供方法 *obj* .__iter__()，该方法返回一个迭代器对象。而迭代器对象 *iter* 必须实现一个方法 *iter* .next()（在Python 3中为 *iter* .__next__()），该方法返回下一个对象，或者在迭代结束时引发StopIteration异常。这两个方法均用于for语句的实现，以及其他一些隐式执行迭代的操作。例如，语句for x in s执行的步骤等同于以下代码：

```
_iter = s.__iter__()
while 1:
    try:
         x =_iter.next()(# Python 3中为_iter.__next__())
    except StopIteration:
         break
```

```
    # 在for循环体内执行语句
    ...
```

3.9.9 数学操作

表3-19列出了对象在模拟数字时必须实现的特殊方法。根据第4章中讲述的优先规则，数学操作总是从左至右进行求值。执行表达式 *x* + *y* 时，解释器会试着调用方法 *x*.__add__(*y*)。以字母r开头的特殊方法支持以反向的操作数进行运算，它们只在左操作数没有实现指定操作时被调用。例如，如果表达式 *x* + *y* 中的 *x* 不支持__add()__方法，解释器就会试着调用方法 *y*.__radd__(*x*)。

表**3-19** 数学操作的方法

方　法	结　果
__add__(*self* , *other*)	*self* + *other*
__sub__(*self* , *other*)	*self* - *other*
__mul__(*self* , *other*)	*self* * *other*
__div__(*self* , *other*)	*self* / *other*（仅在Python 2 中使用）
__truediv__(*self* , *other*)	*self* / *other*（Python 3）
__floordiv__(*self* , *other*)	*self* // *other*
__mod__(*self* , *other*)	*self* % *other*
__divmod__(*self* , *other*)	divmod(*self* , *other*)
__pow__(*self* , *other* [, *modulo*])	*self* ** *other* , pow(*self* , *other* , *modulo*)
__lshift__(s *elf* , *other*)	*self* << *other*
__rshift__(*self* , *other*)	*self* >> *other*
__and__(*self* , *other*)	*self* & *other*
__or__(*self* , *other*)	*self* |*other*

__xor__(self, *other*)	*self* ^ *other*
__radd__(*self* , *other*)	*other* + *self*
__rsub__(*self* , *other*)	*other* - *self*
__rmul__(*self* , *other*)	*other* * *self*
__rdiv__(*self* , *other*)	*other* / *self* （仅在Python 2 中使用）
__rtruediv__(*self* , *other*)	*other* / *self* （Python 3）
__rfloordiv__(*self* , *other*)	*other* // *self*
__rmod__(*self* , *other*)	*other* % *self*
__rdivmod__(*self* , *other*)	divmod(*other* , *self*)
__rpow__(*self* , *other*)	*other* ** *self*
__rlshift__(*self* , *other*)	*other* << *self*
__rrshift__(*self* , *other*)	*other* >> *self*
__rand__(*self* , *other*)	*other* & *self*
__ror__(*self* , *other*)	*other* |*self*
__rxor__(*self* , *other*)	*other* ^ *self*
__iadd__(*self* , *other*)	*self* += *other*
__isub__(*self* , *other*)	*self* -= *other*
__imul__(*self* , *other*)	*self* *= *other*
__idiv__(*self* , *other*)	*self* /= *other* （仅在Python 2中使用）
__itruediv__(*self* , *other*)	*self* /= *other* （Python 3）

__ifloordiv__(*self* , *other*)	*self* //= *other*
__imod__(*self* , *other*)	*self* %= *other*
__ipow__(*self* , *other*)	*self* **= *other*
__iand__(*self* , *other*)	*self* &= *other*
__ior__(*self* , *other*)	*self* |= *other*
__ixor__(*self* , *other*)	*self* ^= *other*
__ilshift__(*self* , *other*)	*self* <<= *other*
__irshift__(*self* , *other*)	*self* >>= *other*
__neg__(*self*)	- *self*
__pos__(*self*)	+ *self*
__abs__(*self*)	abs(*self*)
__invert__(*self*)	~ *self*
__int__(*self*)	int(*self*)
__long__(*self*)	long(*self*)（仅在Python 2 中使用）
__float__(*self*)	float(*self*)
__complex__(*self*)	complex(*self*)

方法__iadd()__和__isub()__等用于实现原地算术操作，如a+=b和a-=b（也称为增量赋值）。这些运算符与标准算术方法之间的区别在于，原地运算符的实现能够提供某种自定义，如性能优化。例如，如果 *self* 参数不是共享的，就可以原地修改对象的值，而不必为结果分配一个新创建的对象。

除法运算符共有3种，__div__()、__truediv()__和__floordiv__()，它们用于实现常规除法（/）和截断除法（//）操作。存在3种除法操作的原因是，在Python 2.2中整数除法的语义开始有了变化，而这种变化在Python 3中则变成了默认行为。在Python 2中，Python的默认行为是将/运算符映射到__div__()方法，如果操作数都为整数，这种

操作会把结果截断为一个整数。在Python 3中，除法被映射到__truediv__() 方法，对于整数操作数将返回一个浮点数。在Python 2中，后面这种行为是一项可选特性，在程序中包含语句from__future__ import division 即可启用该特性。

转换方法__int__()、__long__()、__float__() 和__complex__() 用于将对象转换为4种内置的数值类型之一。出现显式的类型转换时（如int() 和float()），就会调用这些方法。但这些方法不能用于在数学操作中隐式地强制类型转换。例如，表达式3 + *x* 会引发一个TypeError 错误，即使x 是定义了整数转换方法__int__() 的用户定义对象也是如此。

3.9.10 可调用接口

对象通过提供__call__(*self* [,**args* [, ***kwargs*]]) 方法可以模拟函数的行为。如果一个对象 *x* 提供了该方法，就可以像函数一样调用它。也就是说， *x* (*arg1* , *arg2* ,...) 等同于调用 *x* .__call__(*self* ,*arg1* ,*arg2* ,...) 。模拟函数的对象可以用于创建仿函数（functor）或代理（proxy）。下面给出了一个简单的例子：

```
class DistanceFrom(object):
    def__init__(self,origin):
        self.origin = origin
    def__call__(self, x):
        return abs(x - self.origin)

nums = [1, 37, 42, 101, 13, 9, -20]
nums.sort(key=DistanceFrom(10))       # 按照与10的距离进行排序
```

在这个例子中，DistanceFrom 类创建的实例模拟了一个单参数函数。这些实例可用于代替普通的函数，如本例中对于sort() 的调用。

3.9.11 上下文管理协议

with 语句支持在另一个称为上下文管理器 的对象的控制下执行一系列语句。它的语法如下所示：

```
with context

 [ as var

]:
     statements
```

其中 *context* 对象需要实现表3-20中所示的方法。执行with 语句时，就会调用__enter__() 方法。该方法的返回值将被放入由可选的as var 说明符指定的变量中。只要控制流离开与with 语句相关的语句块，就会立即调用__exit__() 方法。__exit__() 方法接收当前异常的类型、值和跟踪作为参数。如果没有要处理的错

误，所有3个值都将被置为None 。

表**3-20**　上下文管理器的特殊方法

方　　法	描　　述
__enter__(*self*)	进入新的上下文时调用该方法，其返回值将被放入由 *with* 语句的 *as* 说明符指定的变量中
__exit__(*self* , *type* , *value* , *tb*)	离开上下文时调用该方法。如果有异常出现， *type* 、 *value* 和 *tb* 的值分别为异常的类型、值和跟踪信息。上下文管理接口的首要用途是简化涉及系统状态（如打开文件、网络连接和锁定的对象）的对象的资源控制。实现该接口后，当执行离开使用对象的上下文时，该对象可以安全地释放资源。第5章将介绍这方面的细节

3.9.12　对象检查与dir()

dir() 函数通常用于检查对象。实现__dir__(self) 方法后，对象就可以使用dir() 返回名称列表。定义该方法可以更加方便地隐藏不想让用户直接访问的对象内部细节。但要记住，用户仍然可以检查实例和类的底层__dict__ 属性，从而了解已定义的所有内容。

第4章　运算符与表达式

本章介绍Python的内置运算符、表达式和求值规则。尽管本章大部分篇幅讲的都是Python的内置类型，但用户定义的对象也可以轻松地重新定义任意运算符，从而实现它们自己的行为。

4.1　数值操作

所有数值类型均可进行以下操作。

操　作	描　述	操　作	描　述
`x + y`	加法	`x ** y`	乘方（x^y）
`x - y`	减法	`x % y`	取模（$x \bmod y$）
`x * y`	乘法	`-x`	一元减法
`x / y`	除法	`+x`	一元加法
`x // y`	截断除法		

截断除法运算符`//`，也称为地板除法（floor division）。它把结果截取为一个整数，并且对整数和浮点数均有效。在Python 2中，如果操作数是整数，常规除法运算符（`/`）也会将结果截取为整数。因此，`7/4`的结果是`1`，而不是`1.75`。但这种行为在Python 3中有所变化，除法的结果为浮点数。取模运算符返回的是除法`x//y`的余数，例如`7 % 4`的结果是`3`。对于浮点数，取模运算符返回的是`x // y`的浮点数余数，也就是`x - (x // y) * y`。对于复数，取模（`%`）和截断除法运算符（`//`）是无效的。

以下移位运算符和按位逻辑运算符只能应用于整数。

操　作	描　述	操　作	描　述
`x << y`	左移	`x | y`	按位或
`x >> y`	右移	`x ^ y`	按位异或
`x & y`	按位与	`~x`	按位求反

按位运算符假定整数是以二进制补码形式表示的，而且符号位可以无限向左扩展。如果要使用会映射到本机硬件整数的原始位模式，就需要注意这一点。因为Python不会截取位，或者说允许值上溢，因此结果的数量级可能会变成任意大。

另外，以下内置函数支持所有数值类型。

操　作	描　述	操　作	描　述
abs(*x*)	绝对值	pow(*x* ,*y* [, *modulo*])	返回(*x* ** *y*) % modulo
divmod(*x* ,*y*)	返回(*x* // *y* , *x* % *y*)	round(*x* , [*n*])	四舍五入为接近的10^{-n} 的倍数（只返回浮点数）

abs() 函数返回数的绝对值。divmod() 函数返回除法操作的商和余数，只对非复数有效。pow() 函数可以用于代替** 运算符，但也支持三重取模运算（通常用在密码算法中）。round() 函数将一个浮点数 *x* 四舍五入为最近的10^{-n} 的倍数。如果省略 *n* ，它将被设为0 。如果 *x* 与两个倍数值的距离相等，Python 2就会把 *x* 四舍五入为距离0最近的倍数（例如，0.5 被舍入为1.0 ，而-0.5 被舍入为-1.0 ）。这里要注意一点，Python 3会将距离两个倍数值相等的值舍入为最近的偶数倍数（例如，0.5 被舍入为0.0 ，而1.5 被舍入为2.0 ）。对于要移植到Python 3的程序而言，这是一个要注意的可移植性问题。

以下比较运算符具有标准的数学解释，如果表达式为真则返回布尔值True ，为假则返回布尔值False 。

操　作	描　述	操　作	描　述
x < *y*	小于	*x* != *y*	不等于
x > *y*	大于	*x* >= *y*	大于等于
x == *y*	等于	*x* <= *y*	小于等于

比较运算可以连接在一起，如 *w* < *x* < *y* < *z* 。这类表达式的求值等价于 *w* < *x* and *x* < *y* and *y* < *z* 。x < y > z 这样的表达式是合法的，但很可能会把阅读代码的人搞糊涂（注意，这个表达式中的x 和z 之间并没有比较操作）。不允许对复数进行比较，这会引发TypeError 异常。

只有当操作数属于同一类型时，对这些操作数进行运算才是有效的。对于内置数值，

Python将强制进行类型转换，将一种类型转换为另一种类型，转换规则如下。

（1）如果操作数之一为复数，则将另一个操作数也转换为复数。

（2）如果操作数之一为浮点数，则将另一个操作数也转换为浮点数。

（3）否则，两个操作数肯定同时为整数，不需要进行转换。

对于用户定义的对象，如果用于涉及多种操作数的表达式，其行为取决于对象的实现。一般来说，解释器不会尝试执行任何隐式类型转换。

4.2 序列操作

序列类型（包括字符串、列表和元组）支持的运算符如下所示。

操　作	描　述	操　作	描　述
s + *r*	连接	for *x* in *s* :	迭代
s * *n* ,*n* * *s*	制作 *s* 的 *n* 个副本， *n* 为整数	all(*s*)	如果 *s* 中的所有项都为True，则返回True
v1 ,*v2* ...,*vn* = *s*	变量解包（unpacking）	any(*s*)	如果 *s* 中的任意项为True，则返回True
s [*i*]	索引	len(*s*)	长度
s [*i* :*j*]	切片	min(*s*)	*s* 中的最小项
s [*i* :*j* :*stride*]	扩展切片	max(*s*)	*s* 中的最大项
x in *s* ,*x* not *in* *s*	成员关系	sum(*s* [, *initial*])	序列所有项之和可设置一个初始值

+运算符用于连接相同类型的两个序列。 *s* * *n* 运算符制作一个序列的 *n* 个副本。但是，这些副本都是仅仅复制了元素引用的浅复制，如以下代码所示：

```
>>> a = [3,4,5]

>>> b = [a]

>>> c = 4*b
```

```
>>> c


[[3, 4, 5], [3, 4, 5], [3, 4, 5], [3, 4, 5]]
>>> a[0] = -7


>>> c
[[-7, 4, 5], [-7, 4, 5], [-7, 4, 5], [-7, 4, 5]]
>>>
```

注意，修改a 导致列表c 中的每个元素也被修改了。在这个例子中，b 中放入的是对列表a 的引用。复制b 时，就创建了另外4个对a 的引用。最后，当a 被修改后，这种改动被传递给a 的所有其他“副本”。序列操作的这种行为通常是意料之外的，并不符合程序员的本意。解决该问题的方式之一是复制a 的内容，手动构造要复制的序列，例如：

```
a = [ 3, 4, 5 ]
c = [list(a) for j in range(4)]  # list()制作列表a的一个副本
```

标准库中的copy 模块也可以用于对象复制。

所有序列都可以被解包为一列变量名称，例如：

```
items = [ 3, 4, 5 ]
x,y,z = items      # x = 3, y = 4, z = 5

letters = "abc"
x,y,z = letters      # x = 'a', y = 'b', z = 'c'

datetime = ((5, 19, 2008), (10, 30, "am"))
(month,day,year),(hour,minute,am_pm) = datetime
```

将值解包到变量中时，变量的个数必须严格匹配序列中元素的个数。另外，变量的结构也必须匹配序列的结构。例如，示例代码的最后一行将值解包到6个变量中，这6个变量又组成各自包含3个元素的2个元组，而这正好是右边序列的结构。将序列解包到多个变量中适用于任意种类的序列，包括迭代器和生成器创建的序列。

索引运算符s[*n*] 返回序列中的第 *n* 个对象，其中 *s* [0] 是第一个对象。使用负数索引可以从序列尾部开始获取字符。例如， *s* [-1] 返回的是最后一项。此外，试图访问超出边界的元素将引发IndexError 异常。

切片运算符s[*i:j*] 从 *s* 中提取一个子序列，它所包含的元素索引k 的范围是 *i <= k <j* 。 *i* 和 *j* 都必须是整数或长整数。如果忽略开始或结束的索引，Python就会假定它们的默认值分别是序列的开始和结尾。切片运算符支持使用负数索引，并且假定它

关联到序列的结尾。如果i 或j 超出范围，将假定它们引用序列的开始或结尾，这取决于它们的值所引用的元素是位于第一项之前还是最后一项之后。

可以为切片运算符指定一个可选的步进值，表示方法为 *s[i:j:stride]* ，这会让切片跳过一些元素。然而，这种行为从某种程度上说更加微妙。如果提供步进值， *i* 是起始索引， *j* 是结束索引，那么生成的子序列就是元素 *s* [*i*]、 *s* [*i* +*stride*] 、 *s* [*i* +2**stride*] 等，直至到达索引值 *j* 为止（ *j* 不包含在内）。步进值也可以为负数。如果省略起始索引 *i* ，当 *stride* 为正数时它被置为序列的开始，而当 *stride* 为负数时则被置为序列的结尾。如果省略结束索引 *j* ，当 *stride* 为正数时它被置为序列的结尾，而当 *stride* 为负数时则被置为序列的开始。下面举例说明了这一点：

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

b = a[::2]       # b = [0, 2, 4, 6, 8 ]
c = a[::-2]      # c = [9, 7, 5, 3, 1 ]
d = a[0:5:2]     # d = [0,2]
e = a[5:0:-2]    # e = [5,3,1]
f = a[:5:1]      # f = [0,1,2,3,4]
g = a[:5:-1]     # g = [9,8,7,6]
h = a[5::1]      # h = [5,6,7,8,9]
i = a[5::-1]     # i = [5,4,3,2,1,0]
j = a[5:0:-1]    # j = [5,4,3,2,1]
```

x in *s* 运算符测试对象 *x* 是否在序列 *s* 中，返回值为True 或False 。类似地， *x* not in *s* 运算符测试 *x* 是否不在序列 *s* 中。对于字符串对象，in 和not in 运算符接受子字符串。例如，'hello' in 'hello world' 将返回True 。需要注意的是，in 运算符不支持通配符或任意类别的模式匹配。为此，需要使用像re 模块这样的库模块才能利用正则表达式模式。

for x in s 运算符用于迭代序列的所有元素，第5章中将进一步介绍这一点。len(s) 返回一个序列中元素的个数。min(*s*) 和max(*s*) 分别返回一个序列中的最小值和最大值，但它们只有在能够使用< 运算符对序列中的元素进行排序时才有意义（例如，如果对一个文件对象列表求最大值，就没有什么意义）。sum(*s*) 用于对s 中的所有项求和，但通常只在其中的各项代表数时有效。可以给sum() 指定一个可选的初始值，这个值的类型通常决定了结果的类型。例如，如果使用sum(*items* , decimal.Decimal(0)) ，结果就是一个decimal 对象（参见第14章，了解有关decimal 模块的更多信息）。

字符串和元组是不可变的，创建后不能修改。列表可通过以下运算符进行修改。

操　作	描　述	操　作	描　述
s [*i*] = x	索引赋值	del *s* [*i*]	删除一个元素
s [*i* :*j*] = *r*	切片赋值	del *s* [*i* :*j*]	删除一个切片

s [*i* :*j* :*stride*] = *r*	扩展切片赋值	del *s* [*i* :*j* :*stride*]	删除一个扩展切片

s [*i*] = *x* 运算符将列表的元素 *i* 修改为引用对象 *x* ，这将增加x 的引用计数。负索引值将关联到列表结尾，而试图给超出范围的索引赋值将引发IndexError 异常。切片赋值运算符 *s* [*i*:*j*] = *r* 将使用序列 *r* 中的元素替换元素 *k* ，其中 *k* 的范围是 *i* <= *k* < *j* 。切片的两个索引值可以相同，而且如果它们超出范围，将被调整为列表的开始或结尾。如果有必要，将对序列 *s* 进行扩展或收缩，以便容纳 *r* 中的所有元素，例如：

```
a = [1,2,3,4,5]
a[1] = 6             # a = [1,6,3,4,5]
a[2:4] = [10,11]     # a = [1,6,10,11,5]
a[3:4] = [-1,-2,-3]  # a = [1,6,10,-1,-2,-3,5]
a[2:] = [0]          # a = [1,6,0]
```

切片赋值可以提供一个可选的步进参数。但这种行为受到的限制更大，因为右边的参数必须与要替换切片的元素个数完全相同，例如：

```
a = [1,2,3,4,5]
a[1::2] = [10,11]            # a = [1,10,3,11,5]
a[1::2] = [30,40,50]         # ValueError异常。左边的切片中只有两个元素
```

del *s* [*i*] 运算符从列表中删除元素 *i* ，同时减少它的引用计数。del *s* [*i*:*j*] 删除切片内的所有元素。切片删除也可以指定步进参数，形式为 *s* [*i*:*j*:*stride*] 。

使用运算符< 、> 、<= 、>= 、== 和!= 可以对序列进行比较。比较两个序列时，首先比较每个序列的第一个元素。如果它们不同，可以马上得出结论为不同；如果它们相同，就继续比较每个序列的第二个元素。此过程一直持续，直至找出两个不同元素或者两个序列中都没有其他元素为止。如果能同时到达两个序列的结尾，就认为它们是相等的。如果a 是b 的子序列，那么 *a* < *b* 。

字符串的比较使用的是词典式排序。每个字符都分配有一个唯一的数字索引，其值由字符集（如ASCII或Unicode）决定。如果一个字符的索引小于另一个字符，就认为它比较小。关于字符排序要注意一点，即前面的简单比较运算符和与地区或语言设置相关的字符排序规则无关。因此，不能根据某种外语的标准约定使用这些操作对字符串排序（更多信息参见unicodedate 和locale 模块）。

另一条注意事项与字符串有关。Python有两种字符串数据：字节字符串和Unicode字符串。字节字符串与对应的Unicode字符串的区别在于，它们通常是已编码的，而Unicode字符串代表的是原始未编码字符值。因此，禁止在表达式或比较中同时使用字节字符串和Unicode字符串（如使用+连接一个字节字符串和一个Unicode字符串，或者使用= =比较它们）。在Python 3中，混合使用字符串类型将导致TypeError 异常，但是Python 2会试着将字节字符串隐式地转换为Unicode字符串。大多数程序员认为Python 2的这种做法是一个设计失误，经常会导致意想不到的异常和无法解释的程序行为。因此，为了避免碰到这些让人头大的问题，不要在序列操作中混合使用字符串类型。

4.3 字符串格式化

取模运算符（ *s* % *d* ）可生成格式化的字符串，其中 *s* 是一个格式字符串，而 *d* 是一个对象元组或映射对象（字典）。这个运算符的行为类似于C语言中的sprintf() 函数。格式字符串包含两类对象：普通字符（其值不变）和转换说明符，每个转换说明符将被代表相关联元组或映射中元素的格式化字符串代替。如果 *d* 是一个元组，转换说明符的个数必须与 *d* 中对象的个数保持一致。如果 *d* 是一个映射，每个转换说明符都必须与映射中的一个有效键名相关联（方法是使用括号，这一点很快就会讲到）。每个转换说明符都以% 字符开始，以表4-1中的某个转换字符结尾。

表**4-1** 字符串格式化转换

字　　符	输出格式
d, i	十进制整数或长整数
u	无符号整数或长整数
o	八进制整数或长整数
x	十六进制整数或长整数
X	十六进制整数（大写字母）
f	浮点数，如[-]*m.dddddd*
e	浮点数，如[-]*m.ddddde±xx*
E	浮点数，如[-]*m.ddddddE±xx*
g, G	指数小于Ĥ4或更高精度时使用%e 或%E ， 否则使用%f
s	字符串或任意对象。格式化代码使用str() 生成字符串
r	同repr() 生成的字符串
c	单个字符
%	字面量%

在%字符和转换字符之间，可以出现以下修饰符，并且只能按照以下顺序出现。

(1) 位于括号中的一个键名，用于从映射对象中选出一个具体项。如果不存在此类元素，就会引发KeyError 异常。

(2) 下面所列的一个或多个。

- -，左对齐标志。默认值为右对齐。
- +，表示应该包含数值符号（即使为正值也是如此）。
- 0，表示一个零填充。

(3) 一个指定最小自动宽度的数。转换后的值将被打印在至少为这个宽度的字段中，并且在左边填充至满字段宽（如果指定了- 标志，则填充在右边）。

(4) 一个小数点，用于按照精度分割字段宽度。

(5) 一个数，指定要打印字符串中的最大字符个数，浮点数中小数点之后的位数，或者整数的最小位数。

另外，星号（* ）字符用于在任意宽度的字段中替换数。如果存在，宽度将从元组的下一项中读出。

下面的代码给出了一些例子：

```
a = 42
b = 13.142783
c = "hello"
d = {'x':13, 'y':1.54321, 'z':'world'}
e = 5628398123741234
r = "a is %d" % a            # r = "a is 42"
r = "%10d %f" % (a,b)         # r = "       42  13.142783"
r = "%+010d %E" % (a,b)       # r = "+000000042 1.314278E+01"
r = "%(x)-10d  %(y)0.3g" % d  # r = "13        1.54"
r = "%0.4s %s" % (c, d['z'])  # r = "hell world"
r = "%*.*f" % (5,3,b)          # r = "13.143"
r = "e = %d" % e               # r = "e = 5628398123741234"
```

和字典一起使用时，字符串格式化运算符%通常用于模仿在脚本语言中常见的插值（interpolation）功能（如字符串中$var 符号的扩展）。例如，如果你有一个包含数个值的字典，那么你可以在格式化字符串中把这些值扩展到字段中。

```
stock = {
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10 }

r = "%(shares)d of %(name)s at %(price)0.2f" % stock
# r = "100 shares of GOOG at 490.10"
```

下面的代码说明了如何将当前定义变量的值扩展到一个字符串中。调用vars() 函数

时，它将返回一个包含此时已定义的所有变量的字典。

```
name = "Elwood"
age  = 41
r = "%(name)s is %(age)s years old" % vars()
```

4.4 高级字符串格式化

字符串格式化有一种更加高级的形式，即使用字符串的 *s* .format(**args*, ***kwargs*) 方法。该方法接受位置参数和关键字参数的任意集合，并使用它们的值来替换 *s* 中嵌入的占位符。形式为{*n* } 的占位符（ *n* 为数字）将被format() 方法的位置参数 *n* 所代替。而形式为{*name* } 的占位符将被format() 方法的关键字参数 *name* 所代替。如果要输出一个{ 或} ，必须使用{{ 或}} 的形式。例如：

```
r = "{0} {1} {2}".format('GOOG',100,490.10)
r = "{name} {shares} {price}".format(name='GOOG',shares=100,price=490.10)
r = "Hello {0}, your age is {age}".format("Elwood",age=47)
r = "Use {{ and }} to output single curly braces".format()
```

使用占位符还可以执行其他索引和属性查找。例如，在{*name* [*n*]} 中， *n* 为整数，就执行序列查找，而在{*name* [*key*]} 中， *key* 是一个非数字字符串，就执行形式为 *name* ['*key* '] 这样的字典查找。在{*name .attr* } 中执行属性查找。下面给出了一些例子：

```
stock = { 'name' : 'GOOG',
      'shares' : 100,
      'price' : 490.10 }
r = "{0[name]} {0[shares]} {0[price]}".format(stock)

x = 3 + 4j
r = "{0.real} {0.imag}".format(x)
```

这些扩展中只允许使用名称，而不支持任意的表达式、方法调用和其他操作。

另外，还可以指定格式说明符，对输出进行更加精确的控制。方法是用一个冒号（:）给每个占位符添加可选的格式说明符，如{*place:format_spec* } 。使用这种说明符可以指定列宽、小数位和对齐方式，例如：

```
r = "{name:8} {shares:8d} {price:8.2f}".format
(name="GOOG",shares=100,price=490.10)
```

说明符的一般格式是[[*fill* [*align*]][*sign*][0][*width*][.*precision*] [*type*] ，[] 中的每个部分都是可选的。width 说明符指定要使用的最小字段宽度，align 说明符的值可取< 、> 或^ 之一，分别代表在字段中左对齐、右对齐和居中对齐。fill 是一个可选的填充字符，用于填充空白。例如：

```
name = "Elwood"
r = "{0:<10}".format(name)     # r = 'Elwood    '
r = "{0:>10}".format(name)     # r = '    Elwood'
r = "{0:^10}".format(name)     # r = '  Elwood  '
r = "{0:=^10}".format(name)    # r = '==Elwood=='
```

type 说明符表示数据的类型。表4-2列出了支持的格式代码。如果没有提供，默认的格式代码分别是：字符串是s，整数是d，浮点数是f。

表**4-2** 高级的字符串格式化类型说明符代码

字　　符	输出格式
d	十进制整数或长整数
b	二进制整数或长整数
o	八进制整数或长整数
x	十六进制整数或长整数
X	十六进制整数（大写字母）
f, F	浮点数，如[-]m.dddddd
e	浮点数，如[-]m.ddddde±xx
E	浮点数，如[-]m.ddddddE±xx
g, G	指数小于-4 或更高精度时使用%e 或%E，否则使用%f
n	同g，区别在于由当前的区域设置决定小数点字符
%	把一个数乘以100，并使用后面带一个% 号的f 格式显示它
s	字符串或任意对象。格式化代码使用str() 生成字符串
C	单个字符

格式说明符的 *sign* 部分是+、- 或“ ”（空格）之一。+ 表示所有数之前都要加上符

号。- 是默认值，表示只在负数前面加上符号。空格表示在正数前面加上一个空格。说明符的 *precison* 部分用于为十进制数提供精度位数。如果在数的字段宽度前面加上一个0，就会使用0来填充数值前面的空白。以下例子说明了如何格式化不同类别的数：

```
x = 42
r = '{0:10d}'.format(x)        # r = '        42'
r = '{0:10x}'.format(x)        # r = '        2a'
r = '{0:10b}'.format(x)        # r = '    101010'
r = '{0:010b}'.format(x)       # r = '0000101010'

y = 3.1415926
r = '{0:10.2f}'.format(y)      # r = '     3.14'
r = '{0:10.2e}'.format(y)      # r = '  3.14e+00'
r = '{0:+10.2f}'.format(y)     # r = '     +3.14'
r = '{0:+010.2f}'.format(y)    # r = '+000003.14'
r = '{0:+10.2%}'.format(y)     # r = '  +314.16%'
```

通过为格式化函数提供其他字段，可以有选择地提供格式说明符。在格式字符串中，可以使用与普通字段相同的语法访问它们，例如：

```
y = 3.1415926
r = '{0:{width}.{precision}f}'.format(y,width=10,precision=3)
r = '{0:{1}.{2}f}'.format(y,10,3)
```

这种字段嵌套的深度只能为一级，而且只能出现在格式说明符部分中。另外，被嵌套的值不能另外再有它们自己的任何格式说明符。

在使用格式说明符时请注意，对象可以定义它们自己的自定义说明符集合。实际上，高级字符串格式化是调用了每个字段值上的特殊方法__format__(*self, format _spec*)。因此，format() 操作的功能是无限制的，具体取决于它应用的对象。例如，日期、时间和其他种类的对象可以定义它们自己的格式代码。

在某些情况下，可能只是要格式化对象的str() 或repr() 表示，绕过其__format__() 方法实现的功能。为此，可以在格式说明符前面添加!s 或!r 修饰符，例如：

```
name = "Guido"
r = '{0!r:^20}'.format(name)    # r = "    'Guido'    "
```

4.5 字典操作

字典 提供名称和对象之间的映射，它支持的操作如下所示。

操　作	描　述	操　作	描　述
x = d[*k*]	通过键进行索引	*k* in *d*	测试某个键是否存在

d[*k*] = *x*	通过键进行赋值	len(*d*)	字典中的项数
del *d* [*k*]	通过键删除一项		

键的值可以是任意不可变对象，如字符串、数和元组。另外，字典的键也可以是一列用逗号分开的值，例如：

```
d = { }
d[1,2,3] = "foo"
d[1,0,3] = "bar"
```

在这个例子中，键的值表示一个元组，因此上面的赋值语句完全等价于以下代码：

```
d[(1,2,3)] = "foo"
d[(1,0,3)] = "bar"
```

4.6 集合操作

set 和frozenset 类型支持大量常见的集合操作，如下所示。

操　　作	描　　述	操　　作	描　　述
s | t	*s* 和 *t* 的并集	len(s)	集合中项数
s & t	*s* 和 *t* 的交集	max(s)	最大值
s - t	求差集	min(s)	最小值
s ^ t	求对称差集		

并集、交集和差集操作的结果与最左边的操作数具有相同类型。例如，如果s 是一个frozenset ， 而t 是一个set ， 那么结果的类型将是frozenset 。

4.7 增量赋值

Python提供的增量赋值运算符如下所示。

操　　作	描　　述	操　　作	描　　述

x += y	*x = x + y*	*x %= y*	*x = x % y*
x -= y	*x = x – y*	*x &= y*	*x = x & y*
*x *= y*	*x = x * y*	*x |= y*	*x = x | y*
x /= y	*x = x / y*	*x ^= y*	*x = x ^ y*
x //= y	*x = x // y*	*X >>= y*	*x = x >> y*
*x **= y*	*x = x ** y*	*x <<= y*	*x = x << y*

这些运算符可以用在任何使用了普通赋值的地方，例如：

```
a = 3
b = [1,2]
c = "Hello %s %s"
a += 1                          # a = 4
b[1] += 10                      # b = [1, 12]
c %= ("Monty", "Python")       # c = "Hello Monty Python"
```

增量赋值不会违反可变性，也不会原地修改对象。因此，执行代码 *x += y* 时，将创建一个值为 *x + y* 的全新对象 *x* 。用户自定义的类可以使用特殊方法重新定义增量赋值运算符（参见第3章）。

4.8 属性(.)运算符

点（.）运算符用于访问对象的属性，例如：

```
foo.x = 3
print foo.y
a = foo.bar(3,4,5)
```

表达式中可以出现多个点运算符，如`foo.y.a.b` 。点运算符还可以用于函数的中间结果，如`a = foo.bar(3,4,5).spam` 。

用户自定义的类可以重新定义或自定义(.)的行为。第3章和第7章介绍了该内容的更多细节。

4.9 函数调用()运算符

`f(args)` 运算符用于对 *f* 进行函数调用。函数的每个参数都是一个表达式。在调用

函数之前， Python会从左到右对所有参数表达式进行求值，这有时被称为应用序求值（applicative order evaluation）。

使用functools 模块中的partial() 函数可以对函数参数进行部分求值，例如：

```
def foo(x,y,z):
    return x + y + z

from functools import partial
f = partial(foo,1,2)  # 为foo的参数x和y提供值
f(3)                    # 调用foo(1,2,3)，结果是6
```

partial() 函数对一个函数的某些参数求值，返回的对象在之后调用时只需要提供剩下的参数即可。在前面的例子中，变量 *f* 代表一个部分求值的函数，其中前两个参数的值已经计算出来。此时只需要提供函数执行所需要的最后一个参数值。函数参数的部分求值与叫做科里化（currying）的过程关系十分紧密。所谓科里化就是这样一种机制：把一个接受多个参数的函数（例如f(x,y) ）分解为一系列函数，其中每个函数接受其中一个参数（例如，首先通过固定x 的值对f 进行部分求值，从而获得一个新函数，然后为这个新函数提供y 值便可得到结果）。

4.10 转换函数

有时必须在内置类型之间执行转换。要在两种类型之间转换，只需用类型名称作为函数。另外，还有几个内置函数可用于执行特殊类别的转换。所有这些函数均返回一个新对象，该对象代表了转换后的值。

函　　数	描　　述
int(*x* [,*base*])	将 *x* 转换为一个整数。如果 *x* 是一个字符串， *base* 用于指定基数
float(*x*)	将 *x* 转换为一个浮点数
Complex(*real* [, *imag*])	创建一个复数
str(*x*)	将对象 *x* 转换为字符串表示
repr(*x*)	将对象 *x* 转换为一个表达式字符串
format(*x* , [,*format_spec*]	将对象 *x* 转换为格式化字符串
eval(*str*)	对字符串求值并返回对象

tuple(*s*)	将 *s* 转换为元组
list(*s*)	将 *s* 转换为列表
set(*s*)	将 *s* 转换为集合
dict(*d*)	创建字典。 *d* 必须是(*key*,*value*) 元组的序列
frozenset(*s*)	将 *s* 转换为不可变集合
chr(*x*)	将整数转换为字符
unichr(*x*)	将整数转换为Unicode字符（只能用于Python 2）
ord(*x*)	将字符转换为其整数值
hex(*x*)	将整数转换为十六进制字符串
bin(*x*)	将整数转换为二进制字符串
oct(*x*)	将整数转换为八进制字符串

注意，str() 和repr() 函数返回的结果可能不同。repr() 函数通常会创建一个表达式字符串，可以使用eval() 对它求值以重新创建对象。另一方面，str() 函数生成的是对象的一种简明或格式化的表示（由print 语句使用）。format(*x* , [*format_spec*] 函数生成的输出与高级字符串格式化操作的输出相同，但是适用于单个对象x 。它接受一个可选的、包含格式化代码的字符串format_spec 作为输入。ord() 函数返回一个字符的整数顺序值。在Unicode中，这个值就是一个整数代码点。chr() 和unichr() 函数用于将整数转换成字符。

要将字符串转换成数，可以使用int() 、float() 和complex() 函数。eval() 函数还可以将包含合法表达式的字符串转换为对象。例如：

```
a = int("34")             # a = 34
b = long("0xfe76214", 16) # b = 266822164L (0xfe76214L)
b = float("3.1415926")    # b = 3.1415926
c = eval("3, 5, 6")       # c = (3,5,6)
```

在创建容器的函数中，如list() 、tuple() 、set() 等，参数可以是支持迭代的任意对象，而迭代生成的所有项将用于填充要创建的对象。

4.11 布尔表达式与真值

and 、or 和not 关键字可用于构建布尔表达式。这些运算符的行为如下所示。

运 算 符	描 述
x or *y*	如果 *x* 为false ， 则返回 *y* ； 否则返回 *x*
x and *y*	如果 *x* 为false ， 则返回 *x* ； 否则返回 *y*
not *x*	如果 *x* 为false ， 则返回1 ； 否则返回0

使用表达式来判断True 或False 值时， True 、任意非零数值，以及非空的字符串、列表、元组或字典，都将返回True ， 而False 、0 、None ， 以及空的列表、元组和字典，都将返回False 。布尔表达式从左至右进行求值，而且只有在需要时才会计算右边的操作数。例如表达式a and b ， 只有当a 为True 时才会计算b 。这有时称为短路求值 。

4.12 对象等同性与标识

等于运算符（ *x* == *y* ）可以检测x 和y 的值是否相等。对于列表和元组，只有其中的所有元素都相等，它们才相等。对于字典，只有当x 和y 的键都相同，而且键相同的所有对象的值都相等才会返回True 。两个集合相等的条件是它们具有相同的元素，这可用== 运算符进行比较得出。

标识运算符（ *x* is *y* 和x is not *y* ）可以检测两个对象是否引用了内存中的同一个对象。一般而言，可能 *x* == *y* ，但 *x* is not *y* 。

比较操作也可以在两个非兼容对象之间进行，如一个文件和一个浮点数，但返回结果是任意的，可能没有意义。另外这也可能导致异常，具体取决于类型。

4.13 运算优先级

表4-3列出了Python运算符的操作顺序（优先级规则）。除了乘方（** ）运算符之外的所有运算符都是从左至右进行计算，表中按照从高到低的优先级顺序列出了这些运算符。也就是说，表中先列出的运算符优先计算。（注意，位于同一栏中的运算符优先级相等，如 *x* * *y* 、 *x* / *y* 和 *x* % *y* 。）

表**4-3** 运算优先级（从高到低）

运 算 符	名 称
(...), [...], {...}	创建元组、列表和字典

s [*i*], *s* [*i*:*j*]	索引和切片
s.*attr*	属性
f (...)	函数调用
+*x* , -*x* , ~*x*	一元运算符
x ** *y*	乘方（从右至左运算）
x * *y* , *x* / *y* , *x* // *y* , *x* % *y*	乘法、除法、地板除法、取模
x + *y* , *x* - *y*	加法、减法
x << *y* , *x* >> *y*	移位
x & *y*	按位与
x ^ *y*	按位异或
x | *y*	按位或
x < *y* , *x* <= *y* ,	比较、标识和序列成员检查
x > *y* , *x* >= *y* ,	
x == *y* , *x* != *y*	
x is *y* , *x* is not y	
x in *s* , *x* not in *s*	
not *x*	逻辑非
x and *y*	逻辑与
x or *y*	逻辑或

lambda *args* : *expr*	匿名函数

运算顺序并非由表4-3中 *x* 和 *y* 的类型决定。因此，尽管用户自定义对象可以重新定义每个运算符，却无法自定义底层的运算顺序、优先级和关联规则。

4.14 条件表达式

常见的编程模式是根据表达式的结果，有条件地进行赋值，例如：

```
if a <= b:
   minvalue = a
else:
   minvalue = b
```

使用条件表达式可以简化这段代码，例如：

```
minvalue = a if a <=b else b
```

在这类表达式中，首先求值的是中间的条件。如果结果为`True`，再对`if`语句左面的表达式求值，否则就会对`else`后面的表达式求值。

条件表达式应该尽量少用，因为它们可能导致混淆，特别是将其嵌套或者与其他复杂表达式混在一起的时候。然而，条件表达式在列表推导（list comprehension）和生成器表达式中特别有用，例如：

```
values = [1, 100, 45, 23, 73, 37, 69 ]
clamped = [x if x < 50 else 50 for x in values]
print(clamped)      # [1, 50, 45, 23, 50, 37, 50]
```

第5章　程序结构与控制流

本章详细介绍程序结构与控制流。涉及的主题包括条件语句、迭代、异常和上下文管理器。

5.1　程序结构与执行

Python程序由一些语句序列组成。编程语言的所有功能，包括变量赋值、函数定义、类和模块导入，这些语句和其他所有语句拥有平等地位。事实上，不存在“特殊的”语句，每条语句都可以放在程序中的任意位置。例如，下面这段代码定义了一个函数的两个不同版本：

```
if debug:
    def square(x):
        if not isinstance(x,float):
            raise TypeError("Expected a float")
        return x * x
else:
    def square(x):
        return x * x
```

加载源文件时，解释器始终按顺序执行每条语句，直到再无语句可以执行。这种执行模式同时适用于作为主程序运行的文件和通过`import` 加载的库文件。

5.2　执行条件语句

`if` 、`else` 和`elif` 语句用于控制条件代码的执行。条件语句的一般格式如下所示：

```
if expression

:
    statements


elif expression

:
    statements


elif expression

:
    statements


...
else:
```

```
    statements
```

如果不需要执行任何操作，可以省略条件语句的else 和elif 子句。如果特定子句下不存在要执行的语句，可使用pass 语句。

```
if expression:

    pass        # 不执行任何操作
else:
    statements
```

5.3 循环与迭代

可以使用for 和while 语句实现循环，例如：

```
while expression:

    statements


for i in s:
    statements
```

while 语句反复执行循环体中的语句，直到相关表达式求值为假。for 语句迭代s 中的所有元素，直到再无可用元素。for 语句仅适用于可支持迭代的对象。这显然包括内置的序列类型，如列表、元组和字符串，而且还适用于实现了迭代器协议的对象。

如果对象s 能以下面代码演示的方式使用，那它就是支持迭代的。这段代码模拟了for 循环的实现：

```
it = s.__iter__()            # 获得s的迭代器
while 1:
     try:
         i = it.next()        # 获得下一项（在Python 3中需使用__next__）
     except StopIteration:    # 不再有可用项
         break
         # 执行对i的操作
     ...
```

在for *i* in *s* 语句中，变量 *i* 称为迭代变量 。在循环的每次迭代中，它都会从 *s* 接收一个新值。迭代变量的作用域并非for 语句私有。如果前面已经定义了一个相同名称的变量，它的值将被改写。另外，循环结束后迭代变量依然保留最后一个值。

如果迭代中使用的元素是元素大小完全一致的序列，可以使用下面这样的语句把它们的值解包到单独的迭代变量中：

```
for x,y,z in s:
    statements
```

在这个例子中，s 必须包含或能产生一些序列，每个序列包含3个元素。每次迭代时，会把相应序列的各个元素赋值给变量 *x* 、 *y* 和 *z* 。尽管使用这行代码时 *s* 大多为元组序列，但 *s* 中的各项可以为任意类型的序列，包括列表、生成器和字符串。

循环时，除了数据值之外，有时还需要跟踪数字索引，例如：

```
i = 0
for x in s:
    statements


    i += 1
```

Python提供了一个内置函数enumerate()，可以简化上面的代码：

```
for i,x in enumerate(s):
     statements
```

enumerate(*s*) 创建了一个迭代器，其返回值就是一个元组的序列(0, *s* [0])、(1, *s* [1])、(2, *s* [2]) 等。

另一个常见的循环问题与同时迭代两个以上的序列有关。例如，写一个下面这样的循环，每次迭代获得不同序列中的元素：

```
# s和t是两个序列
i = 0
while i < len(s) and i < len(t):
    x = s[i]     # 获得s的一个元素
    y = t[i]     # 获得t的一个元素
    statements
```

```
    i += 1
```

使用zip() 函数可以简化这段代码，例如：

```
# s和t是两个序列
for x,y in zip(s,t):
    statements
```

zip(*s* , *t*) 将序列 *s* 和 *t* 组合为一个元组序列(*s* [0], *t* [0])、(*s* [1], *t* [1])、(*s* [2], *t* [2]) 等，如果 *s* 和 *t* 的长度不等，则至用完长度最短的索引为止。使用zip() 函数时需要注意一点，即在Python 2中，它会将 *s* 和 *t* 中的元素完全消耗尽，创建一个元组的列表。对于包含大量数据的生成器和序列而言，这也许不是你想要的结果。函数itertools.izip() 实现的效果与zip() 相同，但一次只生成一个元组，而不是创建一个很大的元组列表。在Python 3中，zip() 函数生成值的方式也是如此。

使用break 语句可从循环中跳出。例如，下面这段代码的功能是从文件中读取文本行，直至遇到空的文本行为止：

```
for line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        break           # 遇到一个空行，停止读取
    # 处理已读出的文本行
    ...
```

使用continue 语句可以跳到循环的下一次迭代（跳过循环体中的余下代码）。这条语句往往不常用到，但是当逆转条件判断后增加一层缩进，使得程序嵌套太深或过于复杂时，它还是很有用的。例如，以下代码跳过了一个文件中的所有空行：

```
for line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        continue      # 跳过空行
    # 处理已读出的文本行
    ...
```

break 和continue 语句仅应用于正在执行的最内层循环。如果需要跳出多层嵌套循环结构，可以使用异常。Python不提供“goto ”语句。

在循环结构中也可以加入else 语句，例如：

```
# for-else
```

```
for line in open("foo.txt"):
     stripped = line.strip()
     if not stripped:
          break
      # 处理读出的行
      ...
else:
    raise RuntimeError("Missing section separator")
```

循环的else 子句只在循环运行完成后才会执行。也就是说，如果循环体根本不执行，就立即执行else 内容，否则在最后一次迭代后执行。但是，如果先一步使用break 语句终止了循环，else 子句将被跳过。

循环的else 子句的首要用途是在数据循环迭代过早终止时，在代码中设置一些标志或条件，或者对这些标志和条件进行检查。例如，如果不使用else 子句，前面的代码就必须改写，让其使用标志变量：

```
found_separator = False
for line in open("foo.txt"):
     stripped = line.strip()
     if not stripped:
         found_separator = True
         break
     # 处理读出的行
     ...
if not found_separator:
    raise RuntimeError("Missing section separator")
```

5.4 异常

异常 意味着出现错误，并且会中断程序的正常控制流。使用raise 语句可以引发异常。raise 语句的一般格式是raise *Exception* ([*value*])，其中 *Exception* 是异常类型，而 *value* 是一个说明异常相关细节的可选值，例如：

```
raise RuntimeError("Unrecoverable Error")
```

如果raise 语句没有带任何参数，将会再次引发最近一次生成的异常（只有仍在处理一个前面引发的异常时才会如此）。

使用try 和except 语句可以捕捉异常，如下所示：

```
try:
    f = open('foo')
except IOError as e:
    statements
```

出现异常时，解释器将停止执行try 代码块中的语句，并寻找可匹配该异常的except 子句。如果找到，控制权就会传递给except 子句中的第一条语句。执行完except 子句后，控制权就会传递给出现在try-except 代码块之后的第一条语句。否则，异常将传递给try 语句所在的上一级代码块。而这个代码库自身也可能被包括在一个可以处理该异常的try-except 语句中。如果异常传递到程序的最顶级却依然未被捕捉，解释器就会终止程序运行并显示一条错误消息。如果需要，也可以把未捕捉的异常传递给用户自定义的函数sys.excepthook() 进行处理，关于这一点请参见第13章。

except 语句的可选修饰符as *var* 提供了一个变量名称，如果出现异常，就会在其中放置一个提供给raise 语句的异常类型的实例。异常处理程序可以检查该值，从而找出关于异常原因的更多信息。例如，可以使用isinstance() 函数检查异常类型。关于语法有一条注意事项：在以前版本的Python中，except 语句被写作except ExcType , *var* ，其中异常类型和变量是由逗号（,）隔开的。在Python 2.6中，这种语法仍然有效，但已被废弃。新的代码都使用as *var* 语法，因为这在Python 3中是必需的。

使用多条except 子句可以指定多个异常处理代码块，如下所示：

```
try:
   do something
except IOError as e:
   # 处理I/O错误
   ...
except TypeError as e:
   # 处理类型错误
   ...
except NameError as e:
   # 处理名称错误
   ...
```

处理程序也可以捕捉多种类型的异常，例如：

```
try:
   do something
except (IOError, TypeError, NameError) as e:
   # 处理I/O、类型或名称错误
   ...
```

使用pass 语句可以忽略异常，例如：

```
try:
   do something
except IOError:
   pass              # 不做任何处理
```

如果要捕捉除与程序退出相关异常之外的所有异常，请这样使用Exception ：

```
try:
   do something
except Exception as e:
```

```
   error_log.write('An error occurred : %s\n' % e)
```

捕捉所有异常时，注意应该给用户报告准确的错误信息。例如，在前面的代码中，就记录了一条错误消息和相关的异常值。如果没有包含关于异常值的任何信息，那么在代码由于某些意想不到的理由出现故障时，调试起来就会倍加困难。

使用except 语句时如果不带任何异常类型，将会捕捉所有异常，语法如下所示：

```
try:
   do something
except:
   error_log.write('An error occurred\n')
```

正确使用这种形式的except 比看上去更加棘手，应该尽量避免。例如，这段代码还将捕捉键盘中断和程序退出的请求，而这些并不是你想要捕捉的对象。

try 语句也支持else 子句，它必须跟在最后一个except 子句后面。如果try 代码块中的代码没有引发异常，就会执行else 子句中的代码，例如：

```
try:
   f = open('foo', 'r')
except IOError as e:
   error_log.write('Unable to open foo : %s\n' % e)
else:
   data = f.read()
   f.close()
```

finally 语句为try 代码块中的代码定义了结束操作，例如：

```
f = open('foo','r')
try:
   # 进行一些处理
   ...
finally:
   f.close()
   # 无论前面发生什么，都会关闭文件
```

finally 子句不是用于捕捉错误的。相反，它用于提供一些无论是否出现错误都必须执行的代码。如果没有引发异常，finally 子句中的代码将在try 代码块中的代码执行完毕后立即执行。如果引发了异常，控制权首先传递给finally 子句的第一条语句。这段代码执行完毕后，将重新引发异常然后交由另一个异常处理程序进行处理。

5.4.1 内置异常

Python定义的内置异常如表5-1所示。

表**5-1** 内置异常

异　　常	描　　述
BaseException	所有异常的根异常
GeneratorExit	由生成器的.close() 方法引发
KeyboardInterrupt	由键盘中断（通常为Ctrl+C）生成
SystemExit	程序退出/终止
Exception	所有非退出异常的基类
StopIteration	引发后可停止迭代
StandardError	所有内置异常的基类（仅在Python 2中使用）。在Python 3中，下面的所有异常都归在Exception 下
ArithmeticError	算术异常的基类
FloatingPointError	浮点操作失败
ZeroDivisionError	对0进行除或取模操作
AssertionError	由assert 语句引发
AttributeError	当属性名称无效时引发
EnvironmentError	发生在Python外部的错误
IOError	I/O或文件相关的错误
OSError	操作系统错误
EOFError	到达文件结尾时引发
ImportError	import 语句失败

LookupError	索引和键错误
IndexError	超出序列索引的范围
KeyError	字典键不存在
MemoryError	内存不足
NameError	无法找到局部或全局名称
UnboundLocalError	未绑定的局部变量
ReferenceError	销毁被引用对象后使用的弱引用
RuntimeError	一般运行时错误
NotImplementedError	没有实现的特性
SyntaxError	解析错误
IndentationError	缩进错误
TabError	使用不一致的制表符（由-tt 选项生成）
SystemError	解释器中的非致命系统错误
TypeError	给操作传递了错误的类型
ValueError	无效类型
UnicodeError	Unicode错误
UnicodeDecodeError	Unicoed解码错误
UnicodeEncodeError	Unicode编码错误
UnicodeTranslateError	Unicode转换错误

异常的层次结构如上表中所示。只要在except 子句中指定一个特定组的名称，就可以捕捉该组中的所有异常，例如：

```
try:
     statements


except LookupError:    # 捕捉IndexError或KeyError
     statements
```

或者

```
try:
     statements


except Exception:    # 捕捉与程序相关的所有异常
     statements
```

在异常层次结构的最顶层，根据异常是否与程序退出有关对异常进行分组。例如，SystemExit 和KeyboardInterrupt 异常不属于Exception 分组，因为要捕捉所有与程序相关错误的程序通常并不想捕捉程序的异常终止。

5.4.2 定义新异常

所有内置异常都使用类进行定义。要创建新异常，就定义父类为Exception 的新类，例如：

```
class NetworkError(Exception): pass
```

可用如下方式通过raise 语句使用这个新异常：

```
raise NetworkError("Cannot find host.")
```

引发异常时，提供给raise 语句的可选值将被用作异常的类构造函数的参数。通常，它就是一个表示某些错误消息的字符串。但用户自定义的异常可以带有一个或多个异常值，如下所示：

```
class DeviceError(Exception):
     def __init__(self,errno,msg):
         self.args = (errno, msg)
         self.errno = errno
```

```
        self.errmsg = msg

# 引发一个异常（多参数）
raise DeviceError(1, 'Not Responding')
```

如上所示，创建的自定义异常类重新定义了__init__()方法时，就要将包含__init__()方法参数的元组赋值给属性self.args。打印异常跟踪消息时就需要用到这个属性。如果不定义该属性，出现错误时，用户就无法看到关于异常的任何有用信息。

使用继承机制可将异常组织为一个层次结构。例如，前面定义的NetworkError异常可作为更多特定错误的基类，如下所示：

```
class HostnameError(NetworkError): pass
class TimeoutError(NetworkError): pass
def error1():
    raise HostnameError("Unknown host")

def error2():
    raise TimeoutError("Timed out")

try:
    error1()
except NetworkError as e:
    if type(e) is HostnameError:
        # 对这类错误执行特殊操作
        ...
```

在这个例子中，except NetworkError语句捕捉任何从NetworkError异常派生而来的异常。要找出引发异常的具体类型，可以使用type()函数检查执行值的类型。另外，sys.exc_info()函数可用于获得最近一次引发异常的相关信息。

5.5 上下文管理器与with语句

在出现异常时正确地管理各种系统资源（如文件、锁和连接）通常是一个棘手的问题。例如，引发的异常可能导致控制流跳过负责释放关键资源（如锁）的语句。

with语句支持在由上下文管理器对象控制的运行时上下文中执行一系列语句，例如：

```
with open("debuglog","a") as f:
     f.write("Debugging\n")
     statements


     f.write("Done\n")

import threading
lock = threading.Lock()
with lock:
     # 关键部分
     statements
```

```
    # 关键部分结束
```

在第一个例子中，当控制流离开with 语句后面的代码块时，with 语句将自动关闭已打开的文件。在第二个例子中，当控制流进入with 语句后面的代码块时自动请求一个锁，而在控制流离开时又自动释放了这个锁。

with obj 语句允许对象 *obj* 管理控制流进入和离开相关代码块时要执行什么操作。执行with *obj* 语句时，它执行方法 *obj* .__enter__() 来指示正在进入一个新的上下文。当控制流离开该上下文时，就会执行方法 *obj* .__exit__(*type* , *value* , *traceback*)。如果没有引发异常，__exit__() 方法的3个参数均被设为None 。否则，它们将包含与导致控制流离开上下文的异常相关的类型、值和跟踪信息。__exit__() 方法返回True 或False ，分别表示被引发的异常是否得到了处理（如果返回False ，引发的任何异常都将被传递出上下文）。

with *obj* 语句接受一个可选的as *var* 说明符。如果指定了该说明符，obj.__enter__() 方法的返回值将保存在 *var* 中。要强调的一点是， *obj* 不一定就是赋给 *var* 的值。

with 语句只对支持上下文管理协议（__enter__() 和__exit__() 方法）的对象有效。用户定义的类可以实现这些方法，从而实现自定义上下文管理，如下所示：

```
class ListTransaction(object):
    def __init__(self,thelist):
        self.thelist = thelist
    def __enter__(self):
        self.workingcopy = list(self.thelist)
        return self.workingcopy
    def __exit__(self,type,value,tb):
        if type is None:
            self.thelist[:] = self.workingcopy
        return False
```

这个类支持对已有列表进行一系列的修改，但这些修改只有在没有异常发生时才会生效，否则原始列表将保持不变，例如：

```
items = [1,2,3]
with ListTransaction(items) as working:
    working.append(4)
    working.append(5)
print(items)      # 生成[1,2,3,4,5]

try:
    with ListTransaction(items) as working:
        working.append(6)
        working.append(7)
        raise RuntimeError("We're hosed!")
except RuntimeError:
    pass
```

```
print(items)    # 生成[1,2,3,4,5]
```

contextlib 模块支持通过包装生成器函数，更加容易地实现自定义上下文管理器，例如：

```
from contextlib import contextmanager
@contextmanager
def ListTransaction(thelist):
    workingcopy = list(thelist)
    yield workingcopy
    # 仅在没有出现错误时才会修改原始列表
    thelist[:] = workingcopy
```

这个例子把传递给yield 的值用作了__enter__() 方法的返回值。调用__exit__() 方法时，执行将在yield 语句后恢复。如果上下文中引发了异常，它将以异常形式出现在生成器函数中。如果需要，可以捕捉异常，但在这个例子中，异常将被传递出生成器并在其他地方进行处理。

5.6 断言与__debug__

assert 语句可以在程序中引入调试代码。assert 的一般格式为

```
assert test

 [, msg

]
```

其中test 是一个表达式，其值应该为True 或False 。如果test 求值为False ，assert 就会引发AssertionError 异常并使用在assert 中提供的可选消息msg ，例如：

```
def write_data(file,data):
    assert file, "write_data: file not defined!"
    ...
```

assert 语句不应用于必须执行以确保程序正确的代码，因为如果Python以优化模式运行（通过对解释器使用-O 选项进入该模式），它将不会执行。特别是不能用assert 语句检查用户输入。相反，assert 语句用于检查应该始终为真的内容；如果assert 语句引发异常，这意味着程序中存在bug，而不是用户出现了错误。

例如，如果计划把前面的函数write_data() 交付给最终用户使用，应该使用传统的if 语句和相应的错误处理代码替换assert 语句。

除了assert 语句之外，Python还提供内置的只读变量__debug__ ，它的值通常设置

为True，除非解释器以优化模式运行（通过使用-O选项指定）。程序可以在需要时检查这个变量——如果设置了该变量的值，很可能是在执行额外的错误检查程序。__debug__变量的底层实现在解释器中经过了优化，因此实际上没有包含if语句本身的额外控制流逻辑。如果Python以普通模式运行，if __debug__之下的语句会被内联到程序中，但不包括if语句本身。在优化模式中，if __debug__语句以及所有相关语句都将从程序中完全剔除。

使用assert和__debug__语句可以对程序进行高效的双模式开发。例如，在调试模式中，可以随意地在代码中加入断言和调试检查，以便验证操作正确与否。在优化模式中，将省略所有这些额外的检查，因此不会造成额外的性能负担。

第6章　函数与函数式编程

为了便于维护和更好地实现模块化，大量的程序被分解为多个函数。Python不仅简化了函数的定义过程，而且还大量引入了其他函数式编程语言中的优秀特性。本章将介绍函数、作用域规则、闭包、装饰器、生成器、协程以及其他一些函数式编程特性。另外，还将介绍列表推导和生成器表达式，它们都是进行声明式编程和数据处理的强大工具。

6.1　函数

使用def 语句可定义函数：

```
def add(x,y):
    return x + y
```

函数体就是在调用函数时所执行的一系列语句。调用函数的方法是在函数名称后面加上函数参数的元组，如a = add(3, 4) 。参数的顺序和数量必须与函数定义匹配，否则会引发TypeError 异常。

函数的参数可以拥有默认值，方法是在函数定义中为参数赋值，例如：

```
def split(line,delimiter=','):
    statements
```

如果函数定义中存在带有默认值的参数，该参数及其所有后续参数都是可选的。如果未给函数定义中的所有可选参数赋值，就会引发SyntaxError 异常。

默认参数值总是被设为函数定义时作为值传入的对象。示例如下：

```
a = 10
def foo(x=a):
    return x

a = 5            # 给'a'重新赋值
foo()            # 返回10（默认值没有变）
```

另外，使用可变对象作为默认值可能导致意料之外的结果：

```
def foo(x, items=[]):
    items.append(x)
    return items
foo(1)       # 返回[1]
foo(2)       # 返回[1, 2]
foo(3)       # 返回[1, 2, 3]
```

注意，默认参数保留了前面调用时进行的修改。为了防止出现这种情况，最好使用None 值，并在后面加上检查代码：

```
def foo(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items
```

如果给最后一个参数名加上星号（* ），函数就可以接受任意数量的参数：

```
def fprintf(file, fmt, *args):
    file.write(fmt % args)

# 使用fprintf。args被赋值为(42,"hello world", 3.45)
fprintf(out,"%d %s %f", 42, "hello world", 3.45)
```

在这个例子中，所有余下的参数都作为一个元组放入 *args* 变量。要把元组args 当作参数传递给函数，可以在函数调用中使用*args 语法：

```
def printf(fmt, *args):
        # 调用另一个函数，并把args传递给它
        fprintf(sys.stdout, fmt, *args)
```

提供函数参数还有一种方式，即显式地命名每个参数并为其指定一个值，这称为关键字参数 。如下所示：

```
def foo(w,x,y,z):
    statements



# 关键字参数调用
foo(x=3, y=22, w='hello', z=[1,2])
```

使用关键字参数时，参数的顺序无关紧要。但除非提供了默认值，否则必须显式地命名所有必需的函数参数。如果省略任何必需的参数，或者某个关键字的名称与函数定义中的参数名称不匹配，就会引发TypeError 异常。另外，由于所有Python函数都可以使用关键字调用的方式进行调用，因此，在定义函数的时候，使用这种描述性的参数名称通常是个好主意。

位置参数和关键字参数可以出现在同一次函数调用中，前提是所有位置参数必须先出现，给所有非可选参数提供值，并且不能多次定义参数值。例如：

```
foo('hello', 3, z=[1,2], y=22)
foo(3, 22, w='hello', z=[1,2])     # TypeError。w参数具有多个值
```

如果函数定义的最后一个参数以** 开头，所有额外的关键字参数（与任意其他参数名称都不匹配的参数）都可以放入一个字典中，并把这个字典传递给函数。如果要编写的函数需接受大量可扩充的配置选项作为参数，但列出这些参数又显得过于笨重，那么使用**开头的参数就很有用。例如：

```
def make_table(data, **parms):
    # 从parms（字典）获取配置参数
    fgcolor = parms.pop("fgcolor","black")
    bgcolor = parms.pop("bgcolor","white")
    width = parms.pop("width",None)
    ...
    # 无更多选项
    if parms:
         raise TypeError("Unsupported configuration options %s" % list(parms))

make_table(items, fgcolor="black", bgcolor="white", border=1,
                borderstyle="grooved", cellpadding=10,
                width=400)
```

关键字参数和可变长度参数列表可以一起使用，只要** 参数出现在最后即可：

```
# 接受数量不定的位置或关键字参数
def spam(*args, **kwargs):
    # args是一个位置参数的元组
    # kwargs是一个关键字参数的字典
    ...
```

还可以使用**kwargs 语法把关键字参数传递给另一个函数：

```
def callfunc(*args, **kwargs):
    func(*args,**kwargs)
```

*args 和**kwargs 通常用来为其他函数编写包装器和代理。例如，callfunc() 函数接受参数的任意组合，并把它们传递给func() 函数。

6.2 参数传递与返回值

调用函数时，函数参数仅仅是指代传入对象的名称。参数传递的基本语义和其他编程语言中已知的方式不完全相同，如“按值传递”或“按引用传递”。例如，如果传递不可变的值，参数看起来实际是按值传递的。但如果传递可变对象（如列表或字典）给函数，然后再修改此可变对象，这些改动将反映在原始对象中。例如：

```
a = [1, 2, 3, 4, 5]
def square(items):
    for i,x in enumerate(items):
        items[i] = x * x     # 原地修改item中的元素

square(a)      # a变为[1, 4, 9, 16, 25]
```

像这样悄悄修改其输入值或者程序其他部分的函数被认为具有副作用 。一般来说，最好避免使用这种编程风格，因为随着程序的规模和复杂程度不断增加，这类函数会成为各种奇怪编程错误的根源（例如，如果函数具有副作用，只看函数调用是无法明显发现的）。在涉及线程和并发的程序中，这类函数的交互能力很差，因为通常需要使用锁定来防止副作用的影响。

return 语句从函数返回一个值。如果没有指定任何值或者省略return 语句，就会返回None 对象。如果返回值有多个，可以把它们放在一个元组中：

```
def factor(a):
    d = 2
    while (d <= (a / 2)):
        if ((a / d) * d == a):
              return ((a / d), d)
        d = d + 1
    return (a, 1)
```

可将元组中的多个返回值赋给单独的变量：

```
x, y = factor(1243)   # 将返回值放在x和y中
```

或者

```
(x, y) = factor(1243) # 另一种赋值形式，效果相同
```

6.3 作用域规则

系统每次执行一个函数时，就会创建新的局部命名空间。该命名空间代表一个局部环境，其中包含函数参数的名称和在函数体内赋值的变量名称。解析这些名称时，解释器将首先搜索局部命名空间。如果没有找到匹配的名称，它就会搜索全局命名空间。函数的全局命名空间始终是定义该函数的模块。如果解释器在全局命名空间中也找不到匹配值，最终会检查内置命名空间。如果仍然找不到，就会引发NameError 异常。

命名空间的一个特别之处，是在函数中对全局变量的操作。例如，请看以下代码：

```
a = 42
def foo():
    a = 13
foo()
# a仍然是42
```

执行这段代码时，尽管看上去我们在函数foo 中修改了变量a 的值，但a 的返回值仍然是42 。当变量在函数中被赋值时，这些变量始终被绑定到该函数的局部命名空间中，

因此函数体中的变量a 引用的是一个包含值13 的全新对象，而不是外面的变量。使用global 语句可以改变这种行为。global 语句明确地将变量名称声明为属于全局命名空间，只有在需要修改全局变量时才必须使用它。这条语句可以放在函数体中的任意位置，并可重复使用。例如：

```
a = 42
b = 37
def foo():
    global a # 'a'位于全局命名空间中
    a = 13
    b = 0
foo()
# a现在已变为13。b仍然为37
```

Python支持嵌套的函数定义，例如：

```
def countdown(start):
   n = start
   def display(): # 嵌套的函数定义
       print('T-minus %d' % n)
   while n > 0:
       display()
       n -= 1
```

嵌套函数中的变量是由静态作用域 （lexical scoping）限定的。也就是说，解释器在解析名称时首先检查局部作用域，然后由内而外一层层检查外部嵌套函数定义的作用域。如果找不到匹配，那么和之前一样，将搜索全局命名空间和内置命名空间。尽管闭合作用域中的名称能被访问到，Python 2只支持在最里层的作用域（即局部变量）和全局命名空间（使用global ）中给变量重新赋值。因此，内部函数不能给定义在外部函数中的局部变量重新赋值。例如，下面这段代码是不起作用的：

```
def countdown(start):
   n = start
   def display():
       print('T-minus %d' % n)
   def decrement():
       n -= 1             # 在Python 2中无效
   while n > 0:
        display()
        decrement()
```

在Python 2中，解决这种问题的方法是把要修改的值放在列表或字典中。在Python 3中，可以把n 声明为nonlocal ，如下所示：

```
def countdown(start):
   n = start
   def display():
       print('T-minus %d' % n)
   def decrement():
       nonlocal n    # 绑定到外部的n(仅在Python 3中使用)
```

```
    n -= 1
  while n > 0:
      display()
      decrement()
```

nonlocal 声明不会把名称绑定到当前调用栈下方的任意函数中定义的局部变量，即动态作用域 （dynamic scope）中。因此，如果从Perl转而学习Python语言，要注意nonlocal 语句和Perl的local 变量声明不同。

如果使用局部变量时还没给它赋值，就会引发UnboundLocalError 异常，下面的例子演示了可能出现该问题的情况：

```
i = 0
def foo():
    i = i + 1    # 导致UnboundLocalError异常
    print(i)
```

在这个函数中，i 被定义为一个局部变量（因为它在函数内赋值，而且没有使用global 语句）。但是，赋值语句i = i + 1 会尝试在给i 局部赋值之前读取它的值。尽管这个例子中存在一个全局变量i ，但它不会给局部变量i 提供值。函数在定义时就确定了变量是局部的还是全局的，而且在函数中不能突然改变它们的作用域。例如，在前面的代码中，表达式i + 1 中的i 引用的不是全局变量i ，而print(i) 中的i 引用的是前一条语句中创建的局部变量i 。

6.4 作为对象与闭包的函数

函数在Python中是第一类对象。也就是说可以把它们当作参数传递给其他函数，放在数据结构中，以及作为函数的返回结果。下面的例子给出了一个函数，它接受另一个函数作为输入并调用它。

```
# foo.py
def callf(func):
   return func()
```

下面这个例子使用了上面的函数：

```
>>> import foo


>>> def helloworld():


...     return 'Hello World'


...
>>> foo.callf(helloworld)
```

```
    # 传递一个函数作为参数
'Hello World'
>>>
```

把函数当作数据处理时，它将隐式地携带与定义该函数的周围环境相关的信息。这将影响到函数中自由变量的绑定方式。例如，考虑下面这个修改后的foo.py，它现在包含了一个变量定义：

```
# foo.py
x = 42
def callf(func):
   return func()
```

现在观察这个例子的行为：

```
>>> import foo


>>> x = 37


>>> def helloworld():


...     return "Hello World. x is %d" % x


...
>>> foo.callf(helloworld)

     # 传递一个函数作为参数
'Hello World. x is 37'
>>>
```

在这个例子中，注意函数helloworld() 使用的x 的值是在与它相同的环境中定义的。因此，即使foo.py 中也定义了一个变量x，而且这里也是实际调用helloworld() 函数的地方，但x 的值与helloworld() 函数执行时使用的x 不同。

将组成函数的语句和这些语句的执行环境打包在一起时，得到的对象称为闭包 。事实上所有函数都拥有一个指向了定义该函数的全局命名空间的__globals__ 属性，这也解释了前面例子的行为。这始终对应于定义函数的闭包模块。对于前面的例子，可以看到如下内容：

```
>>> helloworld.__globals__


{'__builtins__': <module '__builtin__' (built-in)>,
 'helloworld': <function helloworld at 0x7bb30>,
 'x': 37, '__name__': '__main__', '__doc__': None
```

```
 'foo': <module 'foo' from 'foo.py'>}
>>>
```

使用嵌套函数时，闭包将捕捉内部函数执行所需的整个环境，例如：

```
import foo
def bar():
    x = 13
    def helloworld():
        return "Hello World. x is %d" % x
    foo.callf(helloworld)         # 返回'Hello World, x is 13'
```

如果要编写惰性求值（lazy evaluation）或延迟求值的代码，闭包和嵌套函数特别有用，例如：

```
from urllib import urlopen
# from urllib.request import urlopen (Python 3)
def page(url):
    def get():
        return urlopen(url).read()
    return get
```

在这个例子中，page() 函数实际上并不执行任何有意义的计算。相反，它只会创建和返回函数get()，调用该函数时会获取Web页面的内容。因此，get() 函数中执行的计算实际上延迟到了程序后面对get() 求值的时候。例如：

```
>>> python = page("http://www.python.org")


>>> jython = page("http://www.jython.org

")


>>> python


<function get at 0x95d5f0>
>>> jython


<function get at 0x9735f0>
>>> pydata = python()

        # 获取http://www.python.org
>>> jydata = jython()

        # 获取http://www.jython.org
>>>
```

在这个例子中，两个变量python 和jython 实际上是get() 函数的两个版本。即使创建这些值的page() 函数不再执行，这两个get() 函数也将隐式地携带在创建get() 函数时定义的外部变量的值。因此，执行get() 函数时，它会使用原来提供给page() 函数的url 值调用urlopen(url) 。只需很少的检查工作，就能看到闭包中变量的内容，例如：

```
>>> python.__closure__


(<cell at 0x67f50: str object at 0x69230>,)
>>> python.__closure__[0].cell_contents


'http://www.python.org'
>>> jython.__closure__[0].cell_contents


'http://www.jython.org

'
>>>
```

如果需要在一系列函数调用中保持某个状态，使用闭包是一种非常高效的方式。例如，考虑下面运行了一个简单计数器的代码：

```
def countdown(n):
    def next():
        nonlocal n
        r = n
        n -= 1
        return r
    return next

# 用例
next = countdown(10)
while True:
    v = next()       # 获得下一个值
    if not v: break
```

在这段代码中，闭包用于保存内部计数器的值n 。每次调用内部函数next() 时，它都更新并返回这个计数器变量的前一个值。不熟悉闭包的程序员可能会使用下面这样一个类来实现类似的功能：

```
class Countdown(object):
    def __init__(self,n):
        self.n = n
    def next(self):
        r = self.n
        self.n -= 1
        return r

# 示例用法
```

```
c = Countdown(10)
while True:
    v = c.next()       # 获得下一个值
    if not v: break
```

但是，如果增加Countdown()函数的起始值，并执行一次简单的定时基准测试，就会发现使用闭包的版本运行速度要快得多（在作者的计算机上进行测试的结果是快了大约50%）。

闭包会捕捉内部函数的环境，因此还可用于要包装现有函数，以便往应用程序中增加额外功能。接下来介绍这一点。

6.5 装饰器

装饰器 是一个函数，其主要用途是包装另一个函数或类。这种包装的首要目的是光明正大地修改或增强被包装对象的行为。语法上使用 特殊符号@ 表示装饰器，如下所示：

```
@trace
def square(x):
     return x*x
```

上面的代码是下面代码的简化：

```
def square(x):
    return x*x
square = trace(square)
```

这个例子中定义了函数square()。但在定义之后，函数对象本身就立即被传递给函数trace()，后者返回一个对象替代原始的square。现在，让我们看一下trace的实现，从而解释这样做的用处：

```
enable_tracing = True
if enable_tracing:
    debug_log = open("debug.log","w")

def trace(func):
    if enable_tracing:
        def callf(*args,**kwargs):
            debug_log.write("Calling %s: %s, %s\n" %
                            (func.__name__, args, kwargs))
            r = func(*args,**kwargs)
            debug_log.write("%s returned %s\n" % (func.__name, r))
            return r
        return callf
    else:
        return func
```

在这段代码中，trace() 创建了一个包装器函数，它会写入一些调试输出，然后调用原始函数对象。因此如果调用square() 函数，看到的将是包装器中write() 方法的输出。trace() 函数返回的函数callf 是一个闭包，用于替换原始的函数。关于这种实现的一个有趣方面是，跟踪功能本身只能像上面这样使用全局变量enable_tracing 来启用。如果把这个变量置为False ，trace() 装饰器只是返回未修改的原始函数。因此，禁用跟踪时，使用装饰器不会增加性能负担。

使用装饰器时，它们必须出现在函数或类定义之前的单独行上。可以同时使用多个装饰器，例如：

```
@foo
@bar
@spam
def grok(x):
    pass
```

在这个例子中，装饰器将按照它们出现的先后顺序应用，结果等同于：

```
def grok(x):
    pass
grok = foo(bar(spam(grok)))
```

装饰器也可以接受参数，例如：

```
@eventhandler('BUTTON')
def handle_button(msg):
     ...
@eventhandler('RESET')
def handle_reset(msg):
     ...
```

如果提供参数，装饰器的语义如下所示：

```
def handle_button(msg):
     ...
temp = eventhandler('BUTTON')           # 使用提供的参数调用装饰器
handle_button = temp(handle_button)     # 调用装饰器返回的函数
```

在这个例子中，装饰器函数只接受带有@ 描述符的参数。它接着返回一个函数，这个函数在调用的时候以一个函数作为参数。下面举一个例子：

```
# 事件处理程序装饰器
event_handlers = { }
def eventhandler(event):
     def register_function(f):
          event_handlers[event] = f
          return f
     return register_function
```

装饰器也可以应用于类定义，例如：

```
@foo
class Bar(object):
    def __init__(self,x):
        self.x = x
    def spam(self):
        statements
```

对于类装饰器，应该让装饰器函数始终返回类对象作为结果。需要使用原始类定义的代码可能要直接引用类成员，如`Bar.spam`。如果装饰器函数`foo()`返回一个函数，这种引用就是不正确的。

装饰器与函数其他方面的交互行为（如递归、文档字符串和函数属性）有些怪异。本章稍后将会讨论这些问题。

6.6 生成器与 yield

函数使用`yield`关键字可以定义生成器 对象。生成器是一个函数，它生成一个值的序列，以便在迭代中使用，例如：

```
def countdown(n):
     print("Counting down from %d" % n)
     while n > 0:
          yield n
          n -= 1
     return
```

如果调用该函数，就会发现其中的代码不会开始执行，例如：

```
>>> c = countdown(10)


>>>
```

相反它会返回一个生成器对象。接着，该生成器对象就会在`next()`被调用（在Python 3中是`__next__()`）时执行函数。例如：

```
>>> c.next()

 # 在Python 3中请使用c.__next__()
Counting down from 10
10
>>> c.next()
```

```
9
```

调用next() 时，生成器函数将开始执行语句，直至遇到yield 语句为止。yield 语句在函数执行停止的地方生成一个结果，直到再次调用next() 。然后继续执行yield 之后的语句。

通常不会在生成器上直接调用next() 方法，而是通过for 语句、sum() 或一些消耗序列的其他操作使用生成器。例如：

```
for n in countdown(10):
    statements


a = sum(countdown(10))
```

生成器函数完成的标志是返回或引发StopIteration 异常，这标志着迭代的结束。如果生成器在完成时返回None 之外的值，都是不合法的。

生成器使用时存在一个棘手的问题，即生成器函数仅被部分消耗。例如，请看以下代码：

```
for n in countdown(10):
     if n == 2: break
     statements
```

在这个例子中，通过调用break 退出for 循环，而相关的生成器也没有全部完成。为了处理这种情况，生成器对象提供方法close() 标识关闭。不再使用或删除生成器时，就会调用close() 方法。通常不必手动调用close() 方法，但也可以这么做，例如：

```
>>> c = countdown(10)


>>> c.next()


Counting down from 10
10
>>> c.next()


9
>>> c.close()
```

```
>>> c.next()


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

在生成器函数内部，在yield 语句上出现GeneratorExit 异常时就会调用close() 方法。也可以选择捕捉这个异常，以便执行清理操作。

```
def countdown(n):
    print("Counting down from %d" % n)
    try:
         while n > 0:
              yield n
              n = n - 1
    except GeneratorExit:
         print("Only made it to %d" % n)
```

虽然可以捕捉GeneratorExit 异常，但对于生成器函数而言，使用yield 语句处理异常并生成另一个输出值是不合法的。另外，如果程序当前正在对生成器进行迭代，不应通过另一个的执行线程或从信号处理程序异步调用该生成器上的 close() 方法。

6.7 协程与 yield 表达式

在函数内，yield 语句还可以作为表达式使用，出现在赋值运算符的右边，例如：

```
def receiver():
    print("Ready to receive")
    while True:
          n = (yield)
          print("Got %s" % n)
```

以这种方式使用yield 语句的函数称为协程 ，向函数发送值时函数将执行。它的行为也十分类似于生成器，例如：

```
>>> r = receiver()


>>> r.next()

  # 向前执行到第一条yield语句（在Python 3中是r.__next__()）
Ready to receive
>>> r.send(1)


Got 1
>>> r.send(2)
```

```
Got 2
>>> r.send("Hello")


Got Hello
>>>
```

在这个例子中，一开始调用next() 是必不可少的，这样协程才能执行第一个yield 表达式之前的语句。这时，协程会挂起，等待相关生成器对象r 的send() 方法给它发送一个值。传递给send() 的值由协程中的(yield) 表达式返回。接收到值后，协程就会执行语句，直至遇到下一条yield 语句。

在协程中需要首先调用next() 这件事情很容易被忽略，这经常成为错误出现的原因。因此，建议使用一个能自动完成该步骤的装饰器来包装协程。

```
def coroutine(func):
   def start(*args,**kwargs):
        g = func(*args,**kwargs)
        g.next()
        return g
   return start
```

使用这个装饰器就可以像下面这样编写和使用协程：

```
@coroutine
def receiver():
    print("Ready to receive")
    while True:
          n = (yield)
          print("Got %s" % n)
# 示例用法
r = receiver()
r.send("Hello World")       # 注意：无需初始调用.next()方法
```

协程一般会不断地执行下去，除非被显式关闭或者自己退出。像下面这样使用方法close() 可以关闭输入值的流：

```
>>> r.close()


>>> r.send(4)


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

关闭后，如果继续给协程发送值，就会引发StopIteration 异常。正如前面关于生成器的内容中讲到的那样，close() 操作将在协程内部引发GeneratorExit 异常，例如：

```
def receiver():
    print("Ready to receive")
    try:
        while True:
            n = (yield)
            print("Got %s" % n)
    except GeneratorExit:
        print("Receiver done")
```

可以使用throw(*exctype* [, *value* [, *tb*]]) 方法在协程内部引发异常，其中 *exctype* 是指异常类型， *value* 是指异常的值，而 *tb* 是指跟踪对象。例如：

```
>>> r.throw(RuntimeError,"You're hosed!")


Traceback (most recent call last):


File "<stdin>", line 1, in <module>


File "<stdin>", line 4, in receiver
RuntimeError: You're hosed!
```

以这种方式引发的异常将在协程中当前执行的yield 语句处出现。协程可以选择捕捉异常并以正确方式处理它们。使用throw() 方法作为给协程的异步信号并不安全——永远都不应该通过单独的执行线程或信号处理程序调用这个方法。

如果yield 表达式中提供了值，协程可以使用yield 语句同时接收和发出返回值，例如：

```
def line_splitter(delimiter=None):
    print("Ready to split")
    result = None
    while True:
        line = (yield result)
        result = line.split(delimiter)
```

在这个例子中，我们使用协程的方式与前面相同。但是，现在调用send() 方法也会生成一个结果，例如：

```
>>> s = line_splitter(",")


>>> s.next()
```

```
Ready to split
>>> s.send("A,B,C")


['A', 'B', 'C' ]
>>> s.send("100,200,300")


['100', '200', '300']
>>>
```

理解这个例子中的先后顺序至关重要。首个next() 调用让协程向前执行到(yield result)，这将返回result 的初始值None 。在接下来的send() 调用中，接收到的值被放在line 中并拆分到result 中。send() 方法的返回值就是传递给下一条yield 语句的值。换句话说，send() 方法的返回值来自下一个yield 表达式，而不是接收send() 传递的值的yield 表达式。

如果协程返回值，需要小心处理使用throw() 引发的异常。如果使用throw() 在协程中引发一个异常，传递给协程中下一条yield 语句的值将作为throw() 方法的结果返回。如果需要这个值却又忘记保存它，它就会消失不见。

6.8 使用生成器与协程

乍一看，如何使用生成器和协程解决实际问题似乎并不明显。但在解决系统、网络和分布式计算方面的某些编程问题时，生成器和协程特别有用。例如，生成器函数可用于建立一个处理管道（本质上类似于在UNIX shell中使用一个管道）。第1章中给出一个这样的例子。下面给出另一个例子，其中包括关于查找、打开、读取和处理文件的一组生成器函数：

```
import os
import fnmatch

def find_files(topdir, pattern):
    for path, dirname, filelist in os.walk(topdir):
          for name in filelist:
              if fnmatch.fnmatch(name, pattern):
                  yield os.path.join(path,name)

import gzip, bz2
def opener(filenames):
    for name in filenames:
        if name.endswith(".gz"): f = gzip.open(name)
        elif name.endswith(".bz2"): f = bz2.BZ2File(name)
        else: f = open(name)
        yield f

def cat(filelist):
    for f in filelist:
        for line in f:
```

```
            yield line

def grep(pattern, lines):
    for line in lines:
        if pattern in line:
            yield line
```

下面的例子使用这些函数建立了一个处理管道：

```
wwwlogs = find("www","access-log*")
files   = opener(wwwlogs)
lines   = cat(files)
pylines = grep("python", lines)
for line in pylines:
    sys.stdout.write(line)
```

在这个例子中，程序要处理的是顶级目录"www "的所有子目录中的所有"access-log* "文件中的全部行。程序将测试每个"access-log "文件的文件压缩情况，然后使用正确的文件打开器打开它们。程序将各行连接在一起，并通过查找子字符串“python ”的过滤器进行处理。整个程序是由位于最后的for 语句驱动的。该循环的每次迭代都会通过管道获得一个新值并使用之。此外，这种实现占用内存极少，因为它无需创建任何临时列表或其他大型的数据结构。

协程可用于编写数据流处理程序。以这种方式组织的程序像是反转的管道。你将值发送到一些相互连接的协程中，而不是通过一系列使用for 循环的生成器函数获取值。下面给出了一个例子，其中的协程函数模拟了前面给出的生成器函数：

```
import os
import fnmatch

@coroutine
def find_files(target):
    while True:
        topdir, pattern = (yield)
        for path, dirname, filelist in os.walk(topdir):
            for name in filelist:
                if fnmatch.fnmatch(name,pattern):
                    target.send(os.path.join(path,name))

import gzip, bz2
@coroutine
def opener(target):
    while True:
        name = (yield)
        if name.endswith(".gz"): f = gzip.open(name)
        elif name.endswith(".bz2"): f = bz2.BZ2File(name)
        else: f = open(name)
        target.send(f)

@coroutine
def cat(target):
    while True:
        f = (yield)
```

```
        for line in f:
            target.send(line)
@coroutine
def grep(pattern, target):
    while True:
        line = (yield)
        if pattern in line:
            target.send(line)

@coroutine
def printer():
    while True:
        line = (yield)
        sys.stdout.write(line)
```

以下代码说明了如何将这些协程连接起来，创建一个数据流处理管道：

```
finder = find_files(opener(cat(grep("python",printer()))))

# 现在发送一个值
finder.send(("www","access-log*"))
finder.send(("otherwww","access-log*"))
```

在这个例子中，每个协程都发送数据给在它们的target 参数中指定的另一个协程。和生成器的例子不同，执行完全由将数据发送到第一个协程find_files() 中来驱动。接下来，这个协程将数据转入下一阶段。这个例子有一个关键的地方，即协程管道永远保持活动状态，直到它显式调用close() 方法为止。因此，只要需要，程序可以不断地给协程中注入数据，例如本例中对于send() 方法的两次重复调用。

协程可用于实现某种形式的并发。例如，一个集中式的任务管理器或事件循环，可以安排并将数据发送到成百上千个用于执行各种处理任务的协程中。输入数据“被发送”到协程中这个事实还说明，若程序使用消息队列和消息传递在组件之间进行通信，协程可以很容易地与之在一起混合使用。第20章将进一步介绍此方面的内容。

6.9 列表推导

用到函数的一个常见操作是将其用于一个列表的所有项，并使用结果创建一个新列表，例如：

```
nums = [1, 2, 3, 4, 5]
squares = []
for n in nums:
    squares.append(n * n)
```

这种操作很常见，因此出现了叫做列表推导 的运算符，举一个简单的例子：

```
nums = [1, 2, 3, 4, 5]
squares = [n * n for n in nums]
```

列表推导的一般语法如下所示：

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN ]
```

这种语法大致上等价于以下代码：

```
s = []
for item1

 in iterable1

:
    if condition1

:
        for item2

 in iterable2

:
          if condition2

:
              ...
              for itemN

 in iterableN

:
                  if conditionN

: s.append(expression

)
```

为了解释清楚，下面再列举一些例子：

```
a = [-3,5,2,-10,7,8]
b = 'abc'

c = [2*s for s in a]           # c = [-6,10,4,-20,14,16]
d = [s for s in a if s >= 0]   # d = [5,2,7,8]
e = [(x,y) for x in a          # e = [(5,'a'),(5,'b'),(5,'c'),
          for y in b           #      (2,'a'),(2,'b'),(2,'c'),
          if x > 0 ]           #      (7,'a'),(7,'b'),(7,'c'),
                               #      (8,'a'),(8,'b'),(8,'c')]
f = [(1,2), (3,4), (5,6)]
g = [math.sqrt(x*x+y*y)        # f = [2.23606, 5.0, 7.81024]
     for x,y in f]
```

提供给列表推导的序列其长度不必相同，因为从上面的代码可以看出，我们使用了一组嵌套的`for`循环来迭代它们的内容。结果列表包含了各个表达式的运算值。`if`子句是可选的，但如果使用它，那么只有 *`condition`* 为真的时候才会对 *`expression`* 求值并添加到结果中。

如果使用列表推导构造元组列表，则元组值必须放在圆括号中。例如，`[(x, y) for x in a for y in b]`是合法的语法，而`[x, y for x in a for y in b]`是不合法的。

最后，在Python 2中，定义在列表推导中的迭代变量是在当前作用域中进行求值的，其定义在列表推导执行完成之后仍然保留，这一点很重要。例如，在`[x for x in a]`中，迭代变量`x`改写了前面定义的`x`的值，并在创建结果列表后将其设为`a`中最后一项的值。所幸，Python 3中的情况并不是这样——迭代变量一直是私有变量。

6.10 生成器表达式

生成器表达式 是一个对象，它执行的计算与列表推导相同，但会迭代地生成结果。它的语法也与列表推导相同，但要用圆括号代替方括号，例如：

```
(expression
 for item1
 in iterable1 if condition1

             for item2
 in iterable2
 if condition2

             ...
             for itemN in iterableN if conditionN
)
```

和列表推导不同，生成器表达式实际上不创建列表或者立即对圆括号内的表达式求值。相反，它会创建一个生成器对象，该对象通过迭代并按照需要生成值，例如：

```
>>> a = [1, 2, 3, 4]

>>> b = (10*i for i in a)

>>> b
```

```
<generator object at 0x590a8>
>>> b.next()


10
>>> b.next()


20
...
```

列表与生成器表达式之间的差异十分重要，但很微妙。使用列表推导时，Python实际上创建了包含结果数据的列表。而使用生成器表达式时，Python创建的是只知道如何按照需要生成数据的生成器。在某些应用中，这可能极大地提高性能和内存使用。例如：

```
# 读取一个文件
f = open("data.txt")                                # 打开一个文件
lines = (t.strip() for t in f)                      # 读取行，并删除前后空白

comments = (t for t in lines if t[0] == '#')     # 所有注释
for c in comments:
    print(c)
```

在这个例子中，生成器表达式提取各行并删除其中的空白，但它实际上没有将整个文件读取到内存中。提取注释的表达式也是如此。相反，当程序开始在`for`循环中进行迭代时，才去读取文件的各行。在这个迭代过程中，每一行都是按需生成的，按条件进行了过滤。事实上，该过程从未把整个文件加载到内存中。因此，这是一种从GB级别大小的Python源文件中提取注释的高效方法。

和列表推导不同，生成器表达式不会创建序列形式的对象。你不能对它进行索引，也不能进行任何常规的列表操作，例如`append()`。但是，使用内置的`list()`函数可以将生成器表达式转换为列表：

```
clist = list(comments)
```

6.11 声明式编程

列表推导和生成器表达式与声明式语言中的操作有着很强的联系。事实上，这些特性在一定程度上源自数学集合论。例如，编写像`[x*x for x in a if x > 0]`这样的语句时，有点类似于指定一个集合`{ x2 | x ∈a, x > 0 }`。

可以使用这些声明式特性将程序的结构组织为一系列可以同时操作所有数据的计算，而无需编写手动迭代数据的程序。例如，假定有一个文件portfolio.txt，它包含下面的股票数据：

```
AA 100 32.20
IBM 50 91.10
CAT 150 83.44
MSFT 200 51.23
GE 95 40.37
MSFT 50 65.10
IBM 100 70.44
```

下面这个声明式程序对第二列与第三列的乘积进行求和，从而计算出总价：

```
lines = open("portfolio.txt")
fields = (line.split() for line in lines)
print(sum(float(f[1]) * float(f[2]) for f in fields))
```

在这个程序中，我们并不关心怎样对文件的每行进行循环。相反，我们只是声明了一系列在所有数据上执行的计算。这种方法不仅让代码十分紧凑，而且往往比下面这种传统做法的运行速度更快：

```
total = 0
for line in open("portfolio.txt"):
    fields = line.split()
    total += float(fields[1]) * float(fields[2])
print(total)
```

声明式编程与程序员在UNIX shell中执行的某些操作有着某种程度上的联系。例如，前面使用生成器表达式的例子类似于下面这个单行awk 命令：

```
% awk '{ total += $2 * $3} END { print total }' portfolio.txt


44671.2
%
```

列表推导和生成器表达式的声明式还可以用于模拟数据库处理中常用的SQL select 语句的行为。例如，看看下面这些例子，它们处理的是读入到字典列表中的数据：

```
fields = (line.split() for line in open("portfolio.txt"))
portfolio = [ {'name' : f[0],
               'shares' : int(f[1]),
               'price' : float(f[2]) }
              for f in fields]
# 一些查询
msft = [s for s in portfolio if s['name'] == 'MSFT']
large_holdings = [s for s in portfolio
                      if s['shares']*s['price'] >= 10000]
```

事实上，如果使用与数据库访问相关的模块（参见第17章），经常可以同时使用列表

推导和数据库查询，例如：

```
sum(shares*cost for shares,cost in
        cursor.execute("select shares, cost from portfolio")
          if shares*cost >= 10000)
```

6.12 lambda 运算符

使用lambda 语句可以创建表达式形式的匿名函数：

```
lambda args

 : expression
```

args 是以逗号分隔的一列参数，而 *expression* 是用到这些参数的表达式，例如：

```
a = lambda x,y : x+y
r = a(2,3)            # r的值为5
```

使用lambda 语句定义的代码必须是合法的表达式。lambda 语句中不能出现多条语句和其他非表达式语句，如for 和while 。lambda 表达式遵循与函数相同的作用域规则。

lambda 的首要用途是指定短小的回调函数。例如，如果要在不考虑大小写的情况下对一列名称进行排序，代码可以这样写：

```
names.sort(key=lambda n: n.lower())
```

6.13 递归

定义递归函数很容易，例如：

```
def factorial(n):
    if n <= 1: return 1
    else: return n * factorial(n - 1)
```

但是要注意，Python对于递归函数调用的深度做了限制。函数sys.getrecursionlimit() 返回当前最大的递归深度，而函数sys.setrecursionlimit() 可用于修改这个值。默认值为1000 。尽管可以增大这个值，但程序仍然会受主机操作系统使用的栈大小限制。超出递归深度时，就会引发

RuntimeError 异常。和其他函数式编程语言（如Scheme）不同，Python不会进行尾递归优化。

递归不能用在生成器函数和协程中。例如，下面这段代码打印了一个嵌套列表集中的所有项：

```
def flatten(lists):
    for s in lists:
        if isinstance(s,list):
            flatten(s)
        else:
            print(s)
items = [[1,2,3],[4,5,[5,6]],[7,8,9]]
flatten(items)      # 打印结果为1 2 3 4 5 6 7 8 9
```

但是，如果将print 操作改为yield 语句，这段代码就无法工作。这是因为对 flatten() 函数的递归调用只会创建一个新的生成器对象，而不会实际迭代它。下面给出了递归生成器的有效版本：

```
def genflatten(lists):
    for s in lists:
        if isinstance(s,list):
            for item in genflatten(s):
                yield item
        else:
            yield item
```

还要当心混合使用递归函数和装饰器的问题。如果对递归函数使用装饰器，所有内部的递归调用都会通过装饰后的版本进行，例如：

```
@locked
def factorial(n):
    if n <= 1: return 1
    else: return n * factorial(n - 1) # 调用factorial函数的已包装版本
```

如果使用装饰器的目的是进行一些系统管理，如同步或锁定，最好不要使用递归。

6.14 文档字符串

习惯上我们会将函数的第一条语句写成文档字符串，用于描述函数的用途，例如：

```
def factorial(n):
    """Computes n factorial. For example:

       >>> factorial(6)
       120
       >>>
    """
    if n <= 1: return 1
```

```
    else: return n*factorial(n-1)
```

文档字符串保存在函数的__doc__ 属性中，IDE通常使用该函数提供交互式帮助。

如果需要使用装饰器，要注意使用装饰器包装函数可能会破坏与文档字符串相关的帮助功能。例如，考虑以下代码：

```
def wrap(func):
    call(*args,**kwargs):
        return func(*args,**kwargs)
    return call
@wrap
def factorial(n):
    """Computes n factorial."""
    ...
```

如果用户请求这个版本的factorial() 函数的帮助，将会看到一种相当诡异的解释：

```
>>> help(factorial)


Help on function call in module __main__:

call(*args, **kwargs)
(END)
>>>
```

这个问题的解决办法是编写可以传递函数名称和文档字符串的装饰器函数，例如：

```
def wrap(func):
    call(*args,**kwargs):
            return func(*args,**kwargs)
        call.__doc__ = func.__doc__
        call.__name__ = func.__name__
        return call
```

因为这是一个常见问题，所以functools 模块提供了函数wraps ，用于自动复制这些属性。显而易见，它也是一个装饰器：

```
from functools import wraps
def wrap(func):
    @wraps(func)
    call(*args,**kwargs):
        return func(*args,**kwargs)
    return call
```

functools 模块中定义的@wraps(func) 装饰器可以将属性从 *func* 传递给要定义的包装器函数。

6.15 函数属性

可以给函数添加任意属性，例如：

```
def foo():
    statements



foo.secure = 1
foo.private = 1
```

函数属性保存在函数的__dict__ 属性中，并以字典格式存储。

函数属性主要用在高度专用的应用程序中，如语法分析器生成器（parser generator）和要给函数对象附加额外信息的应用程序框架。

和文档字符串一样，也要注意混合使用函数属性和装饰器的问题。如果使用装饰器包装函数，实际上是由装饰器函数而非原始函数来访问属性。考虑到实际应用，这可能是也可能不是你想要的结果。要将已经定义的函数属性传递给装饰器函数，使用以下模板或者前面内容中提到的functools.wraps() 装饰器：

```
def wrap(func):
    call(*args,**kwargs):
        return func(*args,**kwargs)
    call.__doc__ = func.__doc__
    call.__name__ = func.__name__
    call.__dict__.update(func.__dict__)
    return call
```

6.16 eval()、exec() 和compile() 函数

eval(*str* [, *globals* [, *locals*]]) 函数执行一个表达式字符串并返回结果，例如：

```
a = eval('3*math.sin(3.5+x) + 7.2')
```

类似地，exec(*str* [, *globals* [, *locals*]]) 函数执行一个包含任意Python代码的字符串。提供给exec() 的代码在执行时并无区别，就好像exec 语句这个地方本来就有这些代码一样。例如：

```
a = [3, 5, 10, 13]
```

```
exec("for i in a: print(i)")
```

关于exec 函数有一条注意事项，即在Python 2中，exec 函数实际上被定义为一条语句。因此在遗留代码中，可能会看到不使用圆括号调用exec 函数的语句，如exec "for i in a: print i" 。尽管这种语句在Python 2.6中仍然有效，但在Python 3中使用会报错。现代程序应该把exex() 当作函数来使用。

这两个函数都会在调用者的命名空间中执行，该命名空间用于解析出现在字符串或文件中的任意符号。eval() 和exec() 函数可以接受一个或两个可选的映射对象，分别用作代码执行的全局和局部命名空间，例如：

```
globals = {'x': 7,
           'y': 10,
           'birds': ['Parrot', 'Swallow', 'Albatross']
           }
locals = { }

# 执行时使用上面的字典作为全局和局部命名空间
a = eval("3 * x + 4 * y", globals, locals)
exec("for b in birds: print(b)", globals, locals)
```

如果省略其中一个或两个命名空间，就会使用全局和局部命名空间的当前值。另外，由于嵌套作用域存在一些问题，如果该函数也包含嵌套的函数定义或使用lambda 运算符的话, 在函数体内使用exec() 函数可能会导致SyntaxError 异常。

给exec() 或eval() 函数传递字符串时，语法分析器首先会把这个字符串编译为字节码。因为这个过程十分耗资源，如果代码要反复执行多次，最好是预编译代码，然后在后续的调用中重用字节码。

compile(*str* , *filename* , *kind*) 函数将字符串编译为字节码，其中 *str* 是包含要编译代码的字符串，而 *filename* 是定义该字符串的文件（在跟踪生成中使用）。 *kind* 参数指定了要编译代码的类型single 代表一条语句，exec 代表一组语句，而eval 代表一个表达式。还可以将compile() 函数返回的代码对象传递给eval() 函数和exec() 语句，例如：

```
s = "for i in range(0,10): print(i)"
c = compile(s,'','exec')       # 编译为代码对象
exec(c)                         # 执行它

s2 = "3 * x + 4 * y"
c2 = compile(s2, '', 'eval')   # 编译为表达式
result = eval(c2)               # 执行它
```

第7章　类与面向对象编程

类是创建新对象的机制。本章将详细介绍类，但并不打算深入分析面向对象编程与设计。读者需要了解数据结构，并具备使用C或Java等其他语言进行面向对象编程的一定经验（第3章介绍了关于对象术语和对象内部实现的更多信息）。

7.1　class 语句

类 定义了一组属性，这些属性与一组叫做实例 的对象相关且由其共享。类通常是由函数（称为方法 ， method）、变量（称为类变量 ， class variable）和计算出的属性（称为特性 ， property）组成的集合。

使用class 语句可定义类。类主体包含一系列在类定义时执行的语句，例如：

```
class Account(object):
    num_accounts = 0
    def __init__(self,name,balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1
    def __del__(self):
        Account.num_accounts -= 1
    def deposit(self,amt):
        self.balance = self.balance + amt
    def withdraw(self,amt):
        self.balance = self.balance - amt
    def inquiry(self):
        return self.balance
```

在类主体执行期间创建的值放在类对象中，这个对象充当着命名空间，与模块极为相似。例如，访问Account 类成员的方式如下：

```
Account.num_accounts
Account.__init__
Account.__del__
Account.deposit
Account.withdraw
Account.inquiry
```

需要注意的是，class 语句本身并不创建该类的任何实例（例如，上一个例子实际上不会创建任何账户）。类仅设置将在以后创建的所有实例使用的属性。从这种意义上讲，可以将类看作一个蓝图。

类中定义的函数称为实例方法 。实例方法是一种在类的实例上进行操作的函数，类实例作为第一个参数传递。根据约定，这个参数称为self ，尽管所有合法的标识符都可以使用。在前面的例子中，deposit() 、withdraw() 和inquiry() 都是实例方法。

类变量（如num_accounts ）是可在类的所有实例之间共享的值（也就是说，它们不是单独分配给每个实例的）。上例中的num_accounts 变量用于跟踪存在多少个Account 实例。

7.2 类实例

类的实例是以函数形式调用类对象来创建的。这种方法将创建一个新实例，然后将该实例传递给类的__init__() 方法。__init__() 方法的参数包括新创建的实例self 和在调用类对象时提供的参数。例如：

```
# 创建一些账户
a = Account("Guido", 1000.00)   # 调用Account.__init__(a,"Guido",1000.00)
b = Account("Bill", 10.00)
```

在__init__() 内，通过将属性分配给self 来将其保存到实例中。例如，self.name = name 表示将name 属性保存在实例中。在新创建的实例返回到用户之后，使用点（.）运算符即可访问这些属性以及类的属性，如下所示：

```
a.deposit(100.00)      # 调用Account.deposit(a,100.00)
b.withdraw(50.00)      # 调用Account.withdraw(b,50.00)
name = a.name          # 获取账户名称
```

点（.）运算符用于属性绑定。访问属性时，结果值可能来自多个不同的地方。例如，上例中的a.name 返回实例a 的name 属性。而a.deposit 返回Account 类的deposit 属性（一个方法）。访问属性时，Python首先会检查实例，如果不知道该属性的任何信息，则会对实例的类进行搜索。这是类与其所有实例共享其属性的底层机制。

7.3 作用域规则

尽管类会定义命名空间，但它们不会为方法中用到的名称创建作用域。所以在实现类时，对属性和方法的引用必须是完全限定的。例如，在方法中，只能通过self 引用实例的属性。所以，在前面的例子中使用的是self.balance ，而不是balance 。如果希望从一个方法中调用另一个方法，也可以采用这种方式，如下所示：

```
class Foo(object):
    def bar(self):
        print("bar!")
    def spam(self):
        bar(self)         # 错误！"bar"生成了一个NameError
        self.bar()        # 这条语句能够正常运行
        Foo.bar(self)     # 这条语句也能够正常运行
```

类中没有作用域，这是Python与C++或Java的区别之一。如果你以前使用过这些语言，你会发现Python中的self 参数与this 指针是相同的。需要显式使用self 的原因在于，Python没有提供显式声明变量的方式（如C中int x 或float y 等声明）。因此，无

法知道在方法中要赋值的变量是不是局部变量，或者是否要保存为实例属性。显式使用self 可以解决该问题，存储在self 中的所有值都是实例的一部分，所有其他赋值都是局部变量。

7.4 继承

继承 是一种创建新类的机制，目的是使用或修改现有类的行为。原始类称为基类 或超类 。新类称为派生类 或子类 。通过继承创建类时，所创建的类将“继承”其基类定义的属性。但是，派生类可以重新定义任何这些属性并添加自己的新属性。

在class 语句中用以逗号分隔的基类名称列表来指定继承。如果没有有效的基类，类将继承object ，如前面的例子所示。object 是所有Python对象的根类，提供了一些常见方法（如__str__() ，它可创建供打印函数使用的字符串）的默认实现。

继承通常用于重新定义现有方法的行为。在下面的例子中，一个特殊版的Account 重新定义了inquiry() 方法，让它周期性输出比实际更高的余额，这样的话，如果用户没有密切注意账户情况，当他在支付次级抵押贷款时就可能会透支账户，从而招致高额的罚金。

```
import random
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
           return self.balance * 1.10  # 注意：正在为这个创意申请专利
        else:

           return self.balance

c = EvilAccount("George", 1000.00)
c.deposit(10.0)   # 调用Account.deposit(c,10.0)
available = c.inquiry()   # 调用EvilAccount.inquiry(c)
```

在这个例子中，除了重新定义inquiry() 方法外，EvilAccount 的实例与Account 的实例完全相同。

继承是用功能稍微增强的点（.）运算符实现的。具体来讲，如果搜索一个属性时未在实例或实例的类中找到匹配项，将会继续搜索基类。这个过程会一直继续下去，直到没有更多的基类可供搜索为止。这就是为什么在上一个例子中，c.deposit() 调用了在Account 类中定义的deposit() 的实现。

子类可以定义自己的__init__() 函数，从而向实例添加新属性。例如，下面的EvilAccount 版本添加了新属性evilfactor ：

```
class EvilAccount(Account):
    def __init__(self,name,balance,evilfactor):
        Account.__init__(self,name,balance)  # 初始化Account
        self.evilfactor = evilfactor
    def inquiry(self):
        if random.randint(0,4) == 1:
```

```
            return self.balance * self.evilfactor
        else:
            return self.balance
```

派生类定义__init__()时，不会自动调用基类的__init__()方法。因此，要由派生类调用基类的__init__()方法来对它们进行恰当的初始化。在上一个例子中，可以从调用Account.__init__()的语句中看到这一点。如果基类未定义__init__()，就可以忽略这一步。如果不知道基类是否定义了__init__()，可在不提供任何参数的情况下调用它，因为始终存在一个不执行任何操作的默认__init__()实现。

有时，派生类将重新实现方法，但是还想调用原始的实现。为此，有一种方法可以显式地调用基类中的原始方法，将实例self作为第一个参数传递即可，如下所示：

```
class MoreEvilAccount(EvilAccount):
    def deposit(self,amount):
        self.withdraw(5.00)                 # 减去“便利”费
        EvilAccount.deposit(self,amount)    # 现在进行存款
```

这个例子的微妙之处在于，EvilAccount这个类其实没有实现deposit()方法。该方法是在Account类中实现的。尽管这段代码能够运行，但它可能会引起一些读者的混淆（例如，EvilAccount是否应该实现deposit()?）。因此，替代解决方案是用super()函数，如下所示：

```
class MoreEvilAccount(EvilAccount):
    def deposit(self,amount):
        self.withdraw(5.00)                                 # 减去便利费
        super(MoreEvilAccount,self).deposit(amount)    # 现在进行存款
```

super(cls, instance)会返回一个特殊对象，该对象支持在基类上执行属性查找。如果使用该函数，Python将使用本来应该在基类上使用的正常搜索规则来搜索属性。有了这种方式，就无需写死方法位置，并且能更清晰地陈述你的意图（即你希望调用以前的实现，无论它是哪个基类定义的）。然而，super()的语法尚有不足之处。如果使用Python 3，可以使用简化的语句super().deposit (amount)来执行上面示例中的计算。但在Python 2中，必须使用更冗繁的版本。

Python支持多重继承。通过让一个类列出多个基类即可指定多重继承。例如，下面给出了一个类集合：

```
class DepositCharge(object):
    fee = 5.00
    def deposit_fee(self):
        self.withdraw(self.fee)

class WithdrawCharge(object):
    fee = 2.50
    def withdraw_fee(self):
        self.withdraw(self.fee)
```

```
# 使用了多重继承的类
class MostEvilAccount(EvilAccount, DepositCharge, WithdrawCharge):
    def deposit(self,amt):
        self.deposit_fee()
        super(MostEvilAccount,self).deposit(amt)
    def withdraw(self,amt):
        self.withdraw_fee()
        super(MostEvilAcount,self).withdraw(amt)
```

使用多重继承时，属性的解析会变得非常复杂，因为可以使用很多搜索路径来绑定属性。为了说明解析的复杂性，我们来看以下语句：

```
d = MostEvilAccount("Dave",500.00,1.10)
d.deposit_fee()    # 调用DepositCharge.deposit_fee()。费用是5.00
d.withdraw_fee()   # 调用WithdrawCharge.withdraw_fee()。费用是5.00??
```

在这个例子中，像deposit_fee() 和withdraw_fee() 这样的方法命名都是唯一的，并且可以在各自的基类中找到。但是，withdraw_fee() 函数似乎没有正常工作，因为它并未实际使用在自己的类中初始化的fee 的值。事实是，在两个不同的基类中都定义了类变量fee 。程序中使用了其中的一个值，但使用的到底是哪个值呢？（提示：是DepositCharge.fee ）。

在查找使用了多重继承的属性时，会将所有基类按从“最特殊”的类到“最不特殊”的类这种顺序进行排列。然后在搜索属性时，就会按这个顺序搜索，直至找到该属性的第一个定义。在上面的例子中，类EvilAccount 比Account 更特殊，因为它继承自Account 。同样，在MostEvilAccount 中，DepositCharge 比WithdrawCharge 更特殊，因为它排在基类列表中的第一位。对于任何给定的类，通过打印它的__mro__ 属性即可查看基类的顺序，例如：

```
>>> MostEvilAccount.__mro__
(<class '__main__.MostEvilAccount'>,
 <class '__main__.EvilAccount'>,
 <class '__main__.Account'>,
 <class '__main__.DepositCharge'>,
 <class '__main__.WithdrawCharge'>,
 <type 'object'>)
>>>
```

在大多数情况下，这个列表基于“有意义”的规则排列得出。也就是说，始终先检查派生类，然后再检查其基类，如果一个类具有多个父类，那么始终按类定义中列出的父类顺序检查这些父类。但是，基类的准确顺序实际上非常复杂，不是基于任何“简单的”算法，如深度优先或广度优先搜索。实际上，基类的顺序由C3线性化算法确定，可以在论文“A Monotonic Superclass Linearization for Dylan”（K. Barrett等，发表于OOPSLA’96）中找到该算法的介绍。该算法的一个需要注意的地方是，某些类层次结构将被Python拒绝并会抛出TypeError 错误，例如：

```
class X(object): pass
```

```
class Y(X): pass
class Z(X,Y): pass    # TypeError。
                       # 无法创建一致的方法解析顺序__
```

在这个例子中，方法解析算法拒绝创建类Z，因为它无法确定合理的基类顺序。例如，在继承列表中，类X出现在类Y前面，所以必须首先检查类X。但是，类Y更特殊，因为它继承自类X。因此，如果首先检查X，就不可能解析Y中更为特殊的方法。实际上这种问题应该很罕见——如果出现，通常表明程序存在更为严重的设计问题。

一般来说，在大多数程序中最好避免使用多重继承。但是，多重继承有时可用于定义所谓的混合（mixin）类。混合类通常定义了要“混合到”其他类中的一组方法，目的是添加更多的功能（这与宏很类似）。通常，混合类中的方法将假定其他方法存在，并将以这些方法为基础构建。前面例子中的DepositCharge和WithdrawCharge类就是例证。这些类将向其子类中添加新方法（如deposit_fee()）。但是，DepositCharge这个类永远不会被实例化。实际上，如果你实例化了该类，它并不会创建具有任何用途的实例，也就是说，定义的方法甚至不会正确执行。

最后还要注意一点，如果希望解决本例中存在问题的fee引用，应该将deposit_fee()和withdraw_fee()的实现改为直接使用类名引用该属性，而不是用self（如DepositChange.fee）。

7.5 多态动态绑定和鸭子类型

动态绑定（在继承背景下使用时，有时也称为多态性）是指在不考虑实例类型的情况下使用实例。动态绑定完全由继承属性查找过程处理，这在上一节中已有介绍。只要以*obj*.attr的形式访问属性，就会按照一定的顺序搜索并定位attr：首先是实例本身，接着是实例的类定义，然后是基类。查找过程会返回第一个匹配项。

这种绑定过程的关键在于，它不受对象*obj*的类型影响。因此，如果执行像*obj*.name这样的查找，对所有拥有name属性的*obj*都是适用的。这种行为有时被称为“鸭子类型”（duck typing），这个名称来源于一句谚语：“如果看起来像、叫声像而且走起路来像鸭子，那么它就是鸭子。”

Python程序员经常编写利用这种行为的程序。例如，如果想编写现有对象的自定义版本，可以继承该对象，也可以创建一个外观和行为像它但与它无任何关系的全新对象。后一种方法通常用于保持程序组件的松散耦合。例如，可以编写代码来处理任何种类的对象，只要该对象拥有特定的方法集。最常见的例子就是利用标准库中定义的各种“类似文件”的对象。尽管这些对象的工作方式像文件，但它们并不是继承自内置文件对象的。

7.6 静态方法和类方法

在类定义中，所有函数都被假定在实例上操作，该实例总是作为第一个参数self传递。但是，还可以定义两种常见的方法。

静态方法是一种普通函数，只不过它们正好位于类定义的命名空间中。它不会对任何实例类型进行操作。要定义静态方法，可使用@staticmethod装饰器，如下所示：

```
class Foo(object):
    @staticmethod
    def add(x,y):
        return x + y
```

要调用静态方法，只需用类名作为它的前缀。无需向它传递任何其他信息，例如：

```
x = Foo.add(3,4)     # x = 7
```

如果在编写类时需要采用很多不同的方式来创建新实例，则常常使用静态方法。因为类中只能有一个`__init__()`函数，所以替代的创建函数通常按如下方式定义：

```
class Date(object):
    def __init__(self,year,month,day):
        self.year = year
        self.month = month
        self.day = day
    @staticmethod
    def now():
         t = time.localtime()
         return Date(t.tm_year, t.tm_mon, t.tm_day)
    @staticmethod
    def tomorrow():
         t = time.localtime(time.time()+86400)
         return Date(t.tm_year, t.tm_mon, t.tm_day)

# 创建日期的示例
a = Date(1967, 4, 9)
b = Date.now()         # 调用静态方法now()
c = Date.tomorrow()    # 调用静态方法tomorrow()
```

类方法 是将类本身作为对象进行操作的方法。类方法使用`@classmethod`装饰器定义，与实例方法不同，因为根据约定，类是作为第一个参数（名为`cls`）传递的，例如：

```
class Times(object):
    factor = 1
    @classmethod
    def mul(cls,x):
        return cls.factor*x

class TwoTimes(Times):
    factor = 2

x = TwoTimes.mul(4)    # 调用Times.mul(TwoTimes, 4) -> 8
```

在这个例子中，请注意类`TwoTimes`是如何作为对象传递给`mul()`的。尽管这个例子有些深奥，但类方法还有一些实用且巧妙的用法。例如，你定义了一个类，它继承自前面给出的`Date`类并对其略加定制：

```
class EuroDate(Date):
```

```
    # 修改字符串转换，以使用欧洲日期格式
    def __str__(self):
        return "%02d/%02d/%4d" % (self.day, self.month, self.year)
```

由于该类继承自Date，所以它拥有Date的所有特性。但是now()和tomorrow()方法稍微有点不同。例如，如果调用EuroDate.now()，则会返回Date对象，而不是EuroDate对象。类方法可以解决该问题：

```
class Date(object):
    ...
    @classmethod
    def now(cls):
         t = time.localtime()
         # 创建具有合适类型的对象
         return cls(t.tm_year, t.tm_month, t.tm_day)

class EuroDate(Date):
    ...

a = Date.now()        # 调用Date.now(Date)并返回Date
b = EuroDate.now()    # 调用Date.now(EuroDate)并返回EuroDate
```

关于静态方法和类方法需要注意的一点是，Python不会在与实例方法独立的命名空间中管理它们。因此，可以在实例上调用它们。例如：

```
a = Date(1967,4,9)
b = d.now()         # 调用Date.now(Date)
```

这可能很容易引起混淆，因为对d.now()的调用与实例d没有任何关系。这种行为是Python对象系统与其他面向对象语言（如Smalltalk和Ruby）对象系统的区别之一。在这些语言中，类方法与实例方法是严格分开的。

7.7 特性

通常，访问实例或类的属性时，返回的会是所存储的相关值。特性（property）是一种特殊的属性，访问它时会计算它的值。下面是一个简单的例子：

```
class Circle(object):
     def __init__(self,radius):
         self.radius = radius
     # Circle的一些附加特性
     @property
     def area(self):
         return math.pi*self.radius**2
     @property
     def perimeter(self):
         return 2*math.pi*self.radius
```

得到的Circle 对象的行为如下：

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
50.26548245743669
>>> c.perimeter
25.132741228718345
>>> c.area = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

在这个例子中，Circle 实例存储了一个实例变量c.radius 。c.area 和 c.perimeter 是根据该值计算得来的。@property 装饰器支持以简单属性的形式访问后面的方法，无需像平常一样添加额外的()来调用该方法。对象的使用者很难发现正在计算一个属性，除非在试图重新定义该属性时生成了错误消息（如上面的AttributeError 异常所示）。

这种特性使用方式遵循所谓的统一访问原则 。实际上，如果定义一个类，尽可能保持编程接口的统一总是不错的。如果没有特性，将会以简单属性（如c.radius ）的形式访问对象的某些属性，而其他属性将以方法（如c.area() ）的形式访问。费力去了解何时添加额外的()会带来不必要的混淆。特性可以解决该问题。

Python程序员很少认识到，方法本身是被隐式地作为一类特性处理的。考虑下面这个类：

```
class Foo(object):
     def __init__(self,name):
         self.name = name
     def spam(self,x):
         print("%s, %s" % (self.name, x)
```

用户创建f = Foo("Guido") 这样的实例然后访问f.spam 时，不会返回原始函数对象spam ，而是会得到所谓的绑定方法 （bound method），绑定方法是一个对象，代表将在对象上调用() 运算符时执行的方法调用。绑定方法有点类似于已部分计算的函数，其中的self 参数已经填入，但其他参数仍然需要在使用() 调用该函数时提供。这种绑定方法对象是由在后台执行的特性函数静默地创建的。使用@staticmethod 和@classmethod 定义静态方法和类方法时，实际上就指定了使用不同的特性函数，以不同的方式处理对这些方法的访问。例如，@staticmethod 仅“按原样”返回方法函数，不会进行任何特殊的包装或处理。

特性还可以截获操作权，以设置和删除属性。这是通过向特性附加其他setter 和 deleter 方法来实现的，如下所示：

```
class Foo(object):
    def __init__(self,name):
```

```
        self.__name = name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self,value):
        if not isinstance(value,str):
             raise TypeError("Must be a string!")
        self.__name = value
    @name.deleter
    def name(self):
        raise TypeError("Can't delete name")

f = Foo("Guido")
n = f.name           # 调用f.name() – get函数
f.name = "Monty"    # 调用setter name(f,"Monty")
f.name = 45          # 调用setter name(f,45) -> TypeError
del f.name           # 调用deleter name(f) -> TypeError
```

在这个例子中，首先使用@property 装饰器和相关方法将属性name 定义为只读特性。后面的@name.setter 和@name.deleter 装饰器将其他方法与name 属性上的设置和删除操作相关联。这些方法的名称必须与原始特性的名称完全匹配。在这些方法中，请注意实际的名称值存储在属性__name 中。所存储属性的名称无需遵循任何约定，但它必须与特性名称不同，以便将它与特性的名称区分开。

在以前的代码中，通常会看到用property(*getf* =None, *setf* =None, *delf* =None, *doc* =None) 函数来定义特性，往其中传入一组名称不同的方法，用于执行相关操作。例如：

```
class Foo(object):
    def getname(self):
        return self.__name
    def setname(self,value):
        if not isinstance(value,str):
             raise TypeError("Must be a string!")
        self.__name = value
    def delname(self):
        raise TypeError("Can't delete name")
    name = property(getname,setname,delname)
```

这种老方法仍然可以使用，但装饰器版本会让类看起来更整洁。例如，如果使用装饰器，get 、set 和delete 函数将不会显示为方法。

7.8 描述符

使用特性后，对属性的访问将由一系列用户定义的get 、set 和delete 函数控制。这种属性控制方式可以通过描述符对象 进一步泛化。描述符就是一个代表属性值的对象。通过实现一个或多个特殊的__get__() 、__set__() 和__delete__() 方法，可以将描述符与属性访问机制挂钩，还可以自定义这些操作，如下所示：

```
class TypedProperty(object):
    def __init__(self,name,type,default=None):
        self.name = "_" + name
        self.type = type
        self.default = default if default else type()
    def __get__(self,instance,cls):
        return getattr(instance,self.name,self.default)
    def __set__(self,instance,value):
        if not isinstance(value,self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance,self.name,value)
    def __delete__(self,instance):
        raise AttributeError("Can't delete attribute")

class Foo(object):
    name = TypedProperty("name",str)
    num  = TypedProperty("num",int,42)
```

在这个例子中，类TypedProperty 定义了一个描述符，分配属性时它将进行类型检查，如果尝试删除属性，它将引发错误。例如：

```
f = Foo()
a = f.name          # 隐式调用Foo.name.__get__(f,Foo)
f.name = "Guido"    # 调用Foo.name.__set__(f,"Guido")
del f.name          # 调用Foo.name.__delete__(f)
```

描述符只能在类级别上进行实例化。不能通过在__init__() 和其他方法中创建描述符对象来为每个实例创建描述符。而且，持有描述符的类使用的属性名称比实例上存储的属性名称具有更高的优先级。在上一个例子中，描述符对象接受参数name ，并且对其略加修改（前面加了个下划线），原因就在于此。为了能让描述符在实例上存储值，描述符必须挑选一个与它本身所用名称不同的名称。

7.9 数据封装和私有属性

默认情况下，类的所有属性和方法都是“公共的”。这意味着对它们的访问没有任何限制。这还暗示着，在基类中定义的所有内容都会被派生类继承，并可从派生类内进行访问。在面向对象的应用程序中，通常我们不希望发生这种行为，因为它会暴露对象的内部实现，可能导致在派生类中定义的对象与在基类中定义的对象之间发生命名空间冲突。

为了解决该问题，类中所有以双下划线开头的名称（如__Foo ）都会自动变形，形成具有 *_Classname__* Foo 形式的新名称。这提供了一种在类中添加私有属性和方法的有效方式，因为派生类中使用的私有名称不会与基类中使用的相同私有名称发生冲突，如下所示：

```
class A(object):
    def __init__(self):
        self.__X = 3       # 变形为self._A__X
    def __spam(self):      # 变形为_A__spam()
        pass
    def bar(self):
```

```
        self.__spam()      # 只调用A.__spam()
class B(A):
    def __init__(self):
        A.__init__(self)
        self.__X = 37      # 变形为self._B__X
    def __spam(self):      # 变形为_B__spam()
        pass
```

尽管这种方案似乎隐藏了数据，但并没有严格的机制来实际阻止对类的“私有”属性进行访问。特别是如果已知类名称和相应私有属性的名称，则可以使用变形后的名称来访问它们。通过重定义`__dir__()`方法，类可以降低这些属性的可见性，`__dir__()`方法提供了检查对象的`dir()`函数所返回的名称列表。

尽管这种名称变形似乎是一个额外的处理步骤，但变形过程实际上只在定义类时发生一次。它不会在方法执行期间发生，也不会为程序的执行添加额外的开销。而且要知道，名称变形不会在`getattr()`、`hasattr()`、`setattr()`或`delattr()`等函数中发生，在这些函数中，属性名被指定为字符串。对于这些函数，需要显式使用变形名称（如`__Classname__name`）来访问属性。

建议在定义可变属性时，通过特性来使用私有属性。这样，就可鼓励用户使用特性名称，而无需直接访问底层实例数据（如果你在实例开头添加了一个特性，可能不想采用这种访问方式）。上一节中已经给出了这样的例子。

通过为方法提供私有名称，超类可以阻止派生类重新定义和更改方法的实现。例如，示例中的`A.bar()`方法只调用`A.__spam()`，无论`self`具有何种类型，或者派生类中是否存在不同的`__spam()`方法都是如此。

最后，不要混淆私有类属性的命名和模块中“私有”定义的命名。一个常见的错误是，在定义类时，在属性名上使用单个前导下划线来隐藏属性值（如`_name`）。在模块中，这种命名约定可以阻止通过`from module import *`语句导出名称。但是在类中，这种命名约定既不能隐藏属性，在某个类继承该类并使用相同名称定义一个新属性或方法时，也不能阻止出现名称冲突。

7.10 对象内存管理

定义类后，得到的类是一个可创建新实例的工厂。例如：

```
class Circle(object):
    def __init__(self,radius):
         self.radius = radius

# 创建一些Circle实例
c = Circle(4.0)
d = Circle(5.0)
```

实例的创建包括两个步骤：使用特殊方法`__new__()`创建新实例，然后使用`__init__()`初始化该实例。例如，操作`c = Circle(4.0)`执行以下步骤：

```
c = Circle.__new__(Circle, 4.0)
if isinstance(c,Circle):
    Circle.__init__(c,4.0)
```

类的__new__()方法很少通过用户代码定义。如果定义了它，它通常是用原型__new__(cls, *args, **kwargs)编写的，其中args和kwargs与传递给__init__()的参数相同。__new__()始终是一个类方法，接受类对象作为第一个参数。尽管__new__()会创建一个实例，但它不会自动调用__init__()。

如果看到在类中定义了__new__()，通常表明这个类会做两件事之一。首先，该类可能继承自一个基类，该基类的实例是不可变的。如果定义的对象继承自不可变的内置类型（如整数、字符串或元组），常常会遇到这种情况，因为__new__()是唯一在创建实例之前执行的方法，也是唯一可以修改值的地方（也可以在__init__()中修改，但这时修改可能为时已晚）。例如：

```
class Upperstr(str):
    def __new__(cls,value=""):
        return str.__new__(cls, value.upper())

u = Upperstr("hello")   # 值为"HELLO"
```

__new__()的另一个主要用途是在定义元类时使用。元类将在本章结尾部分介绍。

创建实例之后，实例将由引用计数来管理。如果引用计数到达0，实例将立即被销毁。当实例即将被销毁时，解释器首先会查找与对象相关联的__del__()方法并调用它。而实际上，很少有必要为类定义__del__()方法。唯一的例外是在销毁对象之后需要执行清除操作（如关闭文件、关闭网络连接或释放其他系统资源）。即使在这种情况下，依靠__del__()来完全关闭实例也存在一定的危险，因为无法保证在解释器退出时会调用该方法。更好的方案是定义一个方法，如close()，程序可以使用该方法显式执行关闭操作。

有时，程序会使用del语句来删除对象引用。如果这导致对象的引用计数变成0，则会调用__del__()方法。但是，del语句通常不会直接调用__del__()。

销毁对象存在一个不易察觉的风险，即定义了__del__()的实例无法被Python的循环垃圾回收器回收（这是只在需要时才定义__del__的重要原因）。使用过没有自动垃圾回收功能的语言（如C++）的程序员应该注意编程风格，不要定义不必要的__del__()。尽管定义__del__()很少会破坏垃圾回收器，但是在某些编程模式中（特别是涉及父子关系或图表的编程模式），这可能会引起问题。例如，设想某个对象实现了一种“观察者模式”（Observer Pattern）。

```
class Account(object):
    def __init__(self,name,balance):
        self.name = name
        self.balance = balance
        self.observers = set()
    def __del__(self):
        for ob in self.observers:
```

```
            ob.close()
        del self.observers
    def register(self,observer):
        self.observers.add(observer)
    def unregister(self,observer):
        self.observers.remove(observer)
    def notify(self):
        for ob in self.observers:
            ob.update()
    def withdraw(self,amt):
        self.balance -= amt
        self.notify()

class AccountObserver(object):
     def __init__(self, theaccount):
         self.theaccount = theaccount
         theaccount.register(self)
     def __del__(self):
         self.theaccount.unregister(self)
         del self.theaccount
     def update(self):
         print("Balance is %0.2f" % self.theaccount.balance)
     def close(self):
         print("Account no longer in use")

# 示例设置
a = Account('Dave',1000.00)
a_ob = AccountObserver(a)
```

在这段代码中，Account 类允许一组AccountObserver 对象监控Account 实例，在余额出现变化时接收更新。为此，每个Account 都会保留一组观察者，每个AccountObserver 会保留对账户的引用。每个类都定义了__del__()，以尝试进行某种清除操作（如注销等）。但是，这种尝试不会生效。相反，类会创建一个引用循环，在这个循环中，引用计数永远不会到达0，也永远不会执行清除操作。不仅如此，垃圾回收器（gc 模块）甚至不会清除该类，这会导致永久性的内存泄漏。

解决本示例中这种问题的一种方式是，使用weakref 模块为一个类创建对其他类的弱引用。弱引用 是一种在不增加对象引用计数的情况下创建对象引用的方式。要使用弱引用，需要添加一点额外的功能代码，来检查被引用的对象是否仍然存在。下面是经过修改的观察者类示例：

```
import weakref
class AccountObserver(object):
     def __init__(self, theaccount):
         self.accountref = weakref.ref(theaccount) # 创建weakref
         theaccount.register(self)
     def __del__(self):
         acc = self.accountref()         # 获取账户
         if acc:                         # 如果仍然存在则注销
              acc.unregister(self)
     def update(self):
         print("Balance is %0.2f" % self.accountref().balance)
     def close(self):
         print("Account no longer in use")
```

```
# 示例设置
a = Account('Dave',1000.00)
a_ob = AccountObserver(a)
```

在这个例子中我们创建了弱引用accountref 。要访问底层的Account ，可以像函数一样调用它。这可能返回Account ， 也可能在实例不存在时返回None 。这样修改之后就没有引用循环了。如果销毁了Account 对象，它的__del__ 方法将运行，观察者会收到通知。gc 模块也会正常工作。关于weakref 模块的更多信息参见第13章。

7.11 对象表示和属性绑定

从内部实现上看，实例是使用字典来实现的，可以通过实例的__dict__ 属性访问该字典。这个字典包含的数据对每个实例而言都是唯一的，如下所示：

```
>>> a = Account('Guido', 1100.0)
>>> a.__dict__
{'balance': 1100.0, 'name': 'Guido'}
```

可以在任何时候向实例添加新属性，例如：

```
a.number = 123456    # 将属性'number'添加到a.__dict__
```

对实例的修改始终会反映到局部__dict__ 属性中。同样，如果直接对__dict__ 进行修改，所做的修改也会反映在实例的属性中。

实例通过特殊属性__class__ 链接回它们的类。类本身也只是对字典的浅层包装，你可以在实例的__dict__ 属性中找到这个字典。可以在类字典中找到各种方法。例如：

```
>>> a.__class__
<class '__main__.Account'>
>>> Account.__dict__.keys()
['__dict__', '__module__', 'inquiry', 'deposit', 'withdraw',
'__del__', 'num_accounts', '__weakref__', '__doc__', '__init__']
>>>
```

最后，通过特殊属性__bases__ 中将类链接到它们的基类，该属性是一个基类元组。这种底层结构是获取、设置和删除对象属性的所有操作的基础。

只要使用obj.*name* = *value* 设置了属性，特殊方法obj.__setattr__("*name* ", *value*) 就会被调用。如果使用del obj.name 删除了一个属性，就会调用特殊方法obj.__delattr__("*name* ") 。这些方法的默认行为是修改或删除obj 的局部__dict__ 的值，除非请求的属性正好是一个特性或描述符。在这种情况下，设置和删除操作将由与该特性相关联的设置和删除函数执行。

在查找属性（如obj.*name* ）时，将调用特殊方法obj.__getattrribute__("*name*

")。该方法执行搜索来查找该属性，这通常涉及检查特性、查找局部__dict__属性、检查类字典以及搜索基类。如果搜索过程失败，最终会尝试调用类的__getattr__()方法（如果已定义）来查找该属性。如果这也失败，就会抛出AttributeError异常。

如果有必要，用户定义的类可以实现其自己的属性访问函数。例如：

```
class Circle(object):
    def __init__(self,radius):
        self.radius = radius
    def __getattr__(self,name):
        if name == 'area':
             return math.pi*self.radius**2
        elif name == 'perimeter':
             return 2*math.pi*self.radius
    else:
         return object.__getattr__(self,name)
def __setattr__(self,name,value):
    if name in ['area','perimeter']:
         raise TypeError("%s is readonly" % name)
    object.__setattr__(self,name,value)
```

重新实现这些方法的类应该可以依靠object中的默认实现来执行实际的工作。这是因为默认实现能够处理类的更高级特性，如描述符和特性。

一般来讲，类很少重新定义属性访问运算符。但是，在编写通用的包装器和现有对象的代理时，通常会使用属性访问运算符。通过重新定义__getattr__()、__setattr__()和__delattr__()，代理可以捕获属性访问操作，并透明地将这些操作转发给另一个对象。

7.12 __slots__

通过定义特殊变量__slots__，类可以限制对合法实例属性名称的设置，如下所示：

```
class Account(object):
    __slots__ = ('name','balance')
      ...
```

定义__slots__时，可以将实例上分配的属性名称限制为指定的名称，否则将引发AttributeError异常。这种限制可以阻止其他人向现有实例添加新属性，即便用户将属性名称写错，也不会创建出新的属性来。

在实际使用中，__slots__从未被当作一种安全的特性来实现。它实际上是对内存和执行速度的一种性能优化。使用__slots__的类的实例不再使用字典来存储实例数据，转而采用一种基于数组的更加紧凑的数据结构。在会创建大量对象的程序中，使用__slots__可以显著减少减少内存占用和执行时间。

注意，__slots__与继承的配合使用需要一定的技巧。如果类继承自使

用__slots__的基类，那么它也需要定义__slots__来存储自己的属性（即使它不会添加任何属性也是如此），这样才能利用__slots__提供的优势。如果忘记了这一点，派生类的运行速度将更慢，占用的内存也更多，比完全不使用__slots__时情况更糟。

__slots__的使用还会破坏期望实例具有底层__dict__属性的代码。尽管这一点通常不适用于用户代码，但对于支持对象的实用库和其他工具，其代码可能要依靠__dict__来调试、序列化对象以及执行其他操作。

最后，如果类中重新定义了__getattribute__()、__getattr__()和__setattr__()等方法，__slots__的存在不会对它们的调用产生任何影响。但是，这些方法的默认行为将考虑到__slots__。此外应该强调一点，没有必要向__slots__添加方法或特性名称，因为它们存储在类中，而不是存储在每个实例中。

7.13 运算符重载

通过向类中添加第3章中介绍的特殊方法的实现，可以让用户定义的对象使用Python的所有内置运算符。例如，如果希望向Python添加一种新的数字类型，可以定义一个类并在该类中定义__add__()等特殊方法，让实例能够使用标准数学运算符。

下面的例子演示了这一过程，其中定义了一个类来实现能够使用一些标准运算符的复数。

> **注意**
>
> 由于Python已经提供了复数类型，所以这个类只是用于演示目的。

```
class Complex(object):
    def __init__(self,real,imag=0):
        self.real = float(real)
        self.imag = float(imag)
    def __repr__(self):
        return "Complex(%s,%s)" % (self.real, self.imag)
    def __str__(self):
        return "(%g+%gj)" % (self.real, self.imag)
    # self + other
    def __add__(self,other):
        return Complex(self.real + other.real, self.imag + other.imag)
    # self - other
    def __sub__(self,other):
        return Complex(self.real - other.real, self.imag - other.imag)
```

在这个例子中，__repr__()方法创建一个字符串，可以通过对该字符串进行求值来重新创建对象（也就是Complex(real,imag)）。应该尽可能在所有用户定义对象中遵循这一约定。另一方面，__str__()方法创建具有良好输出格式的字符串（这是将由print语句生成的字符串）。

其他运算符（如__add__()和__sub__()）实现数学运算。对于这些运算符，需要注意的地方是操作数的顺序和类型强制。从在上一个例子中实现的运算符可以看出，__add__()和__sub__()运算符仅 适用于复数出现在运算符左侧的情形。如果复数

出现在运算符右侧，而且最左侧的操作数不是Complex，这些运算符将无效。例如：

```
>>> c = Complex(2,3)
>>> c + 4.0
Complex(6.0,3.0)
>>> 4.0 + c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'Complex'

>>>
```

操作c + 4.0偶尔也会生效。Python的所有内置数字都已拥有.real和.imag属性，所以在计算中使用了它们。如果other对象没有这些属性，实现将无效。如果希望Complex的实现能够与缺少这些属性的对象一同工作，必须添加额外的转换代码来提取所需的信息（具体信息取决于其他对象的类型）。

操作4.0 + c则完全无效，因为内置的浮点类型不知道Complex类的任何信息。要解决此问题，可以向Complex添加逆向操作数（reversed-operand）方法：

```
class Complex(object):
    ...
    def __radd__(self,other):
        return Complex(other.real + self.real, other.imag + self.imag)
    def __rsub__(self,other):
        return Complex(other.real - self.real, other.imag - self.img)
    ...
```

这些方法是备用方法。如果操作4.0 + c失败，Python将在发出TypeError之前首先尝试执行c.__radd__(4.0)。

早期的Python版本会尝试用各种方法来强制转换混合类型操作中的类型。例如，你可能会遇到实现了__coerce__()方法的老式Python类。Python 2.6或Python 3不再使用该方法了。另外，也不要被__int__()、__float__()或__complex__()等特殊方法所欺骗。尽管这些方法是通过显式转换（如int(*x*)或float(*x*)）来调用的，但在混合类型计算中绝不会隐式调用它们来执行类型转换。所以，如果编写的类中的运算符必须处理混合类型，则必须在每个运算符的实现中显式处理类型转换。

7.14 类型和类成员测试

创建类的实例时，该实例的类型为类本身。要测试实例是否是类中的成员，可以使用内置函数isinstance(*obj* ,*cname*)。如果对象 *obj* 属于类 *cname* 或派生自 *cname* 的任何类，该函数将返回True，如下所示：

```
class A(object): pass
class B(A): pass
class C(object): pass

a = A()          # 'A'的实例
```

```
b = B()           # 'B'的实例
c = C()           # 'C'的实例

type(a)            # 返回类对象A
isinstance(a,A)    # 返回True
isinstance(b,A)    # 返回True，B派生自A
isinstance(b,C)    # 返回False，C不是派生自A
```

同样，如果类A 是类B 的子类，内置函数issubclass(A,B) 将返回True ，如下所示：

```
issubclass(B,A)    # 返回True
issubclass(C,A)    # 返回False
```

检查对象的类型时，有一个问题是，程序员经常绕过继承，创建的对象只是模仿另一个对象的行为。例如，考虑下面这两个类：

```
class Foo(object):
    def spam(self,a,b):
        pass

class FooProxy(object):
    def __init__(self,f):
        self.f = f
    def spam(self,a,b):
        return self.f.spam(a,b)
```

在这个例子中，FooProxy 的功能与Foo 相同。它实现了同样的方法，甚至悄悄使用了Foo 。但是，在类型系统中，FooProxy 不同于Foo 。例如：

```
f = Foo()           # 创建Foo
g = FooProxy(f)      # 创建FooProxy
isinstance(g, Foo)   # 返回False
```

如果编写的程序使用了isinstance() 显式检查Foo ，那么该程序一定无法正确检查FooProxy 对象。但是，我们通常并不需要这么严格的限制。相反，因为它具有相同的接口，断言FooProxy 对象可以作为Foo 使用或许更合适。为此，可以定义一个对象，在其中重新定义isinstance() 和issubclass() 的行为，目的是分组对象并对其进行类型检查，如下所示：

```
class IClass(object):
    def __init__(self):
         self.implementors = set()
    def register(self,C):
         self.implementors.add(C)
    def __instancecheck__(self,x):
         return self.__subclasscheck__(type(x))
    def __subclasscheck__(self,sub):
         return any(c in self.implementors for c in sub.mro())
```

```
# 现在使用上面的对象
IFoo = IClass()
IFoo.register(Foo)
IFoo.register(FooProxy)
```

在这个例子中，IClass 类创建了一个对象，该对象仅将一组其他类分组到一个集合中。register() 方法向该集合中添加新类。只要执行isinstance(*x* , IClass) 操作，就会调用__instancecheck__() 这个特殊方法。只要调用issubclass(*C* ,IClass) 操作，就会调用__subclasscheck__() 这个特殊方法。

通过使用IFoo 对象和注册的实现器，现在可以用以下方式执行类型检查：

```
f = Foo()           # 创建Foo
g = FooProxy(f)      # 创建FooProxy
isinstance(f, IFoo)         # 返回True
isinstance(g, IFoo)         # 返回True
issubclass(FooProxy, IFoo) # 返回True
```

需要强调的一点是，这个例子中不会发生强类型检查。IFoo 对象已经重载了实例检查操作，允许断言某个类属于某个组。它不会断言与实际的编程接口相关的任何信息，也不会实际执行其他任何验证操作。实际上，你可以注册任何希望分组到一起的对象的集合，无需考虑这些类如何彼此关联。通常，对类的分组基于某种标准，如将实现相同编程接口的所有类分组到一起。但是，重载__instance-check__() 或__subclasscheck__() 时，不应推断出这一含义。实际的含义应由应用程序决定。

Python提供了一种更加正式的机制来分组对象、定义接口并进行类型检查。这是通过定义抽象基类（将在下一节中介绍）来实现的。

7.15 抽象基类

上一节介绍了isinstance() 和issubclass() 操作可以重载。这可以用于创建将类似的类分组到一起的对象，以及执行各种形式的类型检查。抽象基类 以这一概念为基础，提供了一种方式，用以组织对象的层次结构，做出关于所需方法的断言，以及实现其他一些功能。

要定义抽象基类，需要使用abc 模块。该模块定义了一个元类（ABCMeta ）和一组装饰器（@abstractmethod 和@abstractproperty ），用法如下：

```
from abc import ABCMeta, abstractmethod, abstractproperty
class Foo:                     # 在Python 3中，使用下面的语法
    __metaclass__ = ABCMeta  # class Foo(metaclass=ABCMeta)
    @abstractmethod
    def spam(self,a,b):
        pass
    @abstractproperty
    def name(self):
        pass
```

要定义抽象类，需要将其元类按上例所示设置为ABCMeta （还要注意Python 2与Python 3之间的语法区别）。这一步是必需的，因为抽象类的实现离不开元类（将在下一节介绍）。在抽象类中，@abstractmethod 和@abstractproperty 装饰器指定方法或特性必须由Foo 的子类实现。

抽象类并不能直接实例化。如果尝试为上面的类创建Foo ，将得到以下错误：

```
>>> f = Foo()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Foo with abstract methods spam
>>>
```

这一限制也适用于派生类。例如，如果类Bar 继承自Foo ，但它没有实现一个或多个抽象方法，那么尝试创建Bar 将会失败，并生成类似的错误。由于添加了这一检查过程，需要对必须在子类上实现的方法和特性进行断言的程序员而言，抽象类很有用。

尽管抽象类会在必须实现的方法和特性上强制实施规则，但它不会对参数执行一致性检查或返回值。所以，抽象类不会检查某个子类，查看某个方法是否使用了与抽象方法相同的参数。同样，需要定义特性的抽象类也不会检查某个子类中的特性是否支持在基类中指定的特性的操作集（get 、set 和delete ）。

尽管抽象类无法实例化，但它可以定义要在子类中使用的方法和特性。而且，基类中的抽象方法仍然可以从子类中调用。例如，可以从子类调用Foo.spam(a,b) 。

抽象基类支持对已经存在的类进行注册，使其属于该基类。这是用register() 方法完成的，如下所示：

```
class Grok(object):
    def spam(self,a,b):
        print("Grok.spam")

Foo.register(Grok)       # 向Foo抽象基类注册
```

向抽象基类注册某个类时，对于注册类中的实例，涉及抽象基类的类型检查操作（如isinstance() 和issubclass() ）将返回True 。向抽象类注册某个类时，Python不会检查该类是否实际实现了任何抽象方法或特性。这种注册过程只会影响类型检查。它不会对已注册的类进行额外的错误检查。

与很多其他面向对象的语言不同，Python将内置类型组织到一个相对扁平的层次结构中。例如，如果查看内置类型，如int 或float ，可以看到它们直接继承自所有对象的根，即object ，而不是表示数字的中间基类。因此很难编写根据通用类别（如仅是一个数字的实例）检查和操作对象的程序。

抽象类机制解决了这一问题，它允许将已存在的对象组织到用户可定义的类型层次结构中。而且，一些库模块会根据功能来组织内置类型。collections 模块包含与序列、

集合和字典有关的各种操作的抽象基类。numbers 模块包含与组织数字层次结构相关的抽象基类。更多细节可以在第14章和第15章找到。

7.16 元类

在Python中定义类时，类定义本身将成为一个对象，如下所示：

```
class Foo(object): pass
isinstance(Foo,object)     # 返回True
```

仔细想想，你就会认识到必须存在某个东西去创建Foo 对象。类对象的这种创建方式是由一种名为元类 的特殊对象控制的。简言之，元类就是知道如何创建和管理类的对象。

在上面的例子中，控制Foo 创建的元类是一个名为type 的类。实际上，如果查看Foo 的类型，将会发现它的类型为type ：

```
>>> type(Foo)
<type 'type'>
```

使用class 语句定义新类时，将会发生很多事情。首先，类主体将作为其自己的私有字典内的一系列语句来执行。语句的执行与正常代码中的执行过程相同，只是增加了会在私有成员（名称以__开头）上发生的名称变形。最后，类的名称、基类列表和字典将传递给元类的构造函数，以创建相应的类对象。下面的例子演示了这一过程：

```
class_name = "Foo"              # 类名
class_parents = (object,)       # 基类
class_body = """                # 类主体
def __init__(self,x):
    self.x = x
def blah(self):
    print("Hello World")
"""
class_dict = { }
# 在局部字典class_dict中执行类主体
exec(class_body,globals(),class_dict)

# 创建类对象Foo
Foo = type(class_name,class_parents,class_dict)
```

类创建的最后一步——调用元类type() 的步骤——可以自定义。可以通过多种方式控制类定义的最后一步。首先，类可以显式地指定其元类，这通过设置__metaclass__ 类变量（Python 2）或在基类元组中提供metaclass 关键字参数（Python 3）来实现的。

```
class Foo:                      # 在Python 3中，使用下面的语法
    __metaclass__ = type       # class Foo(metaclass=type)
    ...
```

如果没有显式指定元类，class 语句将检查基类元组（如果存在）中的第一个条目。在这种情况下，元类与第一个基类的类型相同。所以，在编写以下内容时，Foo 的类型将与object 相同。

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

如果没有指定基类，class 语句将检查全局变量__metaclass__ 是否存在。如果找到了该变量，将使用它来创建类。如果设置了该变量，在使用简单的类语句时，它将控制类的创建方式，如下所示：

```
__metaclass__ = type
class Foo:
     pass
```

最后，如果没有找到__metaclass__ 值，Python将使用默认的元类。在Python 2中，默认的元类是types.ClassType ，这是所谓的旧式类 。在Python中，这种类（从Python 2.2开始已不提倡使用）相当于类的原始实现。尽管这些类仍然可以使用，但在新代码中应该避免，这里对此不作进一步介绍。在Python 3中，默认的元类就是type() 。

如果希望在框架中更强有力地控制用户自定义对象的定义，就可以在这种框架中使用元类，这就是元类的主要用途。定义自定义元类时，它通常会继承自type() ，并重新实现__init__() 或__new__() 等方法。下面给出了一个元类的例子，它要求所有方法必须拥有一个文档字符串：

```
class DocMeta(type):
    def __init__(self,name,bases,dict):
        for key, value in dict.items():
            # 跳过特殊方法和私有方法
            if key.startswith("__"): continue
            # 跳过不可调用的任何方法
            if not hasattr(value,"__call__"): continue
            # 检查doc字符串
            if not getattr(value,"__doc__"):
                raise TypeError("%s must have a docstring" % key)
        type.__init__(self,name,bases,dict)
```

在该元类中，__init__() 方法会检查类字典的内容。该方法对字典进行扫描，查找方法并检查是否所有方法都拥有文档字符串。如果没有，则生成一个TypeError 异常。否则将调用type.__init__() 的默认实现来初始化该类。

如果要使用该元类，类需要明确选择它。最常用的实现技巧是首先定义一个基类，如下所示：

```
class Documented:            # 在Python 3中，使用下面的语法
    __metaclass__ = DocMeta   # class Documented(metaclass=DocMeta)
```

然后将该基类用作所有需要添加文档的对象的父类。例如：

```
class Foo(Documented):
    spam(self,a,b):
        "spam does something"
        pass
```

这个例子演示了元类的一个主要用途，那就是检查和收集关于类定义的信息。元类不会更改实际创建的类的任何内容，只是添加一些额外的检查。

在更高级的元类应用程序中，元类可以在创建类之前同时检查和更改类定义的内容。如果要进行更改，应该重新定义在创建类本身之前运行的__new__() 方法。这个技巧通常与使用描述符或特性来包装属性结合使用，因为这样可以捕获在类中使用的名称。例如，下面给出了在7.8节中使用TypedProperty 描述符的修改版本：

```
class TypedProperty(object):
    def __init__(self,type,default=None):
        self.name = None
        self.type = type
        if default: self.default = default
        else:       self.default = type()
    def __get__(self,instance,cls):
        return getattr(instance,self.name,self.default)
    def __set__(self,instance,value):
        if not isinstance(value,self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance,self.name,value)
    def __delete__(self,instance):
        raise AttributeError("Can't delete attribute")
```

在这个例子中，描述符的name 属性被设置为None 。为了填补这一属性，我们将使用元类。例如：

```
class TypedMeta(type):
    def __new__(cls,name,bases,dict):
        slots = []
        for key,value in dict.items():
            if isinstance(value,TypedProperty):
                value.name = "_" + key
                slots.append(value.name)
        dict['__slots__'] = slots
        return type.__new__(cls,name,bases,dict)

# 要使用的用户定义对象的基类
class Typed:                    # 在Python 3中，使用下面的语法
    __metaclass__ = TypedMeta    # class Typed(metaclass=TypedMeta)
```

在这个例子中，元类扫描类字典并查找TypedProperty 的实例。如果找到，它设置name 属性并在slots 中建立名称列表。完成之后，__slots__ 属性将添加到类字典中，并通过调用type() 元类的__new__() 方法来构造该类。下面给出了使用这个新元类

的例子：

```
class Foo(Typed):
    name = TypedProperty(str)
    num  = TypedProperty(int,42)
```

尽管使用元类可以显著改变用户定义的类的行为和语义，但不应该使类的工作方式与标准Python文档中的描述相差过多。如果编写的类不符合标准的类编码规则，用户将会对代码感到困惑。

7.17 类装饰器

上一节展示了如何通过定义元类来自定义类。但是，有时所需做的只是在定义类之后执行一些额外处理，例如将类添加到注册表或数据库。这类问题的替代解决办法是使用类装饰器。类装饰器 是一种函数，它接受类作为输入并返回类作为输出。例如：

```
registry = { }
def register(cls):
     registry[cls.__clsid__] = cls
     return cls
```

在这个例子中，注册函数在类中查找__clsid__ 属性。如果找到，则使用该属性将该类添加到字典中，将类标识符映射到类对象。要使用该函数，可以在类定义前将它用作装饰器。例如：

```
@register
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
       pass
```

此处使用装饰器语法带来了极大的便利。使用另一种方式同样可以实现这个目的：

```
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
      pass
register(Foo)      # 注册类
```

尽管可以在类装饰器函数中对类做很多邪恶的事情，但最好避免过多的魔法，如为类添加一个包装器或者重写类的内容。

第8章　模块、包与分发

大型Python程序以模块和包的形式组织。另外，Python标准库中包含大量模块。本章详细介绍模块和包系统。还将提供有关如何安装第三方模块和分发源代码的信息。

8.1　模块与import 语句

任何Python源文件都能以模块的形式使用。例如，考虑以下代码：

```
# spam.py
a = 37
def foo():
    print("I'm foo and a is %s" % a)
def bar():
    print("I'm bar and I'm calling foo")
    foo()
class Spam(object):
    def grok(self):
        print("I'm Spam.grok")
```

要以模块的形式加载这段代码，可以使用import spam 语句。首次使用import 加载模块时，它将做3件事。

(1) 创建新的命名空间，用作在相应源文件中定义的所有对象的容器。在模块中定义的函数和方法在使用global 语句时将访问该命名空间。

(2) 在新创建的命名空间中执行模块中包含的代码。

(3) 在调用函数中创建名称来引用模块命名空间。这个名称与模块的名称相匹配，按如下方式使用：

```
import spam             # 加载并执行模块spam
x = spam.a              # 访问模块spam的一个成员
spam.foo()              # 调用模块spam中的一个函数
s = spam.Spam()         # 创建spam.Spam()的一个实例
s.grok()
...
```

需要强调的是，import 执行已加载的源文件中的所有语句。如果模块执行计算或生成了除定义变量、函数和类以外的输出，就可以看到生成的这些结果。另外，使用模块时一个常见的疑难点就是对类的访问。请记住，如果文件spam.py 定义了类Spam，必须使用名称spam.Spam 来引用该类。

要导入多个模块，可以为import 提供逗号分隔的模块名称列表，例如：

```
import socket, os, re
```

用于引用模块的名称可以使用as 限定符进行更改，例如：

```
import spam as sp
import socket as net
sp.foo()
sp.bar()
net.gethostname()
```

像这样使用其他的名称加载模块时，新名称仅应用于出现了import 语句的源文件或上下文。其他程序模块仍然可以使用模块原来的名称加载它。

更改已导入模块的名称对于编写可扩展的代码很有用。例如，设想有两个模块xmlreader.py 和csvreader.py，它们都定义了函数read_data(*filename*)，作用是从文件中读取一些数据，但采用不同的输入格式。可以编写代码来选择性地挑选读取模块，例如：

```
if format == 'xml':
    import xmlreader as reader
elif format == 'csv':
    import csvreader as reader
data = reader.read_data(filename)
```

模块是Python中的第一类对象。这意味着它们可以被分配给变量，放置在列表等数据结构中，以及以数据的形式在程序中传递。例如，上一个例子中的reader 变量只用于引用相应的模块对象。在后台，模块对象是位于字典之上的一层，这个字典用于持有模块命名空间的内容。这个字典以模块的__dict__ 形式使用，并且只要查找或更改模块中的值，就会对该字典进行相应操作。

import 语句可以出现在程序中的任何位置。但是，每个模块中的代码仅加载和执行一次，无论import 语句被使用了多少次。后续的import 语句仅将模块名称绑定到前一次导入所创建的模块对象。你可以在变量sys.modules 中找到一个字典，其中包含了所有当前加载的模块。该字典将模块名称映射到模块对象。该字典的内容用于确定import 是否加载模块的全新拷贝。

8.2 从模块导入选定符号

from 语句用于将模块中的具体定义加载到当前命名空间中。from 语句相当于import，但它不会创建一个名称来引用新创建的模块命名空间，而是将对模块中定义的一个或多个对象的引用放到当前命名空间中：

```
from spam import foo   # 导入spam并将foo放在当前命名空间中
foo()                   # 调用spam.foo()
spam.foo()              # NameError: spam
```

from 语句还接受用逗号分隔的对象名称列表。例如：

```
from spam import foo, bar
```

如果要导入一个极长的名称列表，可以将名称放在括号中。这样可以轻松地将import 语句放在多行上，例如：

```
from spam import (foo,
                  bar,
                  Spam)
```

另外，as 限定符可用于重命名使用from 导入的具体对象，例如：

```
from spam import Spam as Sp
s = Sp()
```

星号（* ）通配符也可用于加载模块中的所有定义，但以下划线开头的定义除外，例如：

```
from spam import *   # 将所有定义加载到当前命名空间中
```

from *module* import * 语句只能在模块最顶层使用。具体来讲，在函数体内使用这种导入形式是不合法的，原因在于这种导入语句与函数作用域规则之间具有独特的交互方式（例如，将函数编译为内部字节码时，函数中使用的所有符号都需要明确指定）。

通过定义列表__all__ ，模块可以精确控制from module import * 导入的名称集合，例如：

```
# 模块：spam.py
__all__ = [ 'bar', 'Spam' ]  # 将使用from spam import *导出的名称
```

使用from 导入形式导入定义不会更改定义的作用域规则。例如，考虑以下代码：

```
from spam import foo
a = 42
foo()     # 打印"I'm foo and a is 37"
```

在这个例子中，spam.py 中foo() 的定义引用了全局变量a 。在不同命名空间中引用foo 时，并不会更改该函数中的变量绑定规则。因此，函数的全局命名空间始终是定义该函数的模块，而不是将函数导入并调用该函数的命名空间。这也适用于函数调用。例如，在以下代码中，对bar() 的调用会导致调用spam.foo() ，而不是上一个代码示例中重新定义的foo() ：

```
from spam import bar
def foo():
    print("I'm a different foo")
bar()       # 当bar调用foo()时，它将调用spam.foo()，而不是
            # 上面的foo()定义
```

对于from 导入形式，另一个容易混淆的地方是全局变量的行为。例如，考虑以下代码：

```
from spam import a, foo  # 导入全局变量
a = 42                    # 修改该变量
foo()                     # 打印"I'm foo and a is 37"
print(a)                  # 打印"42"
```

这里需要理解的一点是，Python中的变量赋值不是一种存储操作。也就是说，上例中对a 的赋值不会将新值存储在a 中并覆盖以前的值。而是将创建包含值42的新对象，并用名称a 来引用它。此时，a 不再绑定到导入模块中的值，而是绑定到其他对象。因此，使用from 语句没办法让导入的变量模仿全局变量和C或Fortran等语言中通用块的行为。如果需要在程序中使用可变的全局程序参数，可以将这些参数放在模块中并通过import 语句显式使用模块名称（也就是显式使用spam.a ）。

8.3 以主程序的形式执行

Python源文件可以通过两种方式执行。import 语句在自己的命名空间中以库模块的形式执行代码。但是，代码也可以主程序或脚本的形式执行。将程序作为脚本名称提供给解释器时，就是这样执行的：

```
% python spam.py
```

每个模块会定义一个包含模块名称的变量__name__ 。程序可以检查该变量，以确定它们在哪个模块中执行。解释器的顶层模块名为__main__ 。在命令行中指定或交互式输入的程序将在__main__ 模块中运行。有时，程序可能改变其行为，这取决于程序是以模块的形式导入还是在__main__ 中运行。例如，模块可能包含一些测试代码，如果模块以主程序的形式执行，将执行这些测试代码，如果模块只是由另一个模块导入，则不会执行测试代码。可以通过以下方式实现这一功能：

```
# 检查模块是否以程序的形式运行
if __name__ == '__main__':
     # 是
     statements


else:
     # 否，我必须以模块的形式导入
     statements
```

对于计划用作库的源文件， 使用该技巧来包含可选的测试或示例代码是一种常见的做法。例如，如果你正在开发一个模块，可以将测试库功能的代码放在如上所示的`if` 语句中，并以主程序的形式运行模块。用户在导入你的库时，测试代码并不会运行。

8.4 模块搜索路径

加载模块时，解释器会搜索`sys.path` 中的目录列表。`sys.path` 中的第一个条目通常是一个空字符串`''`，表示当前正在使用的目录。`sys.path` 中的其他条目可能包含目录名称、`.zip` 压缩文件和`.egg` 文件。各个条目在`sys.path` 中排列的顺序决定了加载模块时的搜索顺序。要将新条目添加到搜索路径中，只需将它们添加到该列表中即可。

尽管该路径通常包含目录名称，也可以将包含Python模块的zip压缩文件添加到搜索路径中。通过这种方式，可以方便地将一组模块打包为一个文件。例如，假设你创建了两个模块`foo.py` 和`bar.py`，并将它们放在一个名为`mymodules.zip` 的zip文件中。就可以按如下方式将这个文件添加到Python搜索路径中：

```
import sys
sys.path.append("mymodules.zip")
import foo, bar
```

也可以使用zip文件目录结构中的具体位置。另外，zip文件可以与常规路径名称混合使用，例如：

```
sys.path.append("/tmp/modules.zip/lib/python")
```

除了`.zip` 文件，还可以在搜索路径中添加`.egg` 文件。`.egg` 文件是由`setuptools` 库创建的包。这是安装第三方Python库和扩展时会碰到的一种常见格式。`.egg` 文件实际上只是添加了额外的元数据（如版本号、依赖关系等）的`.zip` 文件。因此，可以使用处理`.zip` 文件的标准工具来从`.egg` 文件中检查和提取数据。

尽管支持zip文件导入，还是有一些限制需要注意。首先，只能从压缩文件中导入`.py`、`.pyw`、`.pyc` 和`.pyo` 文件。使用C编写的共享库和扩展模块无法直接从压缩文件中加载，尽管`setuptools` 等打包系统有时能够提供一种变通方案（通常将C扩展提取到一个临时目录并从该目录加载模块）。而且，从压缩文件加载`.py` 文件时，Python不会创建`.pyc` 和`.pyo` 文件（稍后将介绍）。因此，一定要确保提前创建了这些文件，并将其放在归档文件中，以避免在加载模块时性能下降。

8.5 模块加载和编译

到目前为止，本章将模块描述为包含纯Python代码的文件。但是，使用`import` 加载的模块实际上可分为4个通用类别：

- 使用Python编写的代码（.py 文件）；
- 已被编译为共享库或DLL的C或C++扩展；
- 包含一组模块的包；
- 使用C编写并链接到Python解释器的内置模块。

查找模块（如foo ）时，解释器在sys.path 中的每个目录下搜索以下文件（按搜索顺序列出）：

(1) 目录foo ，它定义了一个包

(2) foo.pyd 、foo.so 、foomodule.so 或foomodule.dll （已编译的扩展）

(3) foo.pyo （只适用于使用了-O 或-OO 选项时）

(4) foo.pyc

(5) foo.py （在Windows上，Python还会查找.pyw 文件。）

稍后将介绍包，已编译的扩展将在第26章介绍。对于.py 文件，首次导入模块时，它会被编译为字节码并作为.pyc 文件写回磁盘。在后续的导入操作中，解释器将加载这段预编译的字节码，除非.py 文件有最新的修改（在这种情况下，将重新生成.pyc 文件）。.pyo 文件与解释器的-O 选项结合使用。这些文件包含已删除了行号、断言和其他调试信息的字节码。因此，这些文件会相对更小，解释器的运行速度也会稍快一些。如果指定了-OO 选项，而不是-O ，那么还会从文件中删除文档字符串。文档字符串只会在创建.pyo 文件时删除，而不是在加载它们的时候。如果sys.path 中的所有目录下都不存在这些文件，解释器将检查该名称是否为内置的模块名称。如果不存在匹配的名称，将引发importError 异常。

只有使用import 语句才能将文件自动编译为.pyc 和.pyo 文件。在命令行或标准输入中指定的程序不会生成这类文件。另外，如果包含模块的.py 文件的目录不允许写入（例如，可能是由于权限不够或者该文件包含在一个zip归档文件中），将不会创建这些文件。解释器的-B 选项也可以禁止生成这些文件。

如果存在.pyc 和.pyo 文件，则可以没有相应的.py 文件。因此，如果在打包代码时不希望包含源文件，可以只打包一组.pyc 文件。但是请注意，Python提供了对内省（introspection）和分解的广泛支持。即使没有提供源文件，细心的用户仍然可以检查并发现程序的大量细节。还请注意，.pyc 文件特定于具体的Python版本。因此，为某个Python版本生成的.pyc 文件可能不适用于未来的Python版本。

import 语句搜索文件时，文件名匹配是区分大小写的，即使机器上的底层文件系统不区分大小写也是如此，如Windows和OS X（不过这些系统会保留名称的大小写形式）。所以，import foo 将只导入文件foo.py ，不会导入文件FOO.PY 。但是，作为一般规则，应该避免使用仅大小写形式不同的模块名称。

8.6 模块重新加载和卸载

Python实际上不支持重新加载或卸载以前导入的模块。尽管可以从sys.modules 删

除模块，但这种方法通常不会从内存中卸载模块。这是因为，对模块对象的引用可能仍然存在于使用`import` 加载该模块的其他程序组件中。而且，如果存在模块中定义类的实例，这些实例将包含对其类对象的引用，类对象进而会拥有对定义它的模块的引用。

由于模块引用存在于多个位置，因此在更改了模块实现之后再重新加载该模块通常行不通。例如，如果从`sys.modules` 删除一个模块，然后使用`import` 重新加载它，不会追溯性地更改程序中以前对该模块的引用。相反，你将拥有由最新的`import` 语句创建的对新模块的引用，以及由其他部分代码中的`import` 语句创建的一组对旧模块的引用。这通常不是我们所期望的，而且在正常的生产代码中使用这种导入方式也不安全，除非你能谨慎控制整个执行环境。

早期的Python版本提供了`reload()` 函数来重新加载模块。但是，使用该函数也不是真正安全的（原因同上），而且通常不鼓励使用它，除非用作调试辅助措施。Python 3完全删除了这一功能。所以，最好不要使用它。

最后，应该注意Python的C/C++扩展无法以任何方式安全地卸载或重新加载。Python没有提供对此操作的任何支持，而且底层操作系统也可能会禁止这么做。因此，唯一的解决办法是重新启动Python解释器进程。

8.7 包

包可以将一组模块汇聚到一个共同的包名称下。这个技巧有助于解决不同应用程序中使用的模块名称之间的命名空间冲突问题。通过创建一个与包同名的目录，并在该目录中创建文件`__init__.py`，就可以定义一个包。然后，如果需要，可以向该目录中放入其他源文件、编译后的扩展和子包。例如，可以按如下形式组织一个包：

```
Graphics/
      __init__.py
      Primitive/
         __init__.py
         lines.py
         fill.py
         text.py
         ...
      Graph2d/
         __init__.py
         plot2d.py
         ...
      Graph3d/
         __init__.py
         plot3d.py
         ...
      Formats/
         __init__.py
         gif.py
      png.py
      tiff.py
      jpeg.py
```

使用`import` 语句从包中加载模块的方式很多。

- import Graphics.Primitive.fill
  这条语句加载子模块Graphics.Primitive.fill 。该模块的内容必须显式命名，如Graphics. Primitive.fill.floodfill(img,x,y,color) 。
- from Graphics.Primitive import fill
  这条语句加载子模块fill ，支持以不带包前缀的形式使用它，如fill.floodfill(img, x,y,color) 。
- from Graphics.Primitive.fill import floodfill
  这条语句加载子模块fill ，支持直接访问floodfill 函数，如floodfill(img,x,y,color) 。

只要第一次导入包中的任何部分，就会执行文件__init__.py 中的代码。这个文件可以为空，但也可以包含可执行特定于该包的初始化工作的代码。在import 语句执行期间，遇到的所有__init__.py 文件都会执行。因此，前面所列的import Graphics.Primitive.fill 语句将会首先执行Graphics 目录中的__init__.py 文件，然后执行Primitive 目录中的__init__.py 文件。

在使用包时，处理下面这条语句时需要小心：

```
from Graphics.Primitive import *
```

使用该语句的程序员通常希望将与某个包相关联的所有子模块导入到当前命名空间中。但是，由于各个系统之间的文件名约定不同（特别是在区分大小写上），Python无法准确地确定都包括哪些模块 。结果，该语句只会导入在Primitive 目录的__init__.py 文件中定义的所有名称。这种行为可以通过定义列表__all__ 来修改，该列表包含与该包相关的所有模块名称。这个列表应该在包__init__.py 文件中定义，例如：

```
# Graphics/Primitive/__init__.py
__all__ = ["lines","text","fill"]
```

现在，用户输入from Graphics.Primitive import * 语句时，将按预期加载所列出的所有子模块。

使用包时需要注意的另一个问题，与子模块想要导入同一个包中的其他子模块有关。例如，假设Graphics.Primitive.fill 模块想要导入Graphics.Primitive.lines 模块。为此，只需使用完全限定名称（如from Graphics.Primitives import lines ）或使用包的相对导入，例如：

```
# fill.py
from . import lines
```

在这个例子中，语句from . import lines 中使用的. 表示与调用模块相同的目录。因此，该语句会在与文件fill.py 相同的目录中寻找模块lines 。特别需要注意的一点是，要避免使用import *module* 这样的语句来导入包的子模块。在早期的Python版本中，无法确定import *module* 语句引用的是标准库模块还是包的子模块。早期的Python版本将首先尝试从与包含import 语句的子模块相同的包目录加载模块，如果没有

找到匹配项，则会继续搜索标准库模块。但是在Python 3中，`import` 会假定一条绝对路径，只会从标准库加载 *`module`* 。相对导入能够更清楚地表明导入意图。

相对导入也可用于加载同一个包的不同目录中包含的子模块。例如，如果模块`Graphics. Graph2D.plot2d` 想要导入`Graphics.Primitives.lines`，它可以使用如下语句：

```
# plot2d.py
from ..Primitives import lines
```

其中的`..` 用于移出一个目录级别，`Primitives` 指定一个不同的包目录。

相对导入只能使用`from` *`module`* `import` *`symbol`* 形式的导入语句来指定。因此，`import ..Primitives.lines` 或`import .lines` 这样的语句在语法上是不对的。而且， *`symbol`* 还必须是一个有效的标识符。最后，相对导入只能在一个包中使用，使用相对导入来引用位于文件系统中不同目录内的模块是不合法的。

单独导入包名称不会导入包中包含的所有子模块。例如，以下代码不会生效：

```
import Graphics
Graphics.Primitive.fill.floodfill(img,x,y,color)  # 失败!
```

但是，由于`import Graphics` 语句会执行`Graphics` 目录中的`__init__.py` 文件，所以相对导入可用于自动加载所有子模块，例如：

```
# Graphics/__init__.py
from . import Primitive, Graph2d, Graph3d

# Graphics/Primitive/__init__.py
from . import lines, fill, text, ...
```

现在`import Graphics` 语句会导入所有子模块，并支持通过完全限定名称来使用它们。再次强调一下，包相对导入应该按如上所示的形式使用。如果使用`import` *`module`* 这样的简单语句，加载的会是标准库模块。

最后，Python导入一个包时，它将定义特殊变量`__path__`，该变量包含一个目录列表，查找包的子模块时将搜索这个列表（`__path__` 是`sys.path` 变量特定于具体包的版本）。`__path__` 可通过`__init__.py` 文件中包含的代码访问，一开始只包括包的目录名这一个元素。如果有必要，包可以向`__path__` 列表提供更多目录，更改查找子模块时使用的搜索路径。如果某个包在文件系统上的组织结构很复杂，并且不能顺利地通过包的层次结构进行匹配，那么更改这个变量将很有帮助。

8.8 分发Python程序和库

要将Python程序分发给他人，应该使用`distutils` 模块。准备阶段，应该首先将你

的程序有序地组织到一个包含README文件、支持文档和源代码的目录中。通常，这个目录将包含一组库模块、包和脚本。模块和包指的是将用import 语句加载的源文件。脚本是将作为主程序由解释器运行的程序（例如，以python scriptname 这种形式运行）。下面的例子给出了一个包含Python 代码的目录：

```
spam/
     README.txt
     Documentation.txt
     libspam.py            # 一个库模块
     spampkg/              # 一个支持模块包
         __init__.py
         foo.py
         bar.py
     runspam.py            # 将作为python runspam.py运行的脚本
```

组织代码时，应确保在顶层目录中运行Python解释器时它也能够正常工作。例如，如果在spam 目录中启动Python，应该能够导入模块、导入包组件以及运行脚本，而无需更改Python的任何设置，如模块搜索路径。

组织好代码之后，在最顶层目录中（如上例中的spam ）创建文件setup.py 。在该文件中添加以下代码：

```
# setup.py
from distutils.core import setup

setup(name = "spam",
           version = "1.0",
           py_modules = ['libspam'],
           packages = ['spampkg'],
           scripts = ['runspam.py'],
           )
```

在setup() 调用中，py_modules 参数是所有单一文件Python模块的列表，packages 是所有包目录的列表，scripts 是脚本文件的列表。如果你的软件没有任何匹配的组件（例如，没有脚本），对应的参数都可以省略。name 是包的名称，version 是字符串形式的版本号。

对setup() 的调用支持很多其他参数，这些参数提供了有关包的各种元数据。表8-1列出了可以指定的最常见参数。所有值都是字符串，除了classifiers 参数，它是一个字符串列表，如['Development Status :: 4 - Beta', 'Programming Language :: Python']，完整的列表可以在http://pypi.python.org 上找到。

表**8-1　setup()** 的参数

参　数	描　述
name	包的名称（必需）

Version	版本号（必需）
author	作者名称
author_email	作者的电子邮件地址
maintainer	维护者的名称
maintainer_email	维护者的电子邮件地址
url	包的主页
description	包的简短描述
long_description	包的详细描述
download_url	包的下载位置
Classifiers	字符串分类器列表

只需创建一个setup.py 文件就足以创建软件的源代码分发版本了。输入以下shell命令即可进行源代码分发：

```
% python setup.py sdist


...
%
```

这将在目录spam/dist下创建一个压缩文件，如spam-1.0.tar.gz或spam-1.0.zip。这就是将提供给其他人安装软件的文件。要安装软件，用户只需解压该归档文件并执行以下步骤：

```
% unzip spam-1.0.zip
...
% cd spam-1.0
% python setup.py install
...
%
```

这将把软件安装在本地Python分发目录下，使其可以普遍使用 。模块和包通常安装

在Python库中名为site-packages的目录下。要找到该目录的准确位置，可以检查sys.path的值。在基于UNIX的系统上，脚本通常安装在与Python解释器相同的目录下；在Windows上，通常安装在Scripts目录下（对于典型安装，完整目录为C:\Python26\Scripts）。

在UNIX上，如果脚本第一行以#!开头并且包含文本python，安装程序将重写该行以指向本地Python安装。因此，如果编写的脚本已被硬编码为特定的Python位置，如/usr/local/bin/python，当安装在Python位于不同位置的其他系统上时，这些脚本仍然能够运行。

setup.py文件还包含其他一些与软件分发有关的命令。如果输入python setup.py bdist，将创建一个二进制分发程序，其中的所有.py文件都已预编译为.pyc文件并放置在与本地平台的目录结构类似的目录结构中。只有当应用程序的某个部分具有平台依赖关系时（例如，如果还有一个需要编译的C扩展），才需要这种分发形式。如果在Windows机器上运行python setup.py bdist_wininst，将创建一个.exe文件。打开该文件时，将启动Windows安装程序对话框，提示用户提供关于软件安装路径的信息。这种分发形式还会添加注册表项，以便以后可以方便地卸载包。

distutils模块假定用户机器上已经拥有Python安装程序（已单独下载）。尽管也可以创建软件包将Python运行时和软件绑定到一个二进制可执行文件中，但这不属于本书介绍的范围（请查看py2exe或py2app等第三方模块，了解更多细节）。如果只需要将库或简单脚本分发给他人，通常无需将Python解释器和运行时与程序代码打包在一起。

最后应该注意，除了这里介绍的选项以外，distutils还有许多其他选项。第26章将介绍如何使用distutils编译C和C++扩展。

尽管不是标准Python分发程序的一部分，但Python软件通常是以.egg文件的形式分发。这种格式可以使用流行的setuptools扩展创建（http://pypi.python.org/pypi/setuptools）。要支持setuptools，只需按如下方式更改setup.py文件的第一部分：

```
# setup.py
try:
    from setuptools import setup
except ImportError:
    from distutils.core import setup

setup(name = "spam",
      ...
)
```

8.9 安装第三方库

查找第三方库和Python扩展的权威资源，是位于http://pypi.python.org上的Python包索引（Python Package Index，PyPI）。安装第三方模块通常很简单，但是对于还要依赖于其他第三方模块的超大型包而言，可能会变得很复杂。对于更主要的扩展，通常可以找到特定平台的本机安装程序，这种安装程序使用一系列对话框来帮助你执行安装过程。对于其他模块，通常需要解压下载文件，查找setup.py文件，然后键入python setup.py

install 来安装软件。

默认情况下，第三方模块安装在Python标准库的site-packages 目录下。访问该目录通常需要根用户或管理员权限。如果不符合此条件，可以键入python setup.py install--user 将模块安装在用户的特定库目录中。这会将包安装在用户的特定目录下，如在UNIX上是/Users/beazley /.local/lib/python2.6/site-pack-ages。

如果希望将软件安装在一个完全不同的地方，可以对setup.py 命令使用--prefix 选项。例如，键入python setup.py install --prefix=/home/beazley/pypackages 会将模块安装在目录/home/beazley/pypackages 下。安装在非标准位置时，你可能必须调整sys.path 的设置，才能使Python找到最新安装的模块。

请注意，Python的很多扩展都包含C或C++代码。如果已下载了源分发版，系统必须安装C++编译器才能运行安装程序。在UNIX、Linux或OS X上，通常不会出现这样的问题。在Windows上，通常必须安装某个版本的Microsoft Visual Studio。如果使用的是Windows平台，最好找到扩展的预编译版本。

如果已安装了setuptools ，则可以使用脚本easy_install 安装包。只需键入easy_install *pkgname* 即可安装特定的包。如果配置正确，键入该命令后将从PyPI下载相应的软件和依赖项，并进行安装。当然，你也可以自己控制安装过程。

如果希望将自己的软件添加到PyPI，只需键入python setup.py register 。这会将最新版本软件的元数据上传到该索引（请注意，必须先注册用户名和密码）。

第9章　输入与输出

本章介绍Python I/O的基本知识，包括命令行选项、环境变量、文件I/O、Unicode，以及如何使用pickle 模块序列化对象。

9.1　读取命令行选项

Python启动时，命令行选项放置在列表sys.argv 中。第一个元素是程序的名称。后续项是在命令行上程序名称之后 添加的选项。下面的程序给出了手动处理简单命令行参数的最简原型：

```
import sys
if len(sys.argv) != 3:
    sys.stderr.write("Usage : python %s inputfile outputfile\n" % sys.argv[0])
    raise SystemExit(1)
inputfile  = sys.argv[1]
outputfile = sys.argv[2]
```

在该程序中，sys.argv[0] 包含所执行脚本的名称。将一条错误消息写入到sys.stderr 并引发包含非零退出代码的SystemExit 异常，这是命令行工具中报告使用错误的标准做法。

尽管你可以手动处理简单脚本的命令选项，但对于更复杂的命令行处理，可以使用optparse 模块。下面给出了一个简单示例：

```
import optparse
p = optparse.OptionParser()

# 该选项接受一个参数
p.add_option("-o",action="store",dest="outfile")
p.add_option("--output",action="store",dest="outfile")

# 该选项设置一个布尔值标志
p.add_option("-d",action="store_true",dest="debug")
p.add_option("--debug",action="store_true",dest="debug")

# 设置选定选项的默认值
p.set_defaults(debug=False)

# 解析命令行
opts, args = p.parse_args()

# 检索选项设置
outfile   = opts.outfile
debugmode = opts.debug
```

在这个例子中添加了两类选项。第一个选项-o 或--output 具有一个必需的参数。这

种行为是通过在对p.add_option() 的调用中指定action='store' 来选定的。第二个选项-d 或--debug 仅用于设置布尔值标志。这是通过在p.add_option() 中指定action='store_true' 来实现的。p.add_option() 的dest 参数选择将在解析后存储参数值的属性名称。p.set_defaults() 方法设置一个或多个选项的默认值。该方法所使用的参数名称应该与为每个选项选择的目标名称匹配。如果没有选定默认值，则默认值将设为None 。

上面的程序能够识别以下所有命令行样式：

```
% python prog.py -o outfile -d infile1 ... infileN
% python prog.py --output=outfile --debug infile1 ... infileN
% python prog.py -h
% python prog.py --help
```

使用p.parse_args() 方法执行解析。该方法返回一个2元组(opts, args) ，其中opts 是包含已解析选项值的对象，args 是命令行上未解析为选项的项列表。选项值使用opts.dest 检索，其中dest 是在添加选项时使用的目标名称。例如，-o 或--output 参数放置在opts.outfile 中，而args 是剩余参数组成的列表，如['infile1', ..., 'infileN'] 。用户请求时，optparse 模块自动提供-h 或--help 选项来列出可用选项。糟糕的选项也会导致错误消息的出现。

这个例子仅仅展示了optparse 模块的最简单用法。有关更高级选项的介绍参见第19章。

9.2 环境变量

可以通过字典os.environ 访问环境变量，例如：

```
import os
path   = os.environ["PATH"]
user   = os.environ["USER"]
editor = os.environ["EDITOR"]
... etc ...
```

可以通过设置os.environ 变量来修改环境变量。例如：

```
os.environ["FOO"] = "BAR"
```

修改os.environ 会同时影响到正在运行的程序和Python创建的子进程。

9.3 文件和文件对象

内置函数open(*name* [,*mode* [,*bufsize*]]) 用于打开和创建文件对象，如下所示：

```
f = open("foo")        # 打开"foo"以供读取
f = open("foo",'r')    # 打开"foo"以供读取（同上）
f = open("foo",'w')    # 打开对象以进行写入
```

文件模式'r'表示读取，'w'表示写入，'a'表示附加。这些文件模式假定采用文本模式打开文件，可以隐式地对换行字符'\n'执行转换。例如，在Windows上，写入字符'\n'实际上会输出2字符序列'\r\n'（并且读取该文件时，'\r\n'又会被转换为一个'\n'字符）。如果正在处理二进制数据，可以将'b'附加到文件模式后面，如'rb'或'wb'。这将禁用换行符转换，如果关注处理二进制数据的代码的可移植性，则应该包含'b'（在UNIX上，一个常见错误就是省略了'b'，因为文本与二进制文件之间没有任何区别）。另外，由于模式之间的区别，也可能看到指定为'rt'、'wt'或'at'的文本模式，这样能更清楚地表明意图。

通过提供加号（+）字符（如'r+'或'w+'），可以打开文件进行原地更新。打开文件进行更新时，可以同时执行输入和输出，只要所有输出操作在任何后续输入操作之前清除其数据即可。如果使用'w+'模式打开文件，其长度首先会被截断为0。

如果使用模式'U'或'rU'打开文件，将会提供通用的换行符支持，以方便阅读。在由各种文件I/O函数返回的字符串中，该功能可将不同的换行符编码（如'\n'、'\r'和'\r\n'）转换为标准'\n'字符，从而简化跨平台工作。举例而言，如果在UNIX系统上编写的脚本必须处理由Windows程序生成的文本文件，这项功能会很有用。

可选的 *bufsize* 参数控制文件的缓冲行为，其中0表示没有缓冲，1表示进行了行缓冲，而负值要求采用系统默认设置。任何其他正值都表示将使用的近似缓冲区大小（以字节为单位）。

Python 3向open()函数添加了4个额外的参数，它的调用方式如下：open(*name* [,*mode* [,*bufsize* [, *encoding* [, *errors* [, *newline* [, *closefd*]]]]]])。*encoding* 是一个编码名称，如'utf-8'或'ascii'。 *errors* 是处理编码错误的错误处理策略（如需了解更多信息，请参见本章后面有关Unicode的内容）。 *newline* 控制通用换行符模式的行为，可以设置为None、''、'\n'、'\r'或'\r\n'。如果设为None，'\n'、'\r'或'\r\n'形式的结尾符都会被转换为'\n'。如果设为''（空字符串），所有这些行结束形式都会被识别为换行符，但在输入文本中不会转换。如果 *newline* 拥有任何有效的值，该值将用于结束各行。 *closefd* 控制在调用close()方法时，是否实际关闭底层文件描述符。默认情况下，该值设为True。

表9-1列出了file对象支持的方法。

表**9-1** 文件方法

方　法	描　述
f.read([n])	最多读取 *n* 个字节
f.readline([n])	读取单行输入的最多 *n* 个字符。如果省略了 *n*，该方法将读取整行

f.readlines([size])	读取所有行并返回一个列表。 *size* 是可选的，用于指定在读取操作停止前在文件上读取的近似字符数
f. write(*s*)	写入字符串 *s*
f. writelines(*lines*)	写入序列 *lines* 中的所有字符串
f. close()	关闭文件
f. tell()	返回当前文件指针
f. seek(*offset [, whence]*)	查找新文件位置
f. isatty()	如果 *f* 是一个交互式终端，则返回1
f. flush()	清除输出缓冲区
f. truncate(*[size]*)	将文件截断为最多 *size* 字节
f. fileno()	返回一个整数文件描述符
f. next()	返回下一行或引发StopIteration 。在Python 3中，对应的方法称为 *f* .__next__()

read() 方法以字符串的形式返回整个文件，除非使用可选的 *length* 参数指定了最大字符数。readline() 方法返回下一行输入，包括最后的换行符。readlines() 方法以字符串列表的形式返回所有输入行。readline() 方法可以接受的最大行长度为 *n* 。如果读取的一行比 *n* 个字符多，将返回前 *n* 个字符。行中的剩余数据将不会被丢弃，并在执行后续读取操作时返回。readlines() 方法接受一个 *Size* 参数，指定在停止读取之前要读取的近似字符数。实际读取的字符数可能比这个数字大，具体取决于缓存了多少数据。

readline() 和readlines() 方法都能够识别各种平台，正确处理不同的换行符表示形式（如'\n' 与'\r\n' ）。如果文件在通用换行符模式（'U' 或'rU' ）中打开，换行符将被转换为'\n' 。

read() 和readline() 返回一个空字符串来表示文件结束（EOF）。以下代码展示了如何检测EOF条件：

```
while True:
    line = f.readline()
    if not line:        # EOF
        break
```

读取文件中所有行的一种便捷方式是用for 循环进行迭代。例如：

```
for line in f:        # 迭代文件中的所有行
    # 对某一行执行特定操作
    ...
```

注意，在Python 2中，各种读取操作始终返回8位字符串，无论指定的文件模式是什么（是文本还是二进制）。在Python 3中，如果在文本模式下打开文件，这些操作将返回Unicode字符串；如果在二进制模式下打开文件，将返回字节字符串。

write() 方法将一个字符串写入到文件中，writelines() 方法将一个字符串列表写入到文件中。write() 和writelines() 不会将换行字符添加到输出中，所以生成的所有输出都应该已经包含所有必要的格式。这些方法可以将原始字节字符串写入到文件中，但只有以二进制模式打开文件时才能实现。

在内部，每个文件对象都有一个文件指针，用于存储下次读取或写入操作所需的字节偏移位置。tell() 方法以长整型返回文件指针的当前值。seek() 方法根据给定的 *offset* 和 *whence* 中的位置规则随机访问文件的各个部分。如果 *whence* 为0（默认值），seek() 假设 *offset* 为相对于文件的开头的偏移量；如果 *whence* 为1，将参照当前位置偏移；如果 *whence* 为2，将相对于文件末尾进行偏移。seek() 以整数形式返回文件指针的新值。应该注意，文件指针与open() 返回的文件对象相关联，而不是与文件本身相关联。可以在同一个程序中（或在不同程序中）多次打开同一个文件。每个打开文件的实例都拥有自己的文件指针，可以独立操作该指针。

fileno() 方法返回文件的整数文件描述符，有时用在某些库模块中的底层I/O操作中。例如，在UNIX系统上，fcntl 模块使用文件描述符来提供底层文件控制操作。

文件对象还拥有一些只读数据属性，如表9-2所示。

表**9-2**　文件对象属性

属　　性	描　　述
f. closed	布尔值，表示文件状态：如果文件已打开则为False ，如果文件已关闭则为True
f. mode	文件的I/O模式
f. name	如果使用open() 创建文件，则为文件的名称。否则，它将是一个表示文件来源的字符串
f. softspace	布尔值，指示在使用print 语句时，是否应该在一个值之前打印空格字符。模仿文件的类必须提供该名称的一个可写属性，该属性初始化为0（仅在Python 2中使用）
	在通用换行符模式下打开一个文件时，该属性包含可在文件中实际找到的换行符表示。如果没

f.newlines	有遇到换行符，该值为None，将会看到一个包含'\n'、'\r'或'\r\n'的字符串，或者一个包含所有不同换行符编码的元组
f.encoding	一个字符串，指示文件编码（如果存在）（例如'latin-1'或'utf-8'）。如果没有使用任何编码，该值为None

9.4 标准输入、输出和错误

解释器提供了3种标准文件对象，分别为标准输入、标准输出和标准错误，它们在sys模块中分别以sys.stdin、sys.stdout和sys.stderr的形式提供。stdin是与提供给解释器的输入字符流相对应的文件对象。stdout是一个接收由print生成输出的文件对象。stderr是接收错误消息的文件。通常，stdin被映射到用户的键盘，而stdout和stderr在屏幕上生成文本。

上一节描述的方法可用于执行用户的原始I/O。例如，以下代码写入标准输出并从标准输入中读取一行输入：

```
import sys
sys.stdout.write("Enter your name : ")
name = sys.stdin.readline()
```

另外，内置函数raw_input*(prompt)*也可以从stdin读取一行文本，并可以打印一个提示符：

```
name = raw_input("Enter your name : ")
```

raw_input()读取的行不包含行末的换行符。这与直接从sys.stdin读取不同，在sys.stdin中，换行符都包含在输入文本中。在Python 3中，raw_input()已被重命名为input()。

键盘中断（通常由Ctrl+C生成）会生成KeyboardInterrupt异常，可使用异常处理程序捕获该异常。

如有必要，sys.stdout、sys.stdin和sys.stderr的值可以替换为其他文件对象，在这种情况下，print语句和输入函数将使用新值。如果需要还原sys.stdout的原始值，首先应该保存该值。在解释器启动时，sys.stdout、sys.stdin和sys.stderr的原始值可以分别在sys.__stdout__、sys.__stdin__和sys.__stderr__中获得。

注意，在某些情况下，使用集成开发环境（IDE）时可能会更改sys.stdin、sys.stdout和sys.stderr。例如，当Python在IDLE下运行时，sys.stdin将被替换为一个行为类似文件的对象，但它其实是开发环境中的一个对象。在这种情况下，某些底层方法（如read()和seek()）可能不可用。

9.5 print语句

Python 2有一个特殊的print 语句，可根据sys.stdout 中包含的文件生成输出。print 接受一个逗号分隔的对象列表，例如：

```
print "The values are", x, y, z
```

对于每个对象，将调用str() 函数来生成输出字符串。这些输出字符串然后会连接在一起，彼此之间用一个空格分开，从而得到最终的输出字符串。输出通过一个换行符终止，除非为print 语句提供了后置逗号。在这种情况下，下一条print 语句将在打印更多项目之前插入一个空格。该空格的输出由用于输出的文件的softspace 属性控制。

```
print "The values are ", x, y, z, w
# 打印相同的文本，使用两条print语句
print "The values are ", x, y,    # 省略结束的换行符
print z, w                        # 在z的前面打印一个空格
```

要生成格式化输出，可以使用第4章中介绍的字符串格式运算符（% ）或.format() 方法。下面给出一个例子：

```
print "The values are %d %7.5f %s" % (x,y,z) # 已格式化的I/O
print "The values are {0:d} {1:7.5f} {2}".format(x,y,z)
```

可以更改print 语句的目标，方法是添加特殊的>>*file* 修饰符和一个逗号，其中 *file* 是一个允许写入的文件对象，例如：

```
f = open("output","w")
print >>f, "hello world"
...
f.close()
```

9.6 print() 函数

Python 3中最重要的更改之一是print 被转变为函数。在Python 2.6中，如果在使用的每个模块中包含了语句from __future__ import print_function ，也可以将print 用作函数。print() 函数的工作方式与上一节中介绍的print 语句非常相似。

要打印一系列以空格分隔的值，只需将这些值提供给print() ，例如：

```
print("The values are", x, y, z)
```

要禁止或更改行终止，可以使用end=*ending* 关键字参数。例如：

```
print("The values are", x, y, z, end='')    # 禁止换行符
```

要将输出重定向到一个文件，可以使用file=outfile 关键字参数。例如：

```
print("The values are", x, y, z, file=f)    # 重定向到文件对象f
```

要更改项之间的分隔字符，可以使用sep=sepchr 关键字参数。例如：

```
print("The values are", x, y, z, sep=',')   # 在值之间添加逗号
```

9.7 文本输出中的变量插入

生成输出时的一个常见问题是，生成其中包含了嵌入式变量替换的大型文本片段。很多脚本语言（如Perl和PHP）都允许用$- 变量替换形式（如$name 、$address 等）将变量插入到字符串中。Python无法直接实现这一功能，但可以通过将格式化I/O与三引号的字符串来模仿这种行为。例如，可以如下例所示编写一封简单的信函，在其中填写一个name 、一个item 名称和一个amount ：

```
# 注意："""后紧跟的结束斜杠可以防止第一行显示为空行
form = """\
Dear  %(name)s,
Please send back my %(item)s or pay me $%(amount)0.2f.
                                  Sincerely yours,

                                  Joe Python User
"""
print form % { 'name': 'Mr. Bush',
               'item': 'blender',
               'amount': 50.00,
                 }
```

这段代码生成以下输出：

```
Dear Mr. Bush,

Please send back my blender or pay me $50.00.

                                  Sincerely yours,

                                  Joe Python User
```

format() 方法是一种更加先进的替代方法，它可以使上面的代码更加简洁。例如：

```
form = """\
Dear {name},
Please send back my {item} or pay me {amount:0.2f}.
                                  Sincerely yours,

                                  Joe Python User
"""
```

```
print form.format(name='Mr. Bush', item='blender', amount=50.0)
```

对于某些格式类型，也可以使用Template 字符串，如下所示：

```
import string
form = string.Template("""\
Dear $name,
Please send back my $item or pay me $amount.
                                  Sincerely yours,

                                  Joe Python User
""")
print form.substitute({'name': 'Mr. Bush',
                       'item': 'blender',
                       'amount': "%0.2f" % 50.0})
```

在这个例子中，字符串中的特殊$变量表示替换。form.substitute() 方法获取一个替换字典，返回一个新字符串。尽管前面的方法很简单，但它们并不是最强大的文本生成方案。Web框架和其他大型应用程序框架倾向于提供自己的模板字符串引擎，这些引擎支持嵌入式控制流、变量替换、文件包含和其他高级功能。

9.8 生成输出

直接处理文件是程序员最熟悉的I/O模型。但是，生成器函数也可用于以一个数据片段序列的形式输出I/O流。为此，只需使用yield 语句，就像使用write() 或print 语句一样，例如：

```
def countdown(n):
    while n > 0:
         yield "T-minus %d\n" % n
         n -= 1
    yield "Kaboom!\n"
```

这种输出流生成方式非常灵活，因为输出流的生成与将输出流实际引导至期望目的地的代码是分开的。例如，如果希望将上述输出发送到文件 *f* ，可以这样做：

```
count = countdown(5)
f

.writelines(count)
```

如果希望将输出重定向到套接字 *s* ，可以这样做：

```
for chunk in count:
    s

.sendall(chunk)
```

或者，如果只想将所有输出捕获到一个字符串中，可以这样做：

```
out = "".join(count)
```

更高级的应用程序可以使用这种方法来实现自己的I/O缓冲。例如，一个生成器可以输出小文本片段，但另一个函数可以将这些片段收集到大型缓冲区中，以实现较大的、更高效的I/O操作：

```
chunks = []
buffered_size = 0
for chunk in count:
    chunks.append(chunk)
    buffered_size += len(chunk)
    if buffered_size >= MAXBUFFERSIZE:
         outf.write("".join(chunks))
         chunks.clear()
         buffered_size = 0
outf.write("".join(chunks)
```

对于将输出发送到文件或网络连接的程序，生成器方法还可以显著减少内存的使用，因为整个输出流通常可以在较小的片段中生成和处理，而不需要首先收集到一个大型输出字符串或字符串列表中。编写程序来与Python Web服务网关接口（Web Services Gateway Interface，WSGI）交互时，可能会用到这种输出方法，WSGI用于在Web框架的一些组件之间进行通信。

9.9 Unicode字符串处理

与I/O处理相关的一个常见问题是，处理表示为Unicode的国际字符。如果有一个原始字节字符串 *s* ，其中包含已编码的Unicode字符串表示，那么可以使用 *s* .decode([*encoding* [,*errors*]]) 方法将其转换为真正的Unicode字符串。要将Unicode字符串 *u* 转换为已编码的字节字符串，可以使用字符串方法 *u* .encode([*encoding* [, *errors*]]) 。这两种转换运算符都需要使用一个特殊编码名称，指定如何在Unicode字符值与字节字符串中的一个8位字符序列上建立起相互映射关系。编码参数以字符串的形式指定，是一百多种不同字符编码中的一种。以下值是最常用的。

值	描　　述
'ascii'	7位ASCII码
'latin-1' 或'iso-8859-1'	ISO 8859-1 Latin-1

'cp1252'	Windows 1252编码
'utf-8'	8位变长编码
'utf-16'	16位变长编码（可以为小尾或大尾）
'utf-16-le'	UTF-16，小尾编码
'utf-16-be'	UTF-16，大尾编码
'unicode-escape'	与Unicode字面量u"string" 相同的格式
'raw-unicode-escape'	与Unicode字面量ur"string" 相同的格式

默认编码在site 模块中设置，可以使用sys.getdefaultencoding() 查询。在很多情况下，默认编码是'ascii'，也就是说值在[0x00，0x7f] 范围内的ASCII字符可以直接映射到[U+0000，U+007F] 范围内的Unicode字符。'utf-8' 也是一种非常常见的设置。与常见编码相关的技术细节将在后面介绍。

使用 *s* .decode() 方法时，始终假定 *s* 是一个字节字符串。在Python 2中，这表示 *s* 是一个标准字符串；但在Python 3中， *s* 必须是一种特殊的bytes 类型。类似地，*t* .encode() 的结果始终是一个字节序列。如果注重可移植性，那么需要注意，这些方法在Python中稍微有点混乱。例如，Python 2字符串同时拥有decode() 和encode() 方法，而在Python 3中，字符串只有一个encode() 方法，bytes 类型只有一个decode() 方法。要简化Python 2中的代码，请确保仅对Unicode字符串使用encode() 方法，仅对字节字符串使用decode() 方法。

转换字符串值时，如果遇到无法转换的字符，可能会引发UnicodeError 异常。例如，如果尝试将一个字符串编码为'ascii'，而它包含一个Unicode字符（如U+1F28），将会遇到编码错误，因为该字符值太大，无法使用ASCII字符集表示。encode() 和decode() 方法的errors 参数决定了如何处理编码错误。该参数是一个包含以下值的字符串。

值	描　　述
'strict'	遇到编码和解码错误时，引发UnicodeError 异常
'ignore'	忽略无效字符
'replace'	将无效字符替换为一个替换字符（Unicode中的U+FFFD，标准字符串中的'?'）

'backslashreplace'	将无效字符替换为Python字符转义序列。例如，将字符U+1234替换为'\u1234'
'xmlcharrefreplace'	将无效字符替换为XML字符引用。例如，将字符U+1234替换为'ሴ'

默认错误处理被设置为'strict'。

要在网页上将国际字符嵌入到ASCII编码的文本中，'xmlcharrefreplace'错误处理策略通常很有用。例如，如果输出Unicode字符串'Jalape\u00f1o'并使用'xmlcharrefreplace'处理方法将它编码为ASCII字符，浏览器始终能够将输出文本正确呈现为“Jalapeño”，而不是乱码。

为了便于记忆，一定不要在表达式中混合使用已编码的字节字符串和未编码的字符串（例如使用+来连接字符串）。Python 3禁止这样做，但Python 2将静默地支持这种操作，根据默认编码设置自动将字节字符串转换为Unicode。这种行为通常会导致意外结果或无法理解的错误消息。因此，应该尽可能在程序中区分已编码和未编码的字符数据。

9.10 Unicode I/O

处理Unicode字符串时，无法将原始Unicode数据直接写入文件。这是因为Unicode字符在内部表示为多字节整数，而且将这些整数直接写入到输出流将导致与字节顺序相关的问题。例如，你需要决定是采用“小尾”格式将Unicode字符U+HHLL写为字节序列LL HH，还是采用“大尾”格式写为字节序列HH LL。而且，处理Unicode的其他工具必须知道你所使用的编码形式。

因此，Unicode字符串的外部表示总是根据具体的编码规则来进行，该编码规则应该明确定义如何将Unicode字符表示为字节序列。因此，要支持Unicode I/O，需要将上一节中介绍的编码和解码概念扩展到文件。内置的codecs模块包含一组函数，用于根据各种不同的数据编码方案，在字节数据与Unicode字符串之间来回转换。

处理Unicode文件最直接方式是用codecs.open(*filename* [,*mode* [,*encoding* [,*errors*]]]函数，如下所示：

```
f = codecs.open('foo.txt','r','utf-8','strict')      # 读取
g = codecs.open('bar.txt','w','utf-8')               # 写入
```

这会创建一个文件对象，用于读取或写入Unicode字符串。编码参数指定将在文件中读取或写入数据时，用于转换数据的底层字符编码。 *errors* 参数决定如何处理错误，处理方式可以为上一节中介绍的'strict'、'ignore'、'replace'、'backslashreplace'或'xmlcharrefreplace'之一。

如果已经拥有一个文件对象，可以使用codecs.EncodedFile(*file , inputenc* [, *outputenc* [, *errors*]])函数为该对象添加一个编码包装器，例如：

```
f = open("foo.txt","rb")
...
```

```
fenc = codecs.EncodedFile(f,'utf-8')
```

在这个例子中，根据 *inputenc* 中提供的编码对从文件中读取的数据进行解释。根据 *inputence* 中提供的编码对写入文件的数据进行解释，并根据 *outputence* 中的编码来写入该数据。如果省略了 *outputence* ，则默认编码与 *inputence* 相同。*errors* 与前面介绍的含义相同。为现有文件添加 *EncodedFile* 包装器时，应确保该文件采用二进制模式。否则，换行符转换可能会违背编码规则。

处理Unicode文件时，数据编码通常嵌入到文件本身中。例如，XML解析器可能会查找字符串'<?xml ...>' 的前几个字节来决定文档编码。如果前4个值为'3C 3F 78 6D（'<?xm'），则认为编码是UTF-8。如果前4个值是00 3C 00 3F 或3C 00 3F 00，则认为编码分别为UTF-16大尾或UTF-16小尾。另外，文档编码也可以出现在MIME头部，或者显示为其他文档元素的属性，例如：

```
<?xml ... encoding="ISO-8859-1" ... ?>
```

类似地，Unicode文件也能包含特殊的BOM（Byte-Order Marker，字节顺序标记），指示字节编码的特性。Unicode字符U+FEFF 就是为该用途而保留的。通常，BOM作为文件中的第一个字符写入。程序读取该字符并查看字节的排列顺序，从而确定编码（例如'\xff\xfe' 的编码为UTF-16-LE，'\xfe\xff' 的编码为UTF-16-BE）。一旦确定了编码，就会丢弃BOM字符，然后处理文件的剩余部分。然而，这种额外的BOM处理不会在后台进行。通常，如果应用程序使用到这一功能，你必须给予注意。

从文档读取编码时，可以使用以下代码将输入文件转换为已编码的数据流：

```
f = open("somefile","rb")
# 确定文件的编码
...
# 为文件添加一个合适的编码包装器。
# 假设BOM（如果存在）已被前面的语句丢弃。
fenc = codecs.EncodedFile(f,encoding)
data = fenc.read()
```

9.10.1 Unicode数据编码

表9-3列出了codecs 模块中一些最常用的编码器。

表**9-3 codecs** 模块中的编码器

编 码 器	描 述
'ascii'	ASCII编码
'latin-1'、'iso-8859-1'	Latin-1或ISO-8859-1编码

'cp437'	CP437编码
'cp1252'	CP1252编码
'utf-8'	8位变长编码
'utf-16'	16位变长编码
'utf-16-le'	UTF-16，采用显式小尾编码
'utf-16-be'	UTF-16，采用显式大尾编码
'unicode-escape'	与u"*string* "格式相同
'raw-unicode-escape'	与ur"*string* "格式相同

下面分别详细介绍每个编码器。

1. 'ascii' 编码

在'ascii' 编码中，字符值被限定在[0x00，0x7f] 和[U+0000，U+007F] 范围之内。此范围外的任何字符都是无效的。

2. 'iso-8859-1' 、**'latin-1'** 编码

字符可以是[0x00，0xff] 和[U+0000，U+00FF] 范围内的任何8位值。[0x00，0x7f] 范围内的值与ASCII字符集中的字符对应。[0x80，0xff] 范围内的值与ISO-8859-1或扩展的ASCII字符集中的字符对应。值在[0x00，0xff] 范围外的任何字符都会导致错误。

3. 'cp437' 编码

该编码类似于'iso-8859-1'，但它是Python在作为Windows上的控制台应用程序运行时的默认编码。[x80，0xff] 范围内的某些字符与遗留DOS应用程序中用于呈现菜单、窗口和帧的特殊符号对应。

4. 'cp1252' 编码

这种编码与Windows上使用的'iso-8859-1' 非常相似。但是，该编码定义[0x80，0x9f] 范围内未在'iso-8859-1' 中定义且在Unicode中具有不同码点的字符。

5. 'utf-8' 编码

UTF-8是一种变长编码，可以表示所有Unicode字符。使用单个字节表示0～127范围

内的ASCII字符。所有其他字符使用2或3字节组成的多字节序列表示。这些字节的编码如下所示。

Unicode字符	字节**0**	字节**1**	字节**2**
U+0000 - U+007F	0*nnnnnnn*		
U+007F - U+07FF	110*nnnnn*	10*nnnnnn*	
U+0800 - U+FFFF	1110*nnnn*	10*nnnnnn*	10*nnnnnn*

对于2字节序列，第一个字节始终以位序列`110` 开头。对于3字节序列，第一个字节始终以位序列`1110` 开头。多字节序列中所有后续数据字节都以位序列`10` 开头。

概括来讲，UTF-8格式允许最多6字节的多字节序列。在Python中，4字节`UTF-8` 序列用于编码一个称为代理对 的Unicode字符对。代理对中两个字符的值都在`[U+D800, U+DFFF]` 范围内，并被组合起来编码成一个20位的字符值。代理项编码如下所示：4字节序列`11110`*`nnn`* `10`*`nnnnnn`* `10nmmmm 10mmmmm` 被编码为`U+D800 + N` 和`U+DC00 + M` 对，其中`N` 是编码为4字节UTF-8序列中的20位字符的高10位，`M` 是低10位。5和6字节UTF-8序列（分别由最前面的位序列`111110` 和`1111110` 表示）用于编码长度达32位的字符值。Python不支持这些值，如果在已编码的数据流中出现了这些值，将导致`UnicodeError` 异常。

UTF-8编码具有很多有用的特性，使其可以支持早期的软件。首先，标准ASCII字符采用标准编码来表示。这意味着UTF-8编码的ASCII字符串与传统ASCII字符串是没有差别的。其次，UTF-8不会为多字节字符序列引入嵌入式NULL字节。因此，基于C库的现有软件和要求8位字符串以NULL终结的程序将采用UTF-8字符串。最后，UTF-8编码会保持字符串的字母顺序。也就是说，如果`a` 和`b` 是Unicode字符串，并且`a<b` ，那么当`a` 和`b` 被转换为UTF-8时，`a<b` 仍然成立。因此，排序算法和其他为8位字符串编写的排序算法同样适用于UTF-8。

6. `'utf-16'` 、`'utf-16-be'` 和`'utf-16-le'` 编码

UTF-16是一种变长16位编码，采用这种编码方式时，Unicode字符被写为16位值。除非指定了字节顺序，否则将假定采用大尾编码。另外，字节顺序标记`U+FEFF` 可用于显式指定UTF-16数据流中的字节顺序。在大尾编码中，`U+FEFF` 是零宽非断行空格（nonbreaking space）的Unicode字符，而保留值`U+FFFE` 是一个无效的Unicode字符。因此，编码器可以使用字节序列`FE FF` 或`FF FE` 来确定数据流的字节顺序。读取Unicode数据时，Python会从最终的Unicode字符串中删除字节顺序标记。

`'utf-16-be'` 编码明确选择了UTF-16大尾编码。`'utf-16-le'` 明确选择了UTF-16小尾编码。

尽管UTF-16的一些扩展可以支持比16位更大的字符值，但目前Python不支持所有这些扩展。

7. 'unicode-escape' 和'raw-unicode-escape' 编码

这两种编码方法用于将Unicode字符串转换为Python中的Unicode字符串字面量和Unicode原始字符串字面量中使用的格式，例如：

```
s = u'\u14a8\u0345\u2a34'
t = s.encode('unicode-escape')  #t = '\u14a8\u0345\u2a34'
```

9.10.2 Unicode字符特性

除了执行I/O，使用Unicode的程序可能还需要测试Unicode字符的各种特性，例如大小写形式、数字和空格。unicodedata 模块能够访问一个字符特性数据库。常见的字符特性可使用unicodedata.category(*c*) 函数获得。例如，unicodedata.category(u"A") 返回'Lu'，指示该字符是一个大写字符。

Unicode字符串的另一个棘手问题是，同一个Unicode字符串可以有多种表示。例如，字符U+00F1 （ñ） 可以完全组合为单一字符U+00F1，或者分解为多字符序列U+006eU+0303 （n,˜） 。如果需要一致地处理Unicode字符串，可以使用unicodedata.normalize() 函数来确保一致的字符表示。例如，unicodedata.normalize('NFC', *s*) 将确保 *s* 中的所有字符都完全组合在一起，而不会表示为一个字符组合序列。

关于Unicode字符数据库和unicodedata 模块的更多细节可以在第16章找到。

9.11 对象持久化与pickle 模块

最后，常常需要将对象内容保存到一个文件中或从中进行还原。解决该问题的方法之一是编写两个函数，以一种特殊格式在文件中读取和写入数据。另一种方法是用pickle 和shelve 模块。

pickle 模块将对象序列化为一个字节流，这个字节流可以写入到文件并在以后进行还原。pickle 的接口非常简单，只包含dump() 和load() 操作。例如，以下代码将一个对象写入一个文件：

```
import pickle
obj = SomeObject()
f = open(filename,'wb')
pickle.dump(obj, f)      # 将对象保存到f上
f.close()
```

要还原对象，可以使用以下代码：

```
import pickle
f = open(filename,'rb')
obj = pickle.load(f)   # 还原对象
f.close()
```

可以依次发出一系列dump() 操作来保存对象序列。要还原这些对象，只需使用一个类似的load() 操作序列。

shelve 模块类似于pickle ，但它将对象保存在一个类似字典的数据库中：

```
import shelve
obj = SomeObject()
db = shelve.open("filename")  # 打开一个shelve
db['key'] = obj                # 将对象保存在shelve中
...
obj = db['key']                # 检索对象
db.close()                     # 关闭shelve
```

尽管由shelve 创建的对象类似于一个字典，但它也有一些限制。首先，键必须是字符串。其次，shelve 中存储的值必须与pickle 兼容。大部分Python对象都可以这样存储，但具有特殊用途的对象（如文件和网络连接）需要保持一种内部状态，这种状态无法通过这种方式保存和还原。

pickle 使用的数据格式是Python所专用的。但是，随着Python版本的升级，该格式也被多次改进。可以使用pickle 中dump(*obj , file , protocol*) 操作的一个可选协议参数来选择协议。

默认情况下使用协议0。这是最古老的pickle 数据格式，它将对象存储为几乎所有Python版本都能够理解的格式。但是，这种格式与Python很多更先进的用户定义类特性（如slot）不兼容。协议1和协议2使用一种更高效的二进制数据表示。要使用这些替代协议，可以执行如下操作：

```
import pickle
obj = SomeObject()
f = open(filename,'wb')
pickle.dump(obj,f,2)                              # 使用协议2保存
pickle.dump(obj,f,pickle.HIGHEST_PROTOCOL)   # 使用最先进的协议
f.close()
```

使用load() 还原对象时，不必指定协议。底层协议已被编码到文件中。

类似地，可以打开一个shelve 来使用替代的pickle 协议保存Python对象，如下所示：

```
import shelve
db = shelve.open(filename,protocol=2)
...
```

用户定义的对象通常无需执行任何额外的操作，就可以支持pickle 或shelve 。但是，可以使用特殊方法__getstate__() 和__setstate__() 来协助组合（pickle）过程。如果已定义了__getstate__() 方法，那么将调用它来创建一个表示对象状态的值。__getstate__() 返回的值通常应该是一个字符串、元组、列表或字

典。__setstate__() 方法在拆解（unpickle ）期间接收此值，并且应该可以从该值还原对象状态。下面的例子展示了如何使用这些方法来处理一个包含底层网络连接的对象。尽管无法组合实际连接，但对象保存的信息已足够在拆解连接之后重建连接：

```
import socket
class Client(object):
    def __init__(self,addr):
        self.server_addr = addr
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect(addr)
    def __getstate__(self):
        return self.server_addr
    def __setstate__(self,value):
        self.server_addr = value
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect(self.server_addr)
```

由于pickle 使用的数据格式是Python专用的，所以在与使用不同编程语言编写的应用程序交换数据时，不应使用该功能。而且，出于安全的考虑，程序不应该处理来自不可信来源的已组合的数据。（在拆解期间，经验丰富的攻击者可以操作组合数据格式来执行系统命令。）

pickle 和shelve 模块还拥有很多自定义功能和高级使用选项。更多细节请参见第13章。

第10章　执行环境

本章介绍Python程序的执行环境。目的是说明解释器的运行时行为，包括程序启动、配置和程序终止。

10.1　解释器选项与环境

解释器具有很多可控制其运行时行为和环境的选项。可按以下方式在命令行上为解释器提供选项：

```
python [options
] [-c cmd
 | filename
 | - ] [args
]
```

表10-1列出了最常用的命令行选项。

表**10-1**　解释器命令行参数

选　　项	描　　述
-3	启用将从Python 3中删除或更改某些功能的警告
-B	阻止在导入时创建.pyc 或.pyo 文件
-E	忽略环境变量
-h	打印所有可用命令行选项的列表
-i	在程序执行后进入交互模式
-m module	以脚本的形式运行库模块 *module*
-O	优化模式

-OO	优化模式，在创建.pyo 文件时删除文档字符串
-Qarg	指定Python 2中除法运算符的行为，值为-Qold （默认值）、-Qnew 、-Qwarn 或-Qwarnall 之一
-s	阻止将用户站点目录添加到sys.path
-S	阻止包含site 初始化模块
-t	报告关于不一致的制表符使用警告
-tt	由于不一致的制表符使用而导致TabError 异常
-u	未缓冲的二进制stdout 和stdin
-U	Unicode字面量。所有字符串字面量都以Unicode形式处理（仅在Python 2中使用）
-v	详细模式。跟踪导入语句
-V	打印版本号并退出
-x	跳过源程序的第一行
-c cmd	以字符串形式执行 *cmd*

-i 选项在程序执行完成之后立即启动一个交互式会话，对于调试很有用。-m 选项以脚本的形式运行库模块，在执行主脚本之前在__main__ 模块内部执行。-O 和-OO 选项对字节编译文件应用一些优化，已在第8章介绍。-S 选项省略10.4节中介绍的site 初始化模块。-t 、-tt 和-v 选项报告额外的警告和调试信息。如果程序中的第一行不是一条有效的Python语句，-x 选项可以忽略第一行。（例如，当第一行启动脚本中的Python解释器时。）

程序名称出现在所有解释器选项之后。如果没有给定名称，或者连字符（- ）被用作文件名，解释器将从标准输入读取程序。如果标准输入是一个交互式终端，将会显示一个标题和一个提示符。否则，解释器将打开指定文件并执行其语句，直至到达文件结束标记为止。-c *cmd* 选项可用于以命令行选项的形式执行短程序，例如python –c "print('hello world')" 。

出现在程序名称或连字符（– ）之后的命令行选项将通过sys.argv 传递给程序，如9.1节所述。

另外，解释器还可读取表10-2中列出的环境变量。

表**10-2** 解释器环境变量

环境变量	描　　述
PYTHONPATH	以冒号分隔的模块搜索路径
PYTHONSTARTUP	在以交互方式启动时执行的文件
PYTHONHOME	Python安装的位置
PYTHONINSPECT	相当于-i 选项
PYTHONUNBUFFERED	相当于-u 选项
PYTHONIOENCODING	针对stdin 、 stdout 和stderr 的编码和错误处理。这是一个 *encoding* [:*errors*] 形式的字符串，如utf-8 或utf-8:ignore
PYTHONDONTWRITEBYTECODE	相当于-B 选项
PYTHONOPTIMIZE	相当于-O 选项
PYTHONNOUSERSITE	相当于-s 选项
PYTHONVERBOSE	相当于-v 选项
PYTHONUSERBASE	每个用户站点包的根目录
PYTHONCASEOK	指示对导入所使用的模块名称使用不区分大小写的匹配方式

PYTHONPATH 指定插入到sys.path 开头的模块搜索路径，已在第9章介绍。PYTHONSTARTUP 指定解释器在交互模式下运行时将执行的文件。PYTHONHOME 变量用于设置Python安装的位置，但很少需要该变量，因为Python知道如何找到自己的库和安装了扩展的site-packages目录。如果给定了一个目录，如/usr/local ，解释器期望能够在该位置找到所有文件。如果给定了两个目录，如usr/local:/usr/local/ sparc-solaris-2.6，解释器将在第一个目录中搜索与平台无关的文件，在第二个目录中搜索与平台有关的文件。如果在指定位置没有有效的Python安装，PYTHONHOME 将无效。

Python 3用户可能对PYTHONIOENCODING 环境设置感兴趣，因为它可以同时设置标准I/O流的编码和错误处理。这些设置很重要，因为Python 3在运行交互式解释器提示符时会直接输出Unicode。在检查数据时，这可能会导致意外的异常。例如：

```
>>> a = 'Jalape\xf1o'


>>> a


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/tmp/lib/python3.0/io.py", line 1486, in write
    b = encoder.encode(s)
  File "/tmp/lib/python3.0/encodings/ascii.py", line 22, in encode
    return codecs.ascii_encode(input, self.errors)[0]
UnicodeEncodeError: 'ascii' codec can't encode character '\xf1' in position 7:
ordinal not in range(128)
>>>
```

要解决该问题，可以将环境变量PYTHONIOENCODING 设为'ascii:backslashreplace' 或'utf-8' 等。现在，将得到以下结果：

```
>>> a = 'Jalape\xf1o'


>>> a


'Jalape\xf1o'
>>>
```

在Windows上，一些环境变量（如PYTHONPATH ）还可以从HKEY_LOCAL_MACHINE/Software/Python 中的注册表项读取。

10.2 交互式会话

如果没有给定程序名称，并且解释器的标准输入为一个交互式终端，Python将在交互模式下启动。在这种模式下，将打印标题消息并为用户提供一个提示符。另外，解释器还会执行PYTHONSTARTUP 环境变量（如果已设置）中包含的脚本。该脚本将作为输入程序的一部分执行（也就是说，无需使用import 语句加载它）。该脚本的一项应用就是读取用户配置文件，如.pythonrc 。

接受交互式输入时，将出现两个用户提示符。>>>提示符出现在一条新语句的开头。… 提示符表示一条语句还未结束。例如：

```
>>> for i in range(0,4):


...     print i,


...
```

```
0 1 2 3
>>>
```

在自定义应用程序中，可以修改sys.ps1 和sys.ps2 的值来更改提示符。

在一些系统中，可以将Python编译为使用GNU Readline库。如果启用了该库，它将为Python的交互模式提供命令历史、命令自动完成和其他功能。

默认情况下，在交互模式下发出的命令输出是通过在结果上打印内置repr() 函数的输出来生成的。通过将变量sys.displayhook 设为可显示结果的函数，可以更改这一设置。在下面的例子中，较长的结果被截断了：

```
>>> def my_display(x):

...     r = repr(x)

...     if len(r) > 40: print(r[:40]+"..."+r[-1])

...     else: print(r)

>>> sys.displayhook = my_display

>>> 3+4

7
>>> range(100000)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1...]
>>>
```

最后，在交互模式下，了解最后一次运算的结果存储在特殊变量（_ ）中这一点很有用。该变量可用于检索在后续操作中需要使用的结果，例如：

```
>>> 7 + 3

10
>>> _ + 2

12
>>>
```

变量的设置可以在前面介绍的displayhook() 函数中完成。如果重新定义了displayhook()，并且希望保留此功能，替代函数也应该设置。

10.3 启动Python应用程序

在大多数情况下，都希望程序自动启动解释器，而不是手动启动。在UNIX上，这是通过授予程序执行权限并将程序的第一行设为以下形式来实现的：

```
#!/usr/bin/env python
# Python代码从此位置开始...
print "Hello world"
...
```

在Windows上，双击.py 、.pyw 、.wpy 、.pyc 或.pyo 文件将自动启动解释器。通常，程序在控制台窗口中运行，除非使用.pyw 后缀对它们进行了重命名（在这种情况下，程序将静默运行）。如果需要为解释器提供选项，Python也可以从一个.bat 文件启动。例如，下面这个.bat 文件在一个脚本上运行Python，将命令提示符上提供的所有选项传递给解释器：

```
:: foo.bat
:: Runs foo.py script and passes supplied command line options along (if any)
c:\python26\python.exe c:\pythonscripts\foo.py %*
```

10.4 站点配置文件

典型的Python安装可能包含一些第三方模块和包。要配置这些包，解释器首先导入模块site 。site 的任务是搜索包文件并将更多的目录添加到模块搜索路径sys.path 中。另外，site 模块还会设置Unicode字符串转换所用的默认编码。

site 模块的工作方式是，首先创建一个由sys.prefix 和sys.exec_prefix 的值构成的目录名称列表，如下所示：

```
[ sys.prefix,           # 仅适用于Windows
sys.exec_prefix,        # 仅适用于Windows
sys.prefix + 'lib/pythonvers

/site-packages',
sys.prefix + 'lib/site-python',
sys.exec_prefix + 'lib/pythonvers/site-packages',
sys.exec_prefix + 'lib/site-python' ]
```

另外，如果启用了site 模块，会将用户的特定站点包目录添加到该列表中（将在下一节介绍）。

对于列表中的每个目录，都会检查该目录是否存在。如果存在，则将该目录添加

到sys.path 变量中。接下来检查目录是否包含任何路径配置文件（具有.pth 后缀的文件）。路径配置文件包含目录、zip文件或.egg 文件的列表，该列表与应该添加到sys.path 的路径文件位置相对应。例如：

```
# foo

包配置文件'foo.pth'


foo
bar
```

路径配置文件中的每个目录都必须在单独一行列出。注释和空行将被忽略。site 模块加载该文件时，它会检查每个目录是否存在。如果存在，则将该目录添加到sys.path 。重复的项目仅添加一次。

将所有路径添加到sys.path 之后，就会尝试导入sitecustomize 模块。该模块用于执行任何附加的（和任意的）站点自定义设置。如果导入sitecustomize 失败并抛出ImportError 错误，错误将被忽略。在导入sitecustomize 之后才会将其他用户目录添加到sys.path 。因此，将该文件放在你自己的目录下没什么用。

site 模块还负责设置默认Unicode编码。默认情况下，编码设置为'ascii' 。但是，可以更改编码，只需将sitecustomize.py 中调用sys.setdefaultencoding() 的代码替换为新编码，如'utf-8' 。如果你想试验一下，还可以修改site 的源代码，以根据机器的地区设置自动设置编码。

10.5 用户站点包

正常情况下，已安装的第三方模块可供所有用户访问。但是，每个用户可以在自己的站点目录中安装模块和包。在UNIX和Macintosh系统上，该目录可以在~/.local 下找到，其名称类似于~/.local/lib/python2.6/site-packages 。在Windows系统上，该目录由%APPDATA% 环境变量决定，通常类似于C:\Documents and Settings\David Beazley\Application Data 。在这个文件夹内，可以找到Python\Python26\site-packages 目录。

如果希望编写的Python模块和包能在一个库中使用，可以将它们放在用户自己的站点目录中。如果安装第三方模块，可以为setup.py 提供--user 选项来将它们手动安装在该目录下。例如：python setup.py install –user 。

10.6 启用新功能

会影响与旧Python版本兼容性的新语言特性首次出现在某个版本中时，通常会被禁用。要启用这些功能，可以使用语句from__future__import *feature* ，例如：

```
# 启用新的除法语义
from __future__ import division
```

使用该语句时，应该将它放在模块或程序的最前面。而且__future__ 导入的作用域仅限于使用它的模块。因此，导入新增功能不会影响到Python库模块的行为，也不会影响到需要解释器以前行为才能正常运行的旧代码。

目前，已定义的功能如表10-3所示。

表**10-3** **__future__** 模块中的功能名称

功能名称	描　　述
nested_scopes	支持函数中的嵌套作用域。在Python 2.1中首次引入，是Python 2.2中的默认行为
generators	支持生成器。在Python 2.2中首次引入，是Python 2.3中的默认行为
division	修改了除法语义，整数除法将返回小数结果。例如，1/4的结果是0.25，而不是0。在Python 2.2中首次引入，在Python 2.6中仍是可选功能，是Python 3.0中的默认行为
absolute_import	修改了与包相关的导入行为。目前，当包的子模块执行导入语句时（如导入字符串），它首先在包的当前目录查找，然后在sys.path 的目录中查找。但是，如果包恰好使用了有冲突的名称，将无法加载标准库中的模块。启用该功能后，语句导入模块执行绝对导入。因此，import string 这样的语句将始终从标准库加载字符串模块。在Python 2.5中首次引入，在Python 2.6中仍然被禁用。在Python 3.0中被启用
with_statement	支持上下文管理器和with 语句。在Python 2.5中首次引入，在Python 2.6中作为默认行为启用
print_function	使用Python 3.0 print() 函数代替print 语句。在Python 2.6中首次引入，在Python 3.0中作为默认行为启用

注意，__future_ 中目前没有删除任何功能名称。因此，即使在后续Python版本中默认启用了一项功能，也不会破坏使用该功能名称的现有代码。

10.7 程序终止

输入程序中没有可执行的其他语句时，出现未捕获的SystemExit 异常（由sys.exit() 生成）时，或者解释器收到SIGTERM 或SIGHUP 信号（在UNIX上）时，程序将会终止。程序退出时，解释器减小当前已知命名空间中所有对象的引用计数（同时销毁每个命名空间）。如果一个对象的引用计数达到0，将销毁该对象并调用其__del__() 方法。

应该注意，在某些情况下程序终止时可能不会调用__del__() 。如果对象之间存在循环引用（对象已分配，但不能从已知的命名空间访问），就可能发生这种现象。尽管

Python的垃圾回收器可以收回在执行期间未使用的循环引用，但通常不会在程序终止时调用它。

由于无法保证在程序终止时调用__del__()，所以显式清除某些对象（如已打开的文件和网络连接）是一种不错的做法。要完成该工作，可以向用户定义对象添加专门的清除方法（如close()）。另一种方法是，编写一个终止函数并将其注册到atexit 模块，如下所示：

```
import atexit
connection = open_connection("deaddot.com")

def cleanup():
print "Going away..."
close_connection(connection)

atexit.register(cleanup)
```

也可以通过以下方式调用垃圾回收器：

```
import atexit, gc
atexit.register(gc.collect)
```

有关程序终止还有一个值得注意的地方是，某些对象的__del__ 方法可能会尝试访问全局数据或在其他模块中定义的方法。由于这些对象可能已被销毁，所以__del__ 中将发生NameError 异常，出现如下错误提示：

```
Exception exceptions.NameError: 'c' in <method Bar.__del__
of Bar instance at c0310> ignored
```

如果出现这种情况，意味着__del__ 已过早地终止了。也就是说，尝试执行一个重要操作（如完全断开服务器连接）时失败。如果遇到这种情况，一种不错的做法是，在代码中执行显式关闭步骤，而不是依靠解释器在程序终止时完全销毁对象。通过在__del__() 方法的声明中声明默认参数，也可以消除奇怪的NameError 异常：

```
import foo
class Bar(object):
   def __del__(self, foo=foo):
       foo.bar()       # 使用模块foo中的某些内容
```

在某些情况下，可以在不执行任何清除操作的情况下终止程序执行，这种方法也很有用。这可以通过调用os._exit(*status*) 来完成。该函数提供了针对底层exit() 系统调用的接口，exit() 负责终止Python解释器进程。调用该函数时，程序将立即终止，而且不会执行任何进一步的处理或清除操作。

第11章　测试、调试、探查与调优

与使用C或Java等语言编写的程序不同，Python程序不是由生成可执行程序的编译器处理的。在C或Java程序中，编译器是挡住编程错误的第一道防线，它能够捕获很多错误，如使用数目不对的参数调用函数，或者将不恰当的值赋给变量（进行类型检查）。但是在Python中，只有在程序运行之后才会执行这类检查。因此，只有在运行和测试程序时，才会真正知道它是否能够正常运行。不仅如此，总是有一些错误会被隐藏起来并等待机会出现，除非你在运行程序时能够执行其内部控制流的每一个可能分支。（幸运的是，这种情况通常在程序发布几天后 才会出现）。

为了解决这类问题，本章介绍用于测试、调试和探查Python代码的技术和库模块。在本章的最后，还将介绍一些优化Python代码的策略。

11.1　文档字符串和doctest 模块

如果函数、类或模块的第一行是一个字符串，这个字符串就是文档字符串 。包含文档字符串被认为是一种良好的编程风格，因为这些字符串可用于向Python软件开发工具提供信息。例如，help() 命令检查文档字符串，Python IDE也会检查这些字符串。由于程序员倾向于在交互式shell 中进行试验时查看文档字符串，所以这些字符串中通常会包含简短的交互式例子。例如：

```
# splitter.py
def split(line, types=None, delimiter=None):
    """Splits a line of text and optionally performs type conversion.
       For example:

       >>> split('GOOG 100 490.50')
       ['GOOG', '100', '490.50']
       >>> split('GOOG 100 490.50',[str, int, float])
       ['GOOG', 100, 490.5]
       >>>
       By default, splitting is performed on whitespace, but a different
       delimiter can be selected with the delimiter keyword argument:

       >>> split('GOOG,100,490.50',delimiter=',')
       ['GOOG', '100', '490.50']
       >>>
    """
    fields = line.split(delimiter)
    if types:
        fields = [ ty(val) for ty,val in zip(types,fields) ]
    return fields
```

编写文档时需要注意的一个常见问题是，保持文档与函数的实际实现同步。例如，程序员可能在修改函数后忘记更新文档。

可以使用doctest 模块解决该问题。doctest 模块会收集文档字符串，在其中查找

交互式会话，并且以一系列测试的形式执行它们。要使用doctest，通常会创建一个独立模块用于测试。例如，如果上面的函数位于文件splitter.py中，可以创建一个文件testsplitter.py来进行测试，如下所示：

```
# testsplitter.py
import splitter
import doctest

nfail, ntests = doctest.testmod(splitter)
```

在这段代码中，调用doctest.testmod(*module*)的目的是在指定模块上运行测试，返回测试失败的数量和执行的测试总数。如果所有测试都顺利通过，则不会生成输出。否则，将会生成一个错误报告，指出预期输出与接收到的输出之间的区别。如果希望查看测试的详细输出，可以使用testmod (*module*, verbose=True)。

除了创建独立测试文件外，库模块也可以通过在文件末尾包含以下代码来测试自身：

```
...
if __name__ == '__main__':
    # 测试自身
    import doctest
    doctest.testmod()
```

添加这段代码后，如果文件作为主程序在解释器中运行，就会运行文档测试。否则，如果文件是由import加载的，测试将被忽略。

doctest要求函数输出与从交互式解释器得到的输出完全一致。所以，需要特别注意空格和数字精度的问题。例如，考虑下面这个函数：

```
def half(x):
    """Halves x. For example:

    >>> half(6.8)
    3.4
    >>>
    """
    return x/2
```

如果在该函数上运行doctest，将得到一个类似于以下形式的错误报告：

```
**********************************************************************
File "half.py", line 4, in __main__.half
Failed example:
    half(6.8)
Expected:
    3.4
Got:
```

```
    3.3999999999999999
**********************************************************************
```

要解决该问题，需要让文档与输出完全一致，或者在文档中挑选一个更好的示例。

由于doctest 的使用非常简单，所以几乎没有理由不在程序中使用它。但是请记住，doctest 不是一个可执行全面程序测试的模块。使用它执行全面测试将导致很长且复杂的文档字符串，这与生成有用文档的目标不一致。例如，在用户寻求帮助时，如果文档中列出了涵盖所有细节的50个例子，这会让人很生气。对于这类测试，最好使用unittest 模块。

最后，doctest 模块拥有大量配置选项，用于控制测试执行和报告结果的各个方面。对于该模块最常见的用法而言，这些选项都不是必需的，所以这里不再赘述。请访问 http://docs.python.org/library/doctest.html ，了解更多细节。

11.2 单元测试和unittest 模块

对于更全面的程序测试，可以使用unittest 模块。如果进行单元测试，开发人员会为程序的每个组成元素（如各个函数、方法、类和模块）编写独立的测试案例。然后运行这些测试来验证组成更大程序的基本组件的行为是否正确。随着程序规模不断变大，可以组合针对各种组件的单元测试，创建大型的测试框架和测试工具。这可以极大地简化验证正确行为以及隔离和修复问题等任务。该模块的使用方法可以通过上一节中列出的代码进行说明：

```
# splitter.py
def split(line, types=None, delimiter=None):
    """Splits a line of text and optionally performs type conversion.
       ...
    """
    fields = line.split(delimiter)
    if types:
        fields = [ ty(val) for ty,val in zip(types,fields) ]
    return fields
```

如果需要编写单元测试来测试split() 函数的各个方面，可以创建一个独立模块testsplitter. py ，如下所示：

```
# testsplitter.py
import splitter
import unittest

# 单元测试
class TestSplitFunction(unittest.TestCase):
      def setUp(self):
          # 执行设置操作（如果有的话）
          pass
      def tearDown(self):
```

```
        # 执行清除操作（如果有的话）
        pass
    def testsimplestring(self):
        r = splitter.split('GOOG 100 490.50')
        self.assertEqual(r,['GOOG','100','490.50'])
    def testtypeconvert(self):
        r = splitter.split('GOOG 100 490.50',[str, int, float])
        self.assertEqual(r,['GOOG', 100, 490.5])
    def testdelimiter(self):
        r = splitter.split('GOOG,100,490.50',delimiter=',')
        self.assertEqual(r,['GOOG','100','490.50'])
# 运行unittest
if __name__ == '__main__':
   unittest.main()
```

要运行单元测试，只需在文件testsplitter.py 上运行Python。下面给出了一个例子：

```
% python testsplitter.py

...

----------------------------------------------------------------------
Ran 3 tests in 0.014s

OK
```

unittest 的基本使用包括定义一个继承自unittest.TestCase 的类。在这个类中，各种测试由以名称test 开头的方法定义，如testsimplestring 、testtypeconvert 等。（需要强调的是，可以随意命名，只要它以test 开头即可。）在每个测试内，使用各种断言来检查不同条件。

例如，在编写程序和控制测试过程时，使用unittest.TestCase 的实例t 的以下方法。

```
t

.setUp()
```

在运行任何测试方法之前，调用它来执行设置步骤。

```
t
.tearDown()
```

在运行测试之后，调用它来执行清除操作。

```
t
.assert_(expr
 [, msg
])

t
.failUnless(expr
 [, msg
])
```

如果expr 的计算结果为False，表明测试失败。 *msg* 是一条消息字符串，提供对失败（如果有的话）的解释。

```
t
.assertEqual(x
, y
 [,msg
])

t
.failUnlessEqual(x
```

```
, y
 [, msg
])
```

如果x 和 *y* 不相等，则表明测试失败。 *msg* 是一条解释失败（如果有的话）的消息。

```
t
.assertNotEqual(x
, y
 [, msg
])

t
.failIfEqual(x
, y
, [, msg
])
```

如果 *x* 和 *y* 相等，则表明测试失败。 *msg* 是一条解释失败（如果有的话）的消息。

```
t
.assertAlmostEqual(x
, y
 [, places
 [, msg
]])
```

```
t
.failUnlessAlmostEqual(x
, y
, [, places
 [, msg
]])
```

如果数字x和y未包含在对方的 *places* 小数位数中，则表明测试失败。检查方法是计算 *x* 和 *y* 的差并将结果舍入到给定位数。如果结果为0，则 *x* 和 *y* 的值相近。 *msg* 是一条解释失败（如果有的话）的消息。

```
t
.assertNotAlmostEqual(x
, y
, [, places
 [, msg
]])

t
.failIfAlmostEqual(x
, y
 [, places
 [, msg
]])
```

如果 *x* 和 *y* 在 *places* 小数位数内无法区分大小，则表明测试失败。 *msg* 是一条解释失败（如果有的话）的消息。

```
t
.assertRaises(exc
, callable
, ...)

t
.failUnlessRaises(exc
, callable
, ...)
```

如果可调用对象 *callable* 未引发异常 *exc* ， 则表明测试失败。剩余参数将以参数形式传递给 *callable* 。可以使用异常元组 *exc* 检查多个异常。

```
t
.failIf(expr
 [, msg
])
```

如果 *expr* 计算结果为True ， 则表明测试失败。 *msg* 是一条解释失败（如果有的话）的消息。

```
t
.fail([msg
])
```

表明测试失败。 *msg* 是一条解释失败（如果有的话）的消息。

```
t
.failureException
```

该属性设置为在测试中捕获到的最后一个异常值。如果不仅要检查是否出现异常，还想检查异常是否抛出了恰当的值（例如，如果想检查引发的异常中生成的错误消息），该属性可能很有用。

应该注意，unittest 模块中包含大量高级自定义选项，主要针对测试进行分组、创建测试套件和控制测试的运行环境。这些功能与为代码编写测试的过程没有直接关系（编写测试类的过程往往与实际执行测试的方法彼此独立）。关于如何组织大型程序的测试的更多信息，请查阅http://docs.python.org/library/unittest.html 上的文档。

11.3 Python调试器和pdb 模块

Python提供了一个基于命令的简单调试器，可以在pdb 模块中找到它。pdb 模块支持事后检查（post mortem），检查栈帧，设置断点，单步调试源代码，以及计算代码。

可以通过很多函数从程序或交互式Python shell中调用该调试器。

```
run(statement
 [, globals
 [, locals
]])
```

在调试器控制下执行字符串 *statement* 。在任何代码执行之前，将出现调试器提示符。键入continue 可以强制运行调试器。 *globals* 和 *locals* 分别定义运行代码的全局和局部命名空间。

```
runeval(expression
 [, globals
 [, locals
```

```
]])
```

在调试器控制下计算 *expression* 字符串。在任何代码执行之前，将出现调试器提示符，所以需要键入continue 强制运行调试器，就像使用run() 一样。成功运行之后，将返回表达式的值。

```
runcall(function
 [, argument
, ...])
```

在调试器内调用一个函数。 *function* 是一个可调用对象。其他参数以参数的形式提供给 *function* 。在任何代码执行之前，将出现调试器提示符。在函数运行完成后将返回它的返回值。

```
set_trace()
```

在调用该函数的位置启动调试器。这可用于将调试器断点硬编码到特定代码位置。

```
post_mortem(traceback
)
```

对回溯对象启动事后检查。 *traceback* 通常使用sys.exc_info() 等函数获得。

```
pm()
```

使用最后一个异常的回溯进入事后检查调试。

在用于启动调试器的所有函数中，set_trace() 函数在实践中可能是使用起来最简单的。如果正在处理复杂的应用程序，但在它的某个部分中检测到了一个问题，可以将set_trace() 调用插入到代码中并运行该应用程序即可。这样，在遇到问题时，程序将停止运行并转到调试器，你可以在调试器中检查执行环境。执行过程会在离开调试器之后继续。

11.3.1 调试器命令

调试器启动时，它会显示一个(Pdb) 提示符，例如：

```
>>> import pdb

>>> import buggymodule

>>> pdb.run('buggymodule.start()')

> <string>(0)?()
(Pdb)
```

可以在调试器提示符(Pdb) 下识别以下命令。注意，一些命令具有长短两种形式。在这种情况下，使用括号来表示两种形式。例如，h(elp) 表示h 和help 都是可接受的。

```
[!

]statement
```

在当前栈帧上下文中执行（一行） *statement* 。感叹号可以忽略，但如果语句的第一个词与某个调试器命令类似，则必须使用它来避免歧义。要设置全局变量，可以在同一行上为赋值命令添加global 命令作为前缀：

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)

a(rgs

)
```

打印当前函数的参数列表。

```
alias

[name

[command

]]
```

创建名为 *name* 的别名来执行 *command* 。在 *command* 字符串中，键入别名时子字符串'%1' 、'%2' 等被替换为相应的参数。'%*' 被替换为所有参数。如果没有给定任何命令，则显示当前别名列表。别名可以嵌套使用，可以包含能够在Pdb 提示符下键入的任何有效值，例如：

```
# 打印实例变量（使用"pi classInst"）
alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k]
# 打印self中的实例变量
alias ps pi self

b(reak) [loc

 [, condition

]]
```

在位置 *loc* 处设置断点。 *loc* 指定一个特定文件名和行号，或者指定一个模块中的一个函数名称。可以使用以下语法。

设　　置	描　　述
n	当前文件中的行号
filename:n	另一个文件中的行号
function	当前模块中的函数名称
module.function	某个模块中的函数名称

如果省略了 *loc* ，将打印当前的所有断点。 *condition* 是一个表达式，在打印断点之前，该表达式的值必须计算为True 。所有断点都分配了一个数字，该数字将在完成此命令时作为输出打印出来。这些数字可以在以下多种调试器命令中使用。

```
cl(ear) [bpnumber
 [bpnumber ...
]]
```

清除断点编号列表。如果指定了断点，所有断点都会被清除。

```
commands [bpnumber
]
```

设置在遇到 *bpnumber* 断点时，将自动执行的一系列调试器命令。列出要执行的命令时，只需在后续行中键入它们并使用end 来标记命令序列的结束即可。如果包含continue 命令，在遇到断点时，程序将自动继续执行。如果省略了 *bpnumber* ，将使用最后一个断点集。

```
condition bpnumber
 [condition
]
```

在断点上放置一个条件。 *condition* 是一个表达式，在识别该断点之前，该表达式的值必须计算为True 。省略该条件会清除任何以前的条件。

```
c(ont(inue))
```

继续执行，直至遇到下一个断点。

```
disable [bpnumber
 [bpnumber
 ...]]
```

禁用指定断点集。与clear 不同，可以在以后重新启用这些断点。

```
d(own)
```

将当前帧在栈跟踪中下移一层。

```
enable [bpnumber
 [bpnumber ...
]]
```

启用指定的断点集。

```
h(elp) [command
]
```

显示可用命令的列表，指定一个命令将返回该命令的帮助信息。

```
ignore bpnumber
 [count
]
```

忽略一个断点，以便执行 *count* 。

```
j(ump) lineno
```

设置要执行的下一行。这只能用于在同一执行帧中的不同语句之间移动。而且，无法跳到某些语句中，如循环中的语句。

```
l(ist
) [first
 [, last
]]
```

列出源代码。如果没有参数，该命令将列出当前行前后的11行（该行以及前面和后面的5行）。如果使用一个参数，它将列出该行前后的11行。如果使用两个参数，它将列出指定范围内的行。如果 *last* 小于 *first* ，它将被解释为一个计数。

```
n(ext)
```

执行到当前函数中的下一行。跳过函数调用中包含的代码。

```
p expression
```

在当前上下文中计算表达式的值并打印其值。

```
pp expression
```

与p 命令相同，但使用适合打印的模块（pprint） 对结果进行格式化。

```
q(uit)
```

退出调试器。

```
r(eturn)
```

持续运行，直至当前函数返回值。

```
run [args

]
```

重新启动程序并使用 *args* 中的命令行参数作为sys.argv 的新设置。所有断点和其他调试器设置都会被保留。

```
s(tep)
```

执行一行源代码并停在被调用函数内。

```
tbreak [loc
```

```
 [, condition
]]
```

设置一个临时断点，该断点将在第一次到达时删除。

```
u(p)
```

将当前帧在栈跟踪中上移一层。

```
unalias name
```

删除指定别名。

```
until
```

恢复执行，直至不再控制当前执行帧，或者直至到达一个比当前行号大的行号。例如，如果调试器停止在循环体中的最后一行，键入`until`将执行循环中的所有语句，直至循环结束。

```
w(here)
```

打印栈跟踪。

11.3.2 从命令行进行调试

另一种运行调试器的方法是在命令行上调用它，例如：

```
% python -m pdb someprogram.py
```

在这种情况下，启动程序时调试器将自动启动，在这时可以随意设置断点并进行其他配置更改。要运行程序，只需使用continue 命令。例如，如果想从使用split() 函数的程序中调试该函数，可以输入以下命令：

```
% python –m pdb someprogram.py


> /Users/beazley/Code/someprogram.py(1)<module>()
-> import splitter
(Pdb) b splitter.split


Breakpoint 1 at /Users/beazley/Code/splitter.py:1
(Pdb) c


> /Users/beazley/Code/splitter.py(18)split()
-> fields = line.split(delimiter)
(Pdb)
```

11.3.3 配置调试器

如果用户的主目录或当前目录中包含.pdbrc 文件，那么将读取该文件并执行它，就好像在调试器提示符下键入该文件名一样。可以使用这种方法来指定要在调试器每次启动时执行的调试命令（与每次必须交互式地键入命令相反）。

11.4 程序探查

profile 和cProfile 模块用于收集探查信息。两个模块的工作方式相同，但cProfile 实现为C扩展，速度快得多且更加先进。两个模块都用于收集覆盖信息（也就是执行了哪些函数）以及性能统计信息。探查程序的最简单方式就是从命令行执行程序，如下所示：

```
% python -m cProfile someprogram.py
```

也可以使用profile 模块中的以下函数：

```
run(command [, filename

])
```

在探查器（profiler）中使用exec 语句执行 *command* 的内容。 *filename* 是保存原始探查数据的文件名称。如果忽略该参数，报告将输出到标准输出。

运行探查器之后将生成一个报告，如下所示：

```
126 function calls (6 primitive calls) in 5.130 CPU seconds
Ordered by: standard name
ncalls tottime percall cumtime percall filename:lineno(function)
     1   0.030   0.030   5.070   5.070 <string>:1(?)
 121/1   5.020   0.041   5.020   5.020 book.py:11(process)
     1   0.020   0.020   5.040   5.040 book.py:5(?)
     2   0.000   0.000   0.000   0.000 exceptions.py:101(__init__)
     1   0.060   0.060   5.130   5.130 profile:0(execfile('book.py'))
     0   0.000           0.000         profile:0(profiler)
```

run() 生成的报告各部分说明如下。

部　　分	说　　明
primitive calls	非递归性函数调用的数量
ncalls	调用总数（包括自递归）
tottime	该函数消耗的时间（不统计子函数）
percall	tottime/ncalls
cumtime	函数消耗的总时间
percall	cumtime/(primitive calls)
filename :*lineno* (*function*)	每个函数的位置和名称

第一列中有两个数字（如121/1 ）时，后一个数字表示原始调用的数量，前一个数字表示实际调用数量。

通常对于该模块的大部分应用，例如，只想查看程序将时间消耗在哪些地方，只需检查探查器生成的报告就足够了。但是，如果希望保存数据并进行进一步分析，可以使用`pstats` 模块。关于保存和分析探查数据的更多细节，请访问 http://docs.python.org/library/profile.html 。

11.5 调优与优化

本节介绍一些可以使Python程序运行速度更快和占用更少内存的常用经验规则。这里介绍的技术并不全面，但是程序员在审视自己的代码时应该会从中得到一些启发。

11.5.1 进行计时测量

如果只是想要对长时间运行的Python程序进行计时，最简单的方式通常是在UNIX `time` 等命令的控制下运行它。如果需要对一组长时间运行的语句进行计时，也可以插入对`time.clock()` 的调用来获取已用CPU时间的最新读数，或者插入对`time.time()` 的调用来读取最新的时钟时间。例如：

```
start_cpu = time.clock()
start_real= time.time()
statements


statements


end_cpu = time.clock()
end_real = time.time()
print("%f Real Seconds" % (end_real - start_real))
print("%f CPU seconds" % (end_cpu - start_cpu))
```

记住，只有当需计时的代码会运行很长时间时，这个技巧才真正有效。如果想要对一个特定语句进行基准测试，可以使用`timeit` 模块中的`timeit(code [, setup ])` 函数。例如：

```
>>> from timeit import timeit


>>> timeit('math.sqrt(2.0)','import math')


0.20388007164001465
>>> timeit('sqrt(2.0)','from math import sqrt')


0.14494490623474121
```

在本例中，`timeit()` 的第一个参数是希望对其进行基准测试的代码。第二个参数是

一条语句，这条语句会执行一次来设置执行环境。`timeit()` 函数会运行所提供的语句100万次并报告执行时间。可以向`timeit()` 提供`number=`*`count`* 关键字参数来更改重复次数。

`timeit` 模块还有一个函数`repeat()`，可用于测量。该函数的工作原理与`timeit()` 相同，但它重复计时测量3次并返回一个结果列表。例如：

```
>>> from timeit import repeat

>>> repeat('math.sqrt(2.0)','import math')

[0.20306601524353027, 0.19715800285339355, 0.20907392501831055]
>>>
```

进行性能测量时，常常会参考相关的加速 （speedup）数据，加速通常指的是原始执行时间与新执行时间的商。例如，在前面的计时测量中，使用`sqrt(2.0)` 代替`math.sqrt(2.0)` 表示的加速为0.20388/0.14494，即大约为1.41。

有时会以百分比的形式报告这个数字，如加速大约为41%。

11.5.2 进行内存测量

`sys` 模块有一个`getsizeof()` 函数，可用于分析各个Python对象的内存占用（以字节为单位）。例如：

```
>>> import sys

>>> sys.getsizeof(1)

14
>>> sys.getsizeof("Hello World")

52
>>> sys.getsizeof([1,2,3,4])

52
>>> sum(sys.getsizeof(x) for x in [1,2,3,4])

56
```

对于列表、元组和字典等容器，报告的大小只是容器对象本身的大小，不是容器中包含的所有对象的累计大小。例如，在前面的例子中，列表[1,2,3,4]的报告大小实际上小于4个整数所需的空间（每个整数为14字节）。这是因为列表的内容未包含在总数中。可以像上面显示的那样使用sum()来计算列表内容的总大小。

注意，getsizeof()函数只会提供各个对象粗略的总体内存使用量。在内部，解释器会通过引用计数频繁地共享对象，所以对象消耗的实际内存可能比想象中的要小很多。另外，由于Python的C扩展可以在解释器之外分配内存，所以可能很难精确测量总体内存使用情况。测量实际内存占用的一种辅助技术是，从操作系统的进程查看器或任务管理器检查正在运行的程序。

坦白地说，了解内存使用的一种更好方式可能是，坐下来对其进行仔细分析。如果知道程序将分配各种数据结构，而且知道哪些数据类型将存储在这些结构中（也就是整型、浮点型、字符串等），则可以使用getsizeof()函数的结果来计算程序的内存使用上限，或者至少能够获得足够的信息来进行粗略估算。

11.5.3 反汇编

dis模块可用于将Python函数、方法和类反汇编为低级的解释器指令。该模块定义了一个dis()函数，可以按以下方式使用它：

```
>>> from dis import dis

>>> dis(split)

  2           0 LOAD_FAST                0 (line)
              3 LOAD_ATTR                0 (split)
              6 LOAD_FAST                1 (delimiter)
              9 CALL_FUNCTION            1
             12 STORE_FAST               2 (fields)

  3          15 LOAD_GLOBAL              1 (types)
             18 JUMP_IF_FALSE           58 (to 79)
             21 POP_TOP

  4          22 BUILD_LIST               0
             25 DUP_TOP
             26 STORE_FAST               3 (_[1])
             29 LOAD_GLOBAL              2 (zip)
             32 LOAD_GLOBAL              1 (types)
             35 LOAD_FAST                2 (fields)
             38 CALL_FUNCTION            2
             41 GET_ITER
        >>   42 FOR_ITER                25 (to 70)
             45 UNPACK_SEQUENCE          2
             48 STORE_FAST               4 (ty)
             51 STORE_FAST               5 (val)
             54 LOAD_FAST                3 (_[1])
             57 LOAD_FAST                4 (ty)
             60 LOAD_FAST                5 (val)
             63 CALL_FUNCTION            1
```

```
             66 LIST_APPEND
             67 JUMP_ABSOLUTE         42
        >>   70 DELETE_FAST            3 (_[1])
             73 STORE_FAST             2 (fields)
             76 JUMP_FORWARD           1 (to 80)
        >>   79 POP_TOP

  5     >>   80 LOAD_FAST              2 (fields)
             83 RETURN_VALUE
>>>
```

专家级程序员可以采用两种方式来使用这些信息。首先，通过反汇编来准确显示执行函数过程中涉及的操作。通过仔细分析，可能会发现加速的机会。其次，如果正在编写线程，那么反汇编结果中打印的每一行表示一个解释器操作，每个操作采用原子执行方式。因此，如果尝试跟踪复杂的竞争条件（race condition），该信息可能很有用。

11.5.4 调优策略

下面列出了一些在作者看来已经被证明对Python代码有用的优化策略。

1．理解程序

进行任何优化之前，需要知道通过优化程序某个部分所获得的加速与该部分所占的执行时间直接相关。例如，如果优化一个函数，使其运行速度变为了原来的10倍，但该函数的执行时间仅占程序总执行时间的10%，那么将仅能获得9%～10%的总体加速。鉴于执行优化所涉及的工作，这种优化成果可能并不值得。

首先在要优化的代码上使用探查模块始终是一种不错的做法。你实际上只需要关注占用程序大部分执行时间的函数和方法，而不是偶尔调用的次要操作。

a．理解算法

即使是糟糕的$O(n \log n)$算法实现也会比经过最优化的$O(n^3)$算法性能要高。不要优化低效的算法，首先应寻找更好的算法。

b．使用内置类型

Python内置的元组、列表、集合和字典类型完全是用C实现的，而且是解释器中优化程度最高的数据结构。应该积极使用这些类型来存储和操作程序中的数据，尽量避免构建自定义数据结构（如二进制搜索树、链接列表等）来模仿它们的功能。

尽管这样说，你还是应该积极使用标准库中的类型。一些库模块提供的新类型在处理特定任务上比内置类型性能更高。例如，`collection.deque` 类型提供了与列表类似的功能，但针对在两端插入新项的操作进行了高度优化。相反，列表只有在末尾附加项时才具有较高的效率。如果在前端插入项，需要移动所有其他元素来腾出空间。执行这一操作所需的时间会随着列表的不断变大而增加。为了让你直观地了解两者的差异，下面对在列表和双端队列前端插入100万个项目的操作进行了计时测量：

```
>>> from timeit import timeit

>>> timeit('s.appendleft(37)',

...        'import collections; s = collections.deque()',

...        number=1000000)

0.24434304237365723
>>> timeit('s.insert(0,37)', 's = []', number=1000000)

612.95199513435364
```

2. 不要添加层

任何时候向对象或函数添加额外的抽象或便利层，都会降低程序的运行速度。但是，也需要在可用性与性能之间进行权衡。例如，添加额外层的总体目标通常是为了简化编码，这也有好处。

我们举一个简单的例子，设想一个程序使用`dict()`函数来创建具有字符串键的字典，如下所示：

```
s = dict(name='GOOG',shares=100,price=490.10)
# s = {'name':'GOOG', 'shares':100, 'price':490.10 }
```

程序员可能以这种方式创建字典来节省键入操作（无需在键名称两边添加引号）。但是，这种创建字典的替代方法运行速度非常慢，因为它添加了额外的函数调用。

```
>>> timeit("s = {'name':'GOOG','shares':100,'price':490.10}")

0.38917303085327148
>>> timeit("s = dict(name='GOOG',shares=100,price=490.10)")

0.94420003890991211
```

如果程序在运行时创建了数百万个字典，那么你应该明白第一种方法更快。在绝大部分情况下，添加增强或更改现有Python对象工作方式的任何功能在运行速度上都比较慢。

3. 了解如何基于字典构建类和实例

用户定义的类和实例是用字典构建的。因此，查找、设置或删除实例数据的速度几乎总是比直接在字典上执行这些操作更慢。如果要做的只是构建一个简单的数据结构来存储数据，字典可能是比定义一个类更有效的选择。

为了演示两者间的差异，这里给出了一个简单的类来表示所持有的股票：

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```

如果比较使用该类与使用字典的性能，结果会非常有趣。首先，让我们比较一下创建实例的性能：

```
>>> from timeit import timeit

>>> timeit("s = Stock('GOOG',100,490.10)","from stock import Stock")

1.3166780471801758
>>> timeit("s = {'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 }")

0.37812089920043945
>>>
```

此处创建新对象的加速大约为3.5。接下来，让我们看一下执行简单计算的性能：

```
>>>
 timeit("s.shares*s.price",

...       "from stock import Stock; s = Stock('GOOG',100,490.10)")

0.29100513458251953
>>> timeit("s['shares']*s['price']",

...       "s = {'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 }")

0.23622798919677734
>>>
```

此处加速大约为1.2。这个示例告诉我们，尽管可以使用class 定义新对象，但这并不是处理数据的唯一方式。元组和字典通常就够用了。使用它们会使程序运行更快并占用更少的内存。

4. 使用__slots__

如果程序创建了用户定义类的大量实例，可以考虑在类定义中使用__slots__ 属性。例如：

```
class Stock(object):
    __slots__ = ['name','shares','price']
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```

__slots__ 有时被看作一种安全功能，因为它会限制属性名称的设置。但是，它更主要的用途是性能优化。使用__slots__ 的类不使用字典存储实例数据（而是用一种更高效的内部数据结构）。所以，不仅实例使用的内存少得多，而且访问实例数据的效率要更高。在某些情况下，仅仅添加__slots__ 而不进行其他任何更改就会使程序的运行速度显著提高。

但是，使用__slots__ 时有一个地方要注意。将该功能添加到类中可能会无故破坏其他代码。例如，众所周知，实例将它们的数据存储在可作为__dict__ 属性访问的字典中。定义slots 时，该属性还不存在，所以依赖__dict__ 的任何代码都会失败。

5. 避免使用（.）运算符

使用（.）在对象上查找属性时，总是会涉及名称查找。例如，如果使用x.name ，将会在环境中查找变量名称x ，然后在x 上查找属性name 。对于用户定义对象，属性查找还可能涉及在实例字典、类字典和基类的字典中查找。

对于大量使用方法或模块查找的计算，最好首先将要执行的操作放到一个局部变量中，从而避免属性查找。例如，如果执行大量求平方根操作，使用from math import sqrt 和sqrt(x) 比键入math.sqrt(x) 更快。在11.5.1节中，我们看到了这种方法能够带来大约1.4的加速。

显然，不应该在程序的所有位置消除属性查找，因为这会使代码非常难以理解。但是，对于注重性能的部分，这是一种有用的技巧。

6. 使用异常来处理不常见的情况

为了避免错误，你可能倾向于向程序中添加额外的检查。例如：

```
def parse_header(line):
    fields = line.split(":")
    if len(fields) != 2:
        raise RuntimeError("Malformed header")
```

```
    header, value = fields
    return header.lower(), value.strip()
```

但是，还可以采用另一种方法来处理错误，那就是让程序生成异常并捕获它。例如：

```
def parse_header(line):
    fields = line.split(":")
    try:
        header, value = fields
        return header.lower(), value.strip()
    except ValueError:
        raise RuntimeError("Malformed header")
```

如果在格式正确的行上对两个版本的代码进行基准测试，会发现第二版的运行速度快10%左右。为在正常情况下不会抛出异常的代码设置`try`代码块，这通常比执行`if`语句的速度更快。

7．避免对常见情况使用异常

不要在代码中对常见情况进行异常处理。例如，假设一个程序执行了大量字典查找，但大部分查找操作都是在查找不存在的键。现在，可以考虑两种执行查找的方法：

```
# 方法1：执行查找并捕获异常
try:
    value = items[key]
except KeyError:
    value = None

# 方法2：检查键是否存在，然后执行查找
if key in items:
   value = items[key]
else:
   value = None
```

在简单的性能测试中（假设没有找到所需的键），第二种方法的运行速度是第一种方法的17倍！如果你还不相信的话，第二种方法的运行速度差不多是用`items.get(key)`的两倍，因为`in`运算符的执行速度比方法调用更快。

8．鼓励使用函数式编程和迭代

列表推导、生成器表达式、生成器、协程和闭包比大多数Python程序员想象得要更为高效。尤其对于数据处理而言，与手动迭代数据并执行类似操作的代码相比，列表推导和生成器表达式的运行速度要快得多。这些操作的运行速度也比使用`map()`和`filter()`等函数的旧式Python代码快得多。使用生成器编写的代码不仅运行速度快，而且内存使用效率也高。

9．使用装饰器和元类

装饰器和元类用于修改函数和类。但是，由于它们在定义函数或类时进行操作，所以可以通过多种方式使用它们来改进性能，特别是程序拥有很多可以启用或禁用的可选功能时。第6章给出了使用装饰器启用函数日志记录功能的例子，但它所采用的方式在禁用日志记录时也不会影响性能。

第二部分　Python库

本部分内容

第12章　内置函数和异常

本章介绍Python的内置函数和异常。在本书前面的各章中也有涉及这部分内容，但不够正式。本章只是将所有这些信息合并在起，并进一步说明某些函数比较微妙的功能。此外，Python 2中包含很多已过时并且在Python 3中已去掉的内置函数。本章不会介绍这些函数，而是重点介绍现在常用的函数。

12.1　内置函数和类型

某些类型、函数和变量对解释器来说总是可用的，并且可用在任意源模块中。尽管访问这些函数不必执行任何额外的导入工作，但是它们被放置在了Python 2的`__builtin__`模块和Python 3的`builtins` 模块中。在导入的其他模块内，也将变量`__builtins__` 绑定到该模块。

```
abs(x
)
```

返回 *x* 的绝对值。

```
all(s
)
```

如果可迭代的 *s* 中的所有值都为`True`，则返回`True` 。

```
any(s
)
```

如果可迭代的 *s* 中的任意值为`True`，则返回`True` 。

```
```

```
ascii(x
)
```

就像repr() 那样创建对象 *x* 的一种可打印形式，但是在结果中只使用ASCII字符。非ASCII字符都转换为合适的转义序列。用于在不支持Unicode的终端或shell程序中查看Unicode字符串。只适用于Python 3。

```
basestring
```

这是一个抽象数据类型，它是Python 2中所有字符串（str 和unicode ）的超类。它只用于字符串的类型测试。例如，如果 *s* 是两种字符串类型之一，则isinstance(*s*,basestring) 返回True 。只适用于Python 2。

```
bin(x
)
```

以字符串的形式返回整数 *x* 的二进制表示 。

```
bool([x
])
```

表示布尔值True 和False 的类型。如果用于转换 *x* ，那么若根据常见的真值测试语义（即非零数值、非空列表等） *x* 等于真，则返回True ；否则返回False 。如果不带任何参数调用bool() ，则返回的默认值也是False 。这个bool 类继承自int ，因此在数学计算中，布尔值True 和False 可以用作整数，代表值1和0。

```
bytearray([x
])
```

表示可变字节数组的类型。创建一个实例时， *x* 可能是范围从0到255的可迭代整数序列、8位字符串或字节字面量，或者是指定字节数组大小的整数（在这种情况下，每一项都将初始化为0）。bytearray 对象 *a* 看起来就像一个整数数组。如果执行诸如 *a* [*i*] 这样的查找操作，就会得到一个整数值，代表了索引i 处的字节值。如 *a* [*i*]=*v* 这样的赋值也要求v 是一个整数字节值。但是，bytearray 也提供了通常与字符串有关的所有操作（即切片、find() 、split() 、replace() 等）。使用这些字符串操作时，应当注意在所有字符串字面量前加上一个b ，表明处理的是字节。例如，如果要使用逗号分隔符将字节数组 *a* 拆成几个字段，应该使用 *a* .split(b',')， 而不是 *a* .split(',') 。这些操作的结果总是新的bytearray 对象，而不是字符串。要将一个bytearray a 转换成字符串，可使用 *a* .decode(*encoding*) 方法。encoding 设为'latin-1' ， 将直接把8位字符的bytearray 转换成字符串，而无需对基本字符值进行任何修改。

```
bytearray(s
 ,encoding
)
```

以上是另一种根据字符串 *s* 中的字符创建bytearray 实例的方法，其中 *encoding* 指定在转换中要使用的字符编码。

```
bytes([x
])
```

表示不变字节数组的类型。在Python 2中，这是str() 的别名，用于创建一个标准8位字符串。在Python 3中，bytes 是一个完全独立的类型，它是前面描述的bytearray 类型的一个不可变版本。在这种情况下，参数 *x* 具有相同的含义，并且可以按照相同的方式使用。可移植性方面需要注意：在Python 2中尽管定义了bytes ，但最终对象的行为也与Python 3中的不一致。例如，如果 *a* 是由bytes() 创建的实例，那么在Python 2中 *a* [*i*] 返回一个字符串，而在Python 3中返回一个整数。

```
bytes(s
, encoding
)
```

这是另一种根据字符串 *s* 中的字符创建bytes 实例的方法，其中 *encoding* 指定要使用的字符编码。只适用于Python 3。

```
chr(x)
```

将整数值 *x* 转换为单字符的字符串。在Python 2中， *x* 必须在`0 <= x <= 255` 范围内，而在Python 3中， *x* 必须表示有效的Unicode代码点。如果 *x* 超出范围，则引发 `ValueError` 异常。

```
classmethod(func
)
```

该函数创建 *func* 函数的类方法。通常只在类定义内部使用它，通过`@classmethod` 装饰器来隐式调用它。与正常的方法（method）不同，类方法将类而不是实例作为第一个参数。例如，如果有一个对象`f`，它是`Foo` 类的一个实例，对`f` 调用类方法会将`Foo` 类作为第一个参数传递到该方法，而不是实例`f` 。

```
cmp(x
, y
)
```

比较 *x* 和 *y*，如果 `x < y` 则返回一个负数，如果 `x > y` 则返回一个正数，如果 `x == y` 则返回`0` 。可以比较任意两个对象，如果两个对象没有定义有意义的比较方法（例如，比较一个数值和一个文件对象），则结果可能没有任何意义。在某些情况下，这样的比较也可能引发异常。

```
compile(string
, filename
```

```
, kind
 [, flags
 [, dont_inherit
]])
```

将 *string* 编译为代码对象，以便用于exec()或eval()。 *string* 是一个包含合法Python代码的字符串。如果该代码分布在多行上，则这些行的结尾必须使用新行符（'\n'）结束，并且不能是平台特有的形式（例如，Windows上的'\r\n'）。 *filename* 是一个包含文件名的字符串，在该文件中定义了这个字符串。 *kind* 可以是'exec'，即执行一系列语句，或者是'eval'，即执行一个单独的表达式，或者是'single'，即执行一条可执行语句。参数 *flags* 决定了启用哪些可选功能（与__future__ 模块相关）。使用__future__ 模块中定义的按位OR标志来指定这些功能。例如，如果要启用新的除法语法，就应将 *flags* 设为__future__.division.compiler_flag。如果省略 *flags* 或将其设为0，则使用当前有效的任意功能来编译代码。如果提供了 *flags*，则将指定的功能添加到那些已有效的功能中。如果设置了dont_inherit，则只启用在 *flags* 中指定的那些功能——忽略当前已启用的功能。

```
complex([real
 [, imag
]])
```

创建由实部和虚部构成的复数类型，其中实部和虚部（即 *real* 和 *imag*）可以是任意数值类型。如果省略 *imag*，则将虚部设为0。如果传入一个字符串作为 *real*，则解析该字符串并将它转换为复数。在这种情况下应忽略 *imag*。如果没有给定任何参数，则返回0j。

```
delattr(object
, attr
)
```

删除对象的属性。 *attr* 是一个字符串。与del *object .attr* 相同。

```
dict([m
])
或dict(key1
 = value1
, key2
 = value2
, ...)
```

创建一个字典类型。如果没有给定任何参数，则返回一个空字典。如果 *m* 是映射对象（例如字典），则返回一个具有与 *m* 相同键和值的新字典。例如，如果 *m* 是一个字典，则dict(*m*) 就创建它的一个浅复制。如果 *m* 不是映射对象，那么它必须支持迭代，能够产生一个由(*key ,value*) 对组成的序列。这些键值对用于填充该字典。也可以使用关键字参数调用dict() 。例如，dict(foo=3,bar=7) 会创建字典{ 'foo' : 3, 'bar' : 7 } 。

```
dir([object
])
```

返回属性名的有序列表。如果 *object* 是一个模块，则它包含在该模块中定义的符号列表。如果 *object* 是一个类型或类对象，则它返回一个属性名称列表。如果定义了对象的__dict__ 属性，则通常从该属性获得这些名称，但是也可能会使用其他来源。如果没有给定任何参数，则返回当前本地符号表中的名称。应当注意，该函数主要是提供信息的（例如，在命令行上交互式使用）。不应当用于正式的程序分析，因为得到的信息可能不完整。另外，用户定义的类可以定义一个特殊方法__dir__() ，它可改变该函数的结果。

```
divmod(a
, b
)
```

将长除法的商和余数作为元组返回。对于整数，返回的值是(*a* //*b* ,*a* %*b*)。对于浮点数，返回(math.floor(*a* /*b*),*a* %*b*)。使用复数时不能调用该函数。

```
enumerate(iter[, initial
 value
])
```

给定可迭代对象 *iter*，返回新迭代器（enumerate 类型），它产生的元组包含一个计数和 *iter* 产生的值。例如，如果 *iter* 产生 *a* 、 *b* 、 *c* ，则enumerate(*iter*)产生(0,*a*)、(1,*b*)、(2,*c*)。

```
eval(expr [, globals
 [, locals
]])
```

计算一个表达式的值。 *expr* 是一个字符串或由compile() 创建的代码对象。 *globals* 和 *locals* 分别是针对该操作定义的全局和局部命名空间的映射对象。如果省略，则在调用程序的命名空间中计算该表达式。最常见的情况是将 *globals* 和 *locals* 指定为字典，但是高级应用程序可以提供自定义映射对象。

```
exec(code [, global
 [, locals
]])
```

执行Python语句。 *code* 可以是字符串、文件或由compile() 创建的代码对象。 *globals* 和 *locals* 分别是针对该操作定义的全局和局部命名空间。如果省略，则在调用程序的命名空间中执行该代码。如果没有给定全局或局部字典，则该函数的行为在不同的Python版本之间有些不同。在Python 2中，实际上将exec 作为一个特殊的语言语句实现，而Python 3将其作为一个标准库函数实现。这种实现差别的副作用是在Python 2中，

由exec 求值得到的代码可以在调用程序的命名空间中任意改变局部变量。在Python 3中，可以执行实现此类改变的代码，但是看起来其持续效果不会超过exec() 调用本身。这是因为如果没有提供局部命名空间，Python 3会使用locals() 来得到它。就像在locals() 的文档中看到的那样，返回的字典只能查看而不能修改。

```
filter(function
, iterable
)
```

在Python 2中，该函数创建一个由 *iterable* 生成的对象组成的列表，这些对象在 *function* 中求值结果应为真。在Python 3中，创建的是一个产生此结果的迭代器。如果 *function* 是None ，则使用标识函数（即id() ）并且删除 *iterable* 中所有求值结果为假的的元素。 *iterable* 可以是任何支持迭代的对象。一般来说，使用生成器表达式或列表推导式来筛选数据要快得多（参见第6章）。

```
float([x
])
```

表示一个浮点数值的类型。如果 *x* 是数字，则将其转换为浮点数。如果 *x* 是字符串，则将其解析为浮点数。如果没有提供参数，则返回0.0 。

```
format(value
 [, format_spec
])
```

按照 *format_spec* 中的格式说明将 *value* 转换为有格式的字符串。该操作调用 *value* .__format__() ，它可以按照合适的样式来任意解释格式规格。对于简单数据类型，格式说明符通常包括对齐字符'<' 、'>' 或'^' ；数字（表明字段宽度），以及分别针对整数、浮点数或字符串值的字符代码'd' 、'f' 或's' 。例如，格式说明符d 格式化一个整数，8d 则在一个8字符的字段中右对齐一个整数，而<8d 则在一个8字符的字段中左对齐一个整数。第3章与第4章中介绍了关于format() 和格式说明符的更多内容。

```
frozenset([items
])
```

表示不变集合对象的类型，该对象的值来自于必须为可迭代的items 。这些值必须也是不可变的。如果没有给定参数，则返回一个空集合。

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

返回对象的一个命名属性的值。 *name* 是包含属性名的字符串。如果不存在这样的属性，则可选择返回 *default* ，否则引发AttributeError 。与 *object.name* 相同。

```
globals()
```

返回当前模块中代表全局命名空间的字典。在另一个函数或方法内调用它时，它返回定义该函数或方法的模块所拥有的全局命名空间。

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

如果 *name* 是 *object* 的属性名，则返回True 。否则返回False 。 *name* 是一个字符串。

```
hash(object
```

```
)
```

返回对象的整数散列值（如果有的话）。该散列值主要用在字典、集合及其他映射对象的实现中。对于相等的任意两个对象来说，其散列值是相同的。可变对象不会定义散列值，但是用户定义类可以定义__hash__()方法，以支持该操作。

```
help([object
])
```

在交互式会话中调用内置的帮助系统。object 可以是一个字符串，表示模块名、类名、函数名、方法名、关键字或文档主题名称。如果它是任意其他类型的对象，则会出现与该对象有关的帮助界面。如果没有提供参数，则启动一个交互帮助工具并提供更多信息。

```
hex(x
)
```

根据整数 *x* 创建一个十六进制字符串。

```
id(object
)
```

返回 *object* 的唯一整数标识符。不应以任何方式解释返回值（这是一个内存位置）。

```
input([prompt
])
```

在Python 2中，该函数打印一个提示符，读取输入行并通过eval()对其进行处理（即它与eval(raw_input(*prompt*)相同）。在Python 3中，将提示符打印到标准输出并读取一行输入，同时不进行任何求值或修改。

```
int(x [,base

])
```

表示整数的类型。如果 *x* 是数字，则通过截取到0将其转换为整数。如果它是字符串，则将其解析为整数值。可选的 *base* 设定将字符串转换为整数时的进制。在Python 2中，如果该值超出int 类型的32位范围，则创建一个长整数。

```
isinstance(object

, classobj

)
```

如果 *object* 是一个 *classobj* 的实例、 *classobj* 的子类或属于抽象基类 *classobj* ，则返回True 。参数 *classobj* 也可以是一些允许的类型或类的元组。例如，如果 *s* 是一个元组或列表，则isinstance(*s* , (list,tuple)) 返回True 。

```
issubclass(class1

, class2

)
```

如果 *class1* 是 *class2* 的子类（派生类）或 *class1* 是基于抽象基类 *class2* 注册的，则返回True 。 *class2* 也可以是一些类的元组，在这种情况下要检查每个类。注意issubclass(*A* , *A*) 的值是真。

```
iter(object
```

```
 [,sentinel
])
```

返回一个可生成 *object* 中各项的迭代器。如果省略 *sentinel* 参数，则该对象必须提供可创建迭代器的__iter__() 方法或者实现__getitem__()，后者接受从0 开始的整数参数。如果指定了 *sentinel*，则 *object* 的含义有所不同，它会变成一个无参数的可调用对象。返回的迭代器对象将重复调用该函数，直到返回值等于 *sentinel* 时停止迭代。如果 *object* 不支持迭代，则生成一个TypeError 。

```
len(s)
```

返回 *s* 中包含的项数， *s* 应当是列表、元组、字符串、集合或字典。如果 *s* 是像生成器这样的可迭代对象，则生成TypeError 。

```
ist([items
])
```

表示列表的类型， *items* 可以是任何可迭代对象，该对象的值将用于填充列表。如果 *items* 已经是一个列表，就生成一个副本。如果没有给定参数，则返回一个空列表。

```
locals()
```

返回与调用程序的局部命名空间相对应的字典。该字典应当只用于检查执行环境——修改该字典的内容并不安全。

```
long([x
 [, base
]])
```

在Python 2中表示长整数的类型。如果 *x* 是数字，则通过截取到`0` 将其转换为整数。如果 *x* 是字符串，则将其解析为长整数值。如果没有给定参数，则该函数返回`0L` 。为了可移植性考虑，应当避免直接使用`long` 。必要时使用`int(`*x*` )` 创建`long` 值。要进行类型检查，可使用`isinstance(`*x*` , numbers.Integral)` 检查 *x* 是否为任何整数类型。

```
map(function
, items
, ...)
```

在Python 2中，该函数将 *function* 应用到 *items* 的每一项并返回结果列表。在Python 3中，创建可产生相同结果的迭代器。如果提供多个输入序列，则认为 *function* 要使用那么多参数，并从不同的序列得到每个参数。在Python 2和Python 3中处理多个输入序列的做法并不相同。在Python 2中，结果值的长度与最长的序列等长，较短的输入序列用完时通过将`None` 用作填充值。在Python 3中结果值只与最短序列等长。使用生成器表达式或列表推导（这两种方式的性能更好）都可以很好地实现`map()` 提供的这种功能。例如，通常可以使用`[`*function*` (`*x*` ) for `*x*` in `*s*` ]` 代替`map(`*function*` , `*s*` )` 。

```
max(s
 [, args
, ...])
```

如果只有一个参数 *s* ，该函数返回 *s* 中各项的最大值， *s* 可以是任意可迭代的对象。如果有多个参数，它返回参数中的最大值。

```
min(s
 [, args
, ...])
```

如果只有一个参数 *s* ，该函数返回 *s* 中各项的最小值， *s* 可以是任意可迭代的对象。如果有多个参数，它返回参数中的最小值。

```
next(s
 [, default
])
```

返回迭代器 *s* 中的下一项。如果该迭代器没有下一项，则引发StopIteration 异常，除非指定了 *default* 参数的值。如果指定了 *default* 的值，则会返回 *default* 。为了可移植性，应当用该函数而不是直接调用迭代器 *s* 的 *s* .next() 。在Python 3中，底层的迭代器方法名改为 *s* .__next__() 。如果编写的代码使用next()，则不必担心这种差别。

```
object()
```

Python中所有对象的基类。可以调用它创建一个实例，但是结果并不会特别让你感兴趣。

```
oct(x
)
```

将整数 *x* 转换为一个八进制字符串。

```
open(filename
 [, mode
 [, bufsize
]])
```

在Python 2中，打开文件 *filename* 并返回一个新文件对象（参见第9章）。 *mode* 是一个字符串，说明应如何打开文件：'r' 表示读，'w' 表示写，而'a' 表示添加。第二个字符't' 或'b' 用于说明文件是以文本模式打开（默认）还是以二进制模式打开。例如，'r' 或'rt' 表示以文本模式打开文件，而'rb' 表示以二进制模式打开文件。可以将可选的'+' 添加到模式（mode ）中，以便打开文件进行更新（既可以读也可以写）。如果文件已存在，则'w+' 模式将该文件截取到零长度。'r+' 或'a+' 模式可以打开文件进行读写，但是在打开时保留原有内容不变。如果指定'U' 或'rU' 模式，则在通用换行模式下打开文件。在这种模式中，所有形式的换行符（'\n' 、'\r' 、'\r\n' ）都转换为标准字符'\n' 。如果省略模式参数，则默认为'rt' 模式。参数bufsize 指定缓冲行为，其中0 表示不缓冲、1 表示行缓冲，而其他任意正数表明相应的缓存大小，单位为字节。负数表示应当使用系统默认的缓冲行为（这是默认行为）。

```
open(filename
 [, mode
 [, bufsize
 [, encoding
 [, errors
 [, newline
 [,closefd
]]]]]])
```

在Python 3中，该函数打开文件 *filename* 并返回文件对象。前3个参数与前面介绍的Python 2版本的open() 具有相同的意义。 *encoding* 是诸如'utf-8' 这样的编码名。 *errors* 是错误处理策略，其值可以是'strict' 、'ignore' 、'replace' 、'backslashreplace' 或'xmlcharrefreplace' 。 *newline* 控制通用换行模式的行为，可以设为None 、' '（空格）、'\n' 、'\r' 或'\r\n' 。 *closefd* 是布尔标志，它指定在执行close() 方法时是否关闭基本文件描述符。与Python 2不同，根据所选的I/O模式会返回不同种类的对象。例如，如果以二进制模式打开文件，则得到一个对象，其中像read() 和write() 这样的I/O操作只处理字节数组而不处理字符串。文件I/O是Python 2与Python 3之间存在显著差异的地方。请参见附录A，了解更多信息。

```
ord(c
)
```

返回字符 *c* 的整数序值。如果是普通字符，返回范围在[0,255] 内的值。如果是单个Unicode字符，通常返回范围在[0,65535] 的值。在Python 3中， *c* 也可以是Unicode代理对，在这种情况下，会将其转换为合适的Unicode代码点。

```
pow(x
, y
 [, z
])
```

返回 *x* ** *y* 。如果提供了 *z* ，则该函数返回(*x* ** *y*) % *z* 。如果给定所有3个参数，则它们必须是整数并且 *y* 必须非负数。

```
print(value
, ... [, sep=separator
, end=ending
, file=outfile
])
```

Python 3中用于打印一系列值的函数。可以提供任意数量的值作为输入，所有值都会打印在同一行。关键字参数sep 用于指定不同的分隔符字符（默认是空格）。关键字参数end 指定不同的行结束（默认是'\n' ）。关键字参数file 将输出重定向到文件对象。在Python 2中，如果在代码中添加了语句from __future__ import print_function ，就可以使用该函数。

```
property([fget
 [,fset
 [,fdel
 [,doc
]]])
```

创建类的特性属性。 *fget* 是返回属性值的函数， *fset* 设置属性值，而 *fdel* 删除一个属性。 *doc* 表示文档字符串。可以使用关键字参数提供这些参数，例如 `property(fget=getX, doc="some text")` 。

```
range([start
,] stop
 [, step
])
```

在Python 2中，该函数创建一个完全填充的、从 *start* 到 *stop* 的整数列表。 *step* 表示步长值，如果省略则默认设为1。如果省略 *start* （使用参数调用`range()` 时），则默认为0。负数 *step* 创建降序排列的数值列表。在Python 3中，`range()` 创建特殊的 `range` 对象，它可根据需要计算其值（像`Python` 以前版本中的`xrange()` ）。

```
raw_input([prompt
])
```

Python 2中的函数，从标准输入（`sys.stdin` ）读取一行输入并将其作为字符串返回。如果提供 *prompt* ，则首先将其打印到标准输出（`sys.stdout` ）。打印时会截掉末尾的新行，如果读到EOF，则引发`EOFError` 异常。如果加载了`readline` 模块，则该函数使用它来提供高级行编辑和命令完成功能。在Python 3中使用`input()` 读取输入。

```
repr(object
)
```

返回 *object* 的字符串表示形式。在大多数情况下，返回的字符串是一个表达式，可以传递到`eval()` 用来重新创建对象。注意，在Python 3中该函数的结果可能是一个Unicode字符串，在终端或shell窗口中不能显示（导致异常）。使用`ascii()` 函数可创建 *object* 的ASCII表示形式。

```
```

```
reversed(s
)
```

创建序列s的逆序迭代器。只有当 *s* 实现了序列方法__len__() 和__getitem__() 时才能使用该函数。另外， *s* 必须从*0* 开始建立各项的索引。它不能与生成器或迭代器一起使用。

```
round(x
 [, n
])
```

将浮点数 *x* 四舍五入到最近的*10* 的负 *n* 次方倍数后再四舍五入。如果省略 *n* ，则默认为0。如果两个倍数非常接近，则Python 2返回远离0的四舍五入值（例如，0.5的四舍五入值是1.0，而-0.5的四舍五入值是-1.0）。在Python 3中，如果前一个数字为偶数，则朝0进行四舍五入，否则远离0（例如，0.5四舍五入为0.0，而1.5四舍五入为2）。

```
set([items
])
```

创建一个使用从可迭代对象 *items* 得到的各项来填充的集合。这些项必须不可变。如果 *items* 包含其他集合，则这些集合必须是 *frozenset* 类型。如果省略 *items* ，则返回空集合。

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

设置对象的属性。name 是字符串。与object.name = value 相同。

```
slice([start
,] stop
 [, step
])
```

返回表示指定范围内整数的切片对象。通过扩展切片语法a[i:i:k] 也可以生成切片对象。参见3.9.7节，了解详细内容。

```
sorted(iterable
 [, key=keyfunc
 [, reverse=reverseflag
]])
```

根据 *iterable* 中的各项创建有序列表。key 关键字参数是一个单参数函数，在将这些值传递到比较函数之前对它们进行转换。reverse 关键字参数是一个布尔标志，指定是否对生成的列表进行逆序排列。必须使用关键字指定key 和reverse 参数，例如 sorted(a,key=get_name) 。

```
staticmethod(func
)
```

创建在类中使用的静态方法。通过@staticmethod 装饰器隐式调用该函数。

```
str([object
])
```

表示字符串的类型。在Python 2中，一个字符串包含8位字符，而在Python 3中的字符串都是Unicode。如果提供了 *object* ，则通过调用它的__str__()方法来创建其值的字符串表示形式。这个字符串与打印该对象时看到的内容相同。如果没有给定参数，则创建一个空字符串。

```
sum(items
 [,initial
])
```

计算从可迭代对象 *items* 中得到的所有项的总数。 *initial* 是初始值，默认是*0* 。该函数只适用于数值。

```
super(type
 [, object
])
```

返回表示 *type* 超类的对象。该对象的主要用途是调用基类中的方法，如下所示：

```
class B(A):
     def foo(self):
          super(B,self).foo()
```

如果 *object* 是一个对象，那么isinstance(*object* , *type*)必须为真。如果 *object* 是一个类型，那么它必须是 *type* 的一个子类。参见第7章，了解更多内容。在Python 3中，可以在方法中使用super()， 不必提供参数。在这种情况下，将 *type* 设为定义该方法的类，而将 *object* 设为该方法的第一个参数。尽管这样会使语法很简洁，但是不能与Python 2实现向后兼容，因此如果考虑可移植性，则应避免这样使用。

```
tuple([items
])
```

表示元组的类型。如果提供了 *items* ，则它应是用于填充该元组的可迭代对象。但是，如果 *items* 已经是一个元组，则只是原封不动地将其返回。如果没有给定参数，则返回空元组。

```
type(object

)
```

Python中所有类型的基类。当作为函数调用它时会返回 *object* 的类型。该类型与这个对象的类相同。对于整数、浮点数和列表这样的常见类型来说，该类型就是指如`int`、`float`、`list`等其他内置类之一。对于用户定义的对象，该类型是相关联的类。对于与Python内部相关的对象，通常得到的是对 *types* 模块中定义的一个类的引用。

```
type(name

,bases

,dict

)
```

创建一个新 *type* 对象（相当于定义一个新类）。 *name* 是类型名， *bases* 是基类元组，而 *dict* 是一个包含对应于类主体定义的字典。该函数最常用于元类。第7章进一步介绍了该函数。

```
unichr(x

)
```

将整数或长整数 *x* 转换为一个Unicode字符，其中`0 <= x <= 65535` 。只适用于Python 2。在Python 3中只需使用`chr(x )`即可。

```
unicode(string

 [,encoding

 [,errors
```

```
]])
```

在Python 2中，该函数将 *string* 转换为Unicode字符串。 *encoding* 指定 *string* 的数据编码。如果省略，则使用sys.getdefaultencoding() 返回的默认编码。 *errors* 指定如何处理编码错误，其值可以是'strict' 、'ignore' 、'replace' 、'backslashreplace' 或xml'charrefreplace' 。如需详细信息，请参见第9章和第3章。Python 3中没有该功能。

```
vars([object

])
```

返回 *object* 的符号表（通常在它的__dict__ 属性中）。如果没有给定参数，则返回对应局部命名空间的字典。该函数返回的字典应该是只读的。修改它的内容并不安全。

```
xrange([start

,] stop

 [, step

])
```

表示从 *start* 到 *stop* （不包括 *stop* ）的整数值范围的类型。 *step* 是可选的步长值。实际上并不存储这些值，而是在访问时根据要求进行计算。在Python 2中，编写对某个范围的整数值进行循环操作时首选xrange() 函数。在Python 3中，xrange() 已重命名为range() ，不能再使用xrange() 了。而 *start* 、 *stop* 和 *step* 都只能是机器整数（通常是32位）所支持的值的集合。

```
zip([s1

 [, s2

 [,..]]])
```

在Python 2中，返回一个元组列表，其中第 *n* 个元组是(*s1* [*n*], *s2* [*n*], ...) 。生成的列表被截取为最短参数序列的长度。如果没有给定参数，则返回一个空列表。在Python 3中的行为与此相似，但结果是一个可生成元组序列的迭代器。在Python 2中，要注意对特别长的输入序列使用zip() 可能会无意中占用大量内存。建议使用itertools.izip() 替代。

12.2 内置异常

在exceptions 模块中包含了内置的异常，执行任何程序之前总是先载入该模块。这些异常都定义为类。

12.2.1 异常基类

下列异常可作为所有其他异常的基类。

BaseException

所有异常的根类。所有内置异常都派生自该类。

Exception

所有与程序有关的异常的基类，这些异常包括除SystemExit 、GeneratorExit 和Keyboard- Interrupt 之外的所有内置异常。应通过继承Exception 来定义用户定义的异常。

ArithmeticError

算术异常的基类，这些异常包括OverflowError 、ZeroDivisionError 和FloatingPointError 。

LookupError

索引和键错误的基类，包括IndexError 和KeyError 。

EnvironmentError

在Python外部发生的错误的基类，包括IOError 和OSError 。

前面这些异常都不会被显式引发。但是，可以使用它们捕捉某类错误。例如，以下代码会捕捉任意数值错误：

```
try:
     # 一些操作
     ...
except ArithmeticError as e:
     # 数学错误
```

12.2.2 异常实例

产生异常时会创建异常类的实例。该实例置于except 语句的可选变量中，如下所示：

```
except IOError as e:
     # 处理错误
     # 'e'有一个IOError实例
```

异常e 的实例具有一些标准属性，在某些应用程序中进行检查和/或处理时很有用。

e

.args

引发异常时提供的参数元组。在大多数情况下，这是一个单项元组，其中包含有描述该错误的字符串。对于EnvironmentError 异常，该值是一个两项元组或三项元组，包含整数错误编号、字符串错误消息和可选的文件名。如果需要在不同的环境中重新生成该异常，该元组的内容可能很有用；例如，要在一个不同的Python解释器进程中引发一个异常。

```
e
.message
```

一个字符串，代表在显示异常时打印的错误消息（只支持Python 2）。

```
e
.__cause__
```

使用显式关联异常时的前一个异常（只支持Python 3）。参见附录A。

```
e
.__context__
```

使用隐式关联异常时的前一个异常（只支持Python 3）。参见附录A。

```
e
.__traceback__
```

与异常相关的跟踪对象（只支持Python 3）。参见附录A。

12.2.3 预定义的异常类

以下异常由程序引发。

```
AssertionError
```

失败的assert 语句。

AttributeError

失败的属性引用或赋值。

EOFError

文件末尾。由内置函数input() 和raw_input() 生成。应当注意，诸如文件的read() 和readline() 方法等大多数其他I/O操作将返回空字符串来表明EOF，而不是引发异常。

FloatingPointError

失败的浮点操作。应当注意，浮点异常处理是一个很棘手的问题，而且只有在Python配置和构建时启用了该特性时才会引发该异常。出现浮点错误后，更常见的是悄悄地产生诸如float('an') 或float('inf') 这样的结果。它是ArithmeticError 的子类。

GeneratorExit

在生成器函数内部引发的错误，目的是表明中止。过早销毁生成器（在使用全部生成器值之前）或调用生成器的close() 方法时会引发该错误。如果生成器忽略该异常，则该生成器被中止并且忽略该异常。

IOError

失败的I/O操作。该值是一个IOError 实例，包含属性erno 、strerror 和filename 。erno 是整数错误编号，strerro 是字符串错误消息，而filename 是可选的文件名。它是EnvironmentError 的子类。

ImportError

当import 语句无法找到模块或者from 无法在模块中找到名称时引发该错误。

IndentationError

缩进错误。它是SyntaxError 的子类。

IndexError

序列下标超出范围。它是LookupError 的子类。

KeyError

在映射中未找到键。它是LookupError 的子类。

KeyboardInterrupt

用户按下中断健（通常是Ctrl+C）时引发该错误。

MemoryError

可恢复的内存不足错误。

NameError

在局部或全局命名空间中未找到名称。

NotImplementedError

未实现的功能。当基类需要派生类实现某些方法时可能引发该错误。它是RuntimeError 的子类。

OSError

操作系统错误。主要由os 模块中的函数引发。该值与IOError 的值相同。它是EnvironmentError 的子类。

OverflowError

由于一个整数值太大而无法表示它所造成的结果。通常在将较大的整数值传递给对象，而此对象内部依赖于固定精度机器整数时会引发该异常。例如，如果指定的起始或结束值超出32位的大小，range 或xrange 对象会引发该异常。它是ArithmeticError 的子类。

ReferenceError

在底层对象被销毁后访问弱引用就会产生此错误。参见weakref 模块。

RuntimeError

其他任何类别未包括的一般错误。

StopIteration

引发该异常可指示迭代结束。通常在对象的next() 方法中或生成器函数中出现。

SyntaxError

解析器语法错误。这些实例的属性filename 、lineno 、offset 和text 可以用于收集更多信息。

SystemError

编译器中的内部错误。该值是一个指明问题的字符串。

SystemExit

由sys.exit() 函数引发。该值是表示返回码的整数。如果必须立刻退出，就可以使用os._exit() 。

TabError

不一致的制表符用法。使用-tt 选项运行Python时生成该错误。它是SyntaxError 的子类。

TypeError

将操作或函数应用到类型不合适的对象时出现该错误。

UnboundLocalError

引用了未绑定的局部变量。如果引用了还没有在函数内定义的变量，则发生该错误。它是Name-Error 的子类。

UnicodeError

Unicode编码或解码错误。它是ValueError 的子类。

UnicodeEncodeError

Unicode编码错误。它是UnicodeError 的子类。

UnicodeDecodeError

Unicode解码错误。它是UnicodeError 的子类。

```
UnicodeTranslateError
```

在转换过程中产生的Unicode错误。它是UnicodeError 的子类。

```
ValueError
```

当一个函数或操作的参数是正确类型但值不正确时，就会生成该错误。

```
WindowsError
```

Windows系统调用失败所生成的错误。它是OSError 的子类。

```
ZeroDivisionError
```

除零错误。它是ArithmeticError 的子类。

12.3 内置警告

Python有一个warnings 模块，通常用它通知程序员被废弃的功能。通过包含如下所示代码即可发出警告。

```
import warnings
warnings.warn("The MONDO flag is no longer supported", DeprecationWarning)
```

虽然警告都是经过库模块发出的，但是各种警告的名称都是内置的。警告与异常有些类似。所有继承自Exception 的内置警告都具有一种层次结构。

Warning

所有警告的基类。它是Exception 的子类。

UserWarning

一般的用户定义警告。它是Warning 的子类。

DeprecationWarning

警告使用了被废弃的功能。它是Warning 的子类。

SyntaxWarning

警告使用了被废弃的Python语法。它是Warning 的子类。

RuntimeWarning

警告可能存在潜在的运行时问题。它是Warning 的子类。

FutureWarning

警告某个功能的行为未来会发生变化。它是Warning 的子类。

警告不同于异常，因为通过warn() 函数发出的警告不一定会导致程序停止运行。例如，警告可能就是将某些内容打印到输出或者是引发一个异常。利用warnings 模块或-W 选项可以配置编译器的实际行为。如果使用别人的代码时出现了警告，但又想继续进行操作，就可以使用try 和except 捕捉已转换为异常的警告。例如：

```
try:
      import md5
except DeprecationWarning:
      pass
```

需要强调的是这样的代码很少见。虽然这样会捕捉已经转换为异常的警告，但并不会禁止警告信息（必须使用warnings 模块进行控制）。而且，当Python新版本发布时，如果编写的代码不能正确执行，那么最好忽视警告。

12.4 future_builtins

只能在Python 2中使用future_builtins 模块，该模块实现的内置函数的行为在Python 3中已发生改变。已定义的函数如下所示。

```
ascii(object

)
```

产生的输出与repr() 相同。请参考12.1节中的内容。

```
filter(function

, iterable

)
```

创建迭代器而不是列表。等同于itertools.ifilter() 。

```
hex(object

)
```

创建十六进制字符串，但使用__index__()特殊方法来得到整数值，而不是调用__hex__()。

```
map(function
, iterable
, ...)
```

创建迭代器而不是列表。等同于itertools.imap()。

```
oct(object
)
```

创建八进制字符串，但使用__index__()特殊方法来得到整数值，而不是调用__oct__()。

```
zip(iterable
, iterable
, ... )
```

创建迭代器而不是列表。等同于itertools.izip()。

注意，该模块中列出的函数并没有涵盖内置模块的所有变化。例如，Python 3还将raw_input()重命名为input()，将xrange()改为range()。

第13章　Python运行时服务

本章介绍与Python解释器运行时相关的模块。内容包括垃圾回收、基本的对象管理（复制、列集等）、弱引用和解释器环境。

13.1　atexit

atexit 模块用于注册Python解释器退出时要执行的函数。它只提供一个函数：

```
register(func
 [,args
 [,kwargs
]])
```

将函数 *func* 添加到解释器退出时要执行的函数列表中。 *args* 是传递到该函数的参数元组。 *kwargs* 是关键字参数字典。以 *func* (**args* ,***kwargs*) 的形式调用该函数。解释器退出时，按照注册顺序的逆序调用这些函数（首先调用最新添加的退出函数）。如果出现错误，则将一条异常消息打印到标准错误，但是这个错误会被忽略。

13.2　copy

copy 模块提供了创建复合对象（包括列表、元组、字典和用户定义对象的实例）的深复制和浅复制的函数。

```
copy(x
)
```

创建新的复合对象并通过引用复制 *x* 的成员来创建 *x* 的浅复制。对于内置类型，该函数并不常用。而是使用诸如list(*x*)、dict(*x*)、set(*x*) 等调用方式来创建 *x* 的浅复制（要知道像这样直接使用类型名比使用copy() 快很多）。

```
deepcopy(x
```

```
 [, visit

])
```

通过创建新的复合对象并递归复制 *x* 的所有成员来创建 *x* 的深复制。 *visit* 是一个可选的字典，目的是跟踪受访问的对象，从而检测和避免重复定义的数据结构中的循环。如本章稍后所述，只有在递归调用deepcopy()时，才使用该参数。

尽管通常情况下不需要，但是通过实现方法__copy__(*self*)和__deepcopy__(*self* , *visit*)，类就可以实现自定义的复制方法，这两个方法分别实现了浅复制和深复制操作。__deepcopy__()方法必须使用字典 *visit* ，用来在复制过程中跟踪前面遇到的对象。对于__deepcopy__()方法，除了将visit传到实现中包含的其他deepcopy()操作（如果有的话）之外，没有必要再执行其他操作。

如果类实现了pickle模块所用的方法__getstate__()和__setstate__()，那么copy模块将使用这些方法来创建副本。

注意

- 该模块可以用于像整数和字符串这样的简单类型，不过很少需要这么做。
- 这些复制函数不适用于模块、类对象、函数、方法、回溯、栈帧、文件、套接字和其他类似类型。如果不能复制对象，则会引发copy.error异常。

13.3 gc

gc模块提供了一个控制垃圾回收器的接口，垃圾回收器可用于回收像列表、元组、字典和实例等对象中。创建各种类型的容器对象时，它们会被添加至解释器内部的一个列表上。只要取消分配给容器对象的内存，就会将它们从该列表中删除。如果分配数量超出用户可定义的取消分配数量阈值，就会调用垃圾回收器。垃圾回收器扫描该列表，识别并回收不再使用但是由于循环依赖关系而未取消分配的对象。此外，垃圾回收器使用一种三级代模式（three-level generational scheme），在初次垃圾回收环节未涉及的对象都放在检查频率最低的对象列表上。这样会让具有大量长期存在对象的程序实现更好的性能。

```
collect([generation

])
```

运行一次完整的垃圾回收操作。该函数检查所有代并返回所找到的不可达对象数。generation是范围在0到2的可选整数，用于指定要回收的代。

disable()

禁用垃圾回收。

enable()

启用垃圾回收。

garbage

一个变量，它包含的只读列表中列出了不再使用的用户定义实例，但是不能对这些实例进行垃圾回收，因为它们有引用循环，而且定义了__del__() 方法。不能对这样的对象进行垃圾回收，因为解释器要中断引用循环，就必须首先销毁其中一个对象。但是，我们无法判断剩余对象的__del() 方法是否需要对刚销毁的对象执行重要操作。

get_count()

返回一个元组(*count0 , count1 , count2*)，它包含每代中当前的对象数。

get_debug()

返回当前设置的调试标志。

get_objects()

返回垃圾回收器正在跟踪的所有对象列表。不包括这个返回的列表。

```
get_referrers(obj1
, obj2
, ...)
```

返回直接引用obj1 、obj2 等对象的所有对象列表。返回的列表可能包括还没有被回收的对象，以及部分构建的对象。

```
get_referents(obj1
, obj2
, ...)
```

返回 *obj1* 、 *obj2* 等对象引用的对象列表。例如，如果 *obj1* 是一个容器，则该函数返回容器中的对象列表。

```
get_threshold()
```

返回元组形式的当前回收阈值。

```
isenabled()
```

如果启用了垃圾回收，则返回True 。

```
set_debug(flags
```

```
)
```

设置垃圾回收调试标志，可以用于调试垃圾回收器的行为。 *flags* 是常量DEBUG_STATS 、DEBUG_COLLECTABLE 、DEBUG_UNCOLLECTABLE 、DEBUG_INSTANCES 、DEBUG_OBJECTS 、DEBUG_SAVEALL 和DEBUG_LEAK 的按位OR。DEBUG_LEAK 标志可能最有用，因为调试程序出现内存泄漏时它会让回收器打印有用信息。

```
set_threshold(threshold0
 [, threshold1
[, threshold2
]])
```

设置垃圾回收的回收频率。对象都被划分为三代，其中0代包含最新的对象，而2代包含最早的对象。在一次垃圾回收环节中未被回收的对象都移到下一个最早的代。一旦对象到达2代，它就会留在此代。 *threshold0* 是在0代进行垃圾回收之前必须达到的分配数和取消分配数之差。 *threshold1* 是在扫描1代之前必须进行的0代回收数量。 *threshold2* 是在回收2代垃圾之前必须进行的1代回收数量。默认阈值目前设为(700, 10, 10) 。将 *threshold0* 设为0将禁用垃圾回收。

注意

- 循环引用如果涉及了使用__del__() 方法的对象，则不会进行垃圾回收并且会置于gc.garbage 列表上（非回收对象）。由于存在与对象终止有关的困难问题，因此不会回收这些对象。
- 函数get_referers() 和get_referents() 只适用于支持垃圾回收的对象。此外，这些函数只能用于调试，不应将它们用于其他用途。

13.4 inspect

inspect 模块用于收集关于Python活动对象的信息，如属性、文档字符串、源代码、栈帧等信息。

```
cleandoc(doc
)
```

清理文档字符串 *doc* ，将所有tab 改为空格并删除可能使文档字符串与函数或方法内的其他语句对齐而插入的缩进。

```
currentframe()
```

返回与调用程序的栈帧对应的帧对象。

```
formatargspec(args
 [, varags
 [, varkw
 [, defaults
]]])
```

生成表示getargspec() 返回值的格式良好的字符串。

```
formatargvalues(args
 [, varargs
 [, varkw
 [, locals
]]])
```

生成表示getargvalues() 返回值的格式良好的字符串。

```
getargspec(func
)
```

给定一个函数 *func* ，返回一个命名元组ArgSpec(*args* , *varargs* , *varkw* , *defaults*)。 *args* 是参数名列表，而 *varargs* 是* 参数名（如果有的话）， *varkw* 是** 参数名（如果有的话）， *defaults* 是默认参数值的元组，如果没有默认参数值，则为None 。如果有默认参数值，则 *defaults* 元组表示 *args* 中最后 *n* 个参数的值，其中 *n* 是len(*defaults*)。

```
getargvalues(frame

)
```

返回提供给具有执行帧 *frame* 的函数的参数的值。返回元组ArgInfo(*args* , *varargs* , *varkw* , *locals*)。 *args* 是参数名列表， *varargs* 是* 参数名（如果有的话），而 *varkw* 是** 参数名（如果有的话）。 *locals* 是该帧的本地字典。

```
getclasstree(classes

 [, unique

])
```

给定一个相关类 *classes* 的列表，该函数根据继承关系将这些类排列为层次结构。该层次结构表示为嵌套列表的集合，列表中的每个条目都是一个类列表，这些类继承自紧接该列表之前的类。该列表中的每个条目是一个2元组(*cls* ,*bases*)，其中 *cls* 是类对象，而 *bases* 是基类的元组。如果 *unique* 为True，则每个类只在返回的列表中出现一次。否则，如果使用多种继承关系，则一个类会多次出现。

```
getcomments(object

)
```

返回Python源代码中紧接 *object* 定义之前的注释组成的字符串。如果 *object* 是一个模块，则返回在模块上方定义的注释。如果没有找到注释，则返回None 。

```
getdoc(object
```

```
)
```

返回 *object* 的文档字符串。在返回之前首先使用cleandoc()处理该文档字符串。

```
getfile(object
)
```

返回定义了 *object* 的文件名。如果该信息不可用或不能得到，则返回TypeError（例如，对于内置函数）。

```
getframeinfo(frame
 [, context
])
```

返回命名元组Traceback(*filename* , *lineno* , *function* ,*code* _*context* , *index*)，它包含关于帧对象 *frame* 的信息。 *filename* 和 *line* 指定源代码的位置。参数 *context* 设置要获取的源代码的上下文行数。在返回元组中的 *contextlist* 字段包含与该上下文对应的源代码行列表。字段 *index* 是对应于 *frame* 行的那个列表内的数字索引。

```
getinnerframes(traceback
 [, context
])
```

返回回溯帧和所有内部帧的帧记录列表。每个帧记录是由(*frame* , *filename* , *line* , *funcname* , *contextlist* , *index*)组成的6元组。 *filename* 、 *line* 、 *context* 、 *contextlist* 和 *index* 的含义与getframeinfo()中的含义相同。

```
getmembers(object
 [, predicate
])
```

返回 *object* 的所有成员。通常，通过查找对象的__dict__ 属性来得到这些成员，但是该函数可能返回存储在其他地方的 *object* 属性。例如，__doc__ 中的文档字符串，__name__ 中的对象名等。返回的成员都是一个(*name* , *value*) 对列表。*predicate* 是一个可选函数，它接受一个成员对象作为参数，并返回True 或False 。只有 *predicate* 返回True 的成员才被返回。像isfunction() 和isclass() 这样的函数可以用作谓词（predicate）函数。

```
getmodule(object
)
```

返回定义了 *object* 的模块（如果可能的话）。

```
getmoduleinfo(path
)
```

返回关于Python如何解释文件 *path* 的信息。如果 *path* 不是Python模块，则返回None 。否则，返回一个命名元组ModuleInfo(*name* , *suffix* , *mode* , *module_type*)，其中 *name* 是模块名， *suffix* 是文件名前缀， *mode* 是用于打开该模块的文件模式，而 *module_type* 是一个整数代码，用于设置模块类型。在imp 模块中定义了模块类型代码。

模块类型	说　　明	模块类型	说　　明
imp.PY_SOURCE	Python源文件	imp.PKG_DIRECTORY	包目录
imp.PY_COMPILED	Python已编译对象文件（.pyc ）	imp.C_BUILTIN	内置模块

imp.C_EXTENSION	动态可加载的C扩展	imp.PY_FROZEN	冻结的模块

```
getmodulename(path
)
```

返回用于文件 *path* 的模块名。如果 *path* 看起来不像Python模块，则返回None 。

```
getmro(cls
)
```

返回一个类的元组，表示方法解析排序，这些排序用于解析cls 类中的方法。参见第7章，了解更多相关细节。

```
getouterframes(frame
 [, context
])
```

返回 *frame* 和所有外部帧的帧记录列表。该列表表示调用顺序，其中第一项包含 *frame* 的信息。每个帧记录是一个6元组(*frame* , *filename* , *line* , *funcname* , *contextlist* , *index*)，这些字段的含义与getinnerframes() 中的含义相同。 *context* 参数的含义与getframeinfo() 中的相同。

```
getsourcefile(object
)
```

返回其中定义了 *object* 的Python源文件名。

```
getsourcelines(object
)
```

返回对应于 *object* 定义的元组(*sourcelines* , *firstline*)。 *Sourcelines* 是源代码行的列表，而 *firstline* 是第一行源代码的行号。如果没有找到源代码，则引发IOError 。

```
getsource(object
)
```

将 *object* 的源代码作为一个单独的字符串返回。如果没有找到源代码，则引发 IOError 。

```
isabstract(object
)
```

如果 *object* 是一个抽象基类，则返回True 。

```
isbuiltin(object
)
```

如果 *object* 是一个内置函数，则返回True 。

```
isclass(object
)
```

如果 *object* 是一个类，则返回True 。

```
iscode(object
)
```

如果 *object* 是一个代码对象，则返回True 。

```
isdatadescriptor(object
)
```

如果 *object* 是一个数据描述符对象，则返回True 。如果 *object* 定义了 __get__() 和__set__() 方法，则属于这种情况。

```
isframe(object
)
```

如果 *object* 是一个帧对象，则返回True 。

```
isfunction(object
)
```

如果 *object* 是一个函数对象，则返回True 。

```
isgenerator(object
)
```

如果 *object* 是一个生成器对象，则返回True 。

```
isgeneratorfunction(object
)
```

如果 *object* 是一个生成器函数，则返回 *True* 。它与isgenerator() 不一样，因为它测试 *object* 是否是调用时创建一个生成器的函数。它不适用于检查 *object* 是否为正在运行的生成器。

```
ismethod(object
)
```

如果 *object* 是一个方法，则返回True 。

```
ismethoddescriptor(object
)
```

如果 *object* 是方法描述符对象，则返回True 。如果object 不是方法、类或函数并且定义了__get__() 方法而没有定义__set__()，则属于这种情况。

```
ismodule(object
)
```

如果 *object* 是一个模块对象，则返回True 。

```
isroutine(object
)
```

如果 *object* 是一个用户定义的或内置函数或方法，则返回True 。

```
istraceback(object
)
```

如果 *object* 是一个回溯对象，则返回True 。

```
stack([context
])
```

返回与调用程序栈对应的帧记录列表。每个帧记录是一个6元组(*frame , filename , line , funcname , contextlist , index*)，它包含的信息
与getinnerframes() 返回的信息相同。 *context* 设置在每个帧记录中要返回的源代码的行数。

```
trace([context
])
```

返回当前帧与引发当前异常的帧之间栈的帧记录列表。第一个帧记录是调用程序，最后的帧记录是发生异常的帧。context 设置在每个帧记录中要返回的源代码的行数。

13.5 marshal

marshal 模块将Python对象序列化为“未文档化”的Python特定数据格式。marshal 类似于pickle 和shelve 模块，但是功能不太强大而且设计的初衷是只用于简单对象。通

常不应使用它实现持久化对象（而应使用pickle ）。不过，对于简单的内置类型，marshal 模块是一种保存和加载数据的快速方式。

```
dump(value
, file
 [, version
])
```

将对象值写到打开的文件对象 *file* 中。如果 *value* 是不支持的类型，则引发ValueError 异常。 *version* 是一个整数，它设置要使用的数据格式。在marshal.version 中可以找到默认输出格式，目前设为2。版本0是Python较早版本使用的旧格式。

```
dumps(value
 [,version
])
```

返回由dump() 函数所写的字符串。如果 *value* 是不支持的类型，则引发ValueError 异常。 *Version* 与前面说明的含义相同。

```
load(file
)
```

读取并从打开的文件对象 *file* 返回下一个值。如果没有读到有效值，则引发EOFError 、ValueError 或TypeError 异常。此函数会自动检测输入数据的格式。

```
loads(string
)
```

读取并返回字符串string 中的下一个值。

注意

- 数据都以独立于架构的二进制格式存储。
- 只支持None 、整数、长整数、浮点数、复数、字符串、Unicode字符串、元组、列表、字典和代码对象。列表、元组和字典只能包含所支持的对象。不支持类实例和列表、元组以及字典中的递归引用。
- 如果内置整数类型精度不够高，则将整数提升为长整数。例如，被列集的数据包含一个64位整数，但是要在32位机器上读取该数据时。
- marshal 并不能防止错误出现或防止恶意构建的数据侵害，不应当用来对来自不受信任来源的数据取消列集。
- marshal 比pickle 的速度更快，但是不够灵活。

13.6 pickle

pickle 模块用于将Python对象序列化为适合在文件中存储、可通过网络传输或可置于数据库中的字节流。该过程有不同的叫法，可以称为pickling、序列化 、列集化 或平面化 。使用反序列化（unpickle）过程也可以将生成的字节流转换回Python对象。

下列函数用于将对象转换为字节流。

```
dump(object
, file
 [, protocol
 ])
```

将 *object* 的序列化表示形式转储到文件对象 *file* 中。 *protocol* 设置数据输出格式。协议0（默认）是基于文本的格式并向后兼容较早的Python版本。协议1是二进制协议并且也兼容更早的Python版本。协议2是较新的协议，它提供了更高效的类和实例序列化。协议3用于Python 3并且不能向后兼容。如果 *protocol* 是负值，则选择最新的协议。变量pickle.HIGHEST_PROTOCOL 包含可用的最高级协议。如果 *object* 不支持序列化，则引发pickle.PicklingError 异常。

```
dumps(object
 [, protocol
])
```

```

```

与dump()相同，但是返回一个包含已序列化数据的字符串。

下面的例子显示如何使用这些函数将对象保存为文件：

```
f = open('myfile', 'wb')
pickle.dump(x, f)
pickle.dump(y, f)
... dump more objects

 ...
f.close()
```

下列函数用于还原已序列化的对象。

```
load(file

)
```

从文件对象 *file* 加载并返回一个对象的序列化表示形式。不必指定输入协议，因为它会自动检测协议。如果该文件包含无法解码的损毁数据，则引发pickle.UnpicklingError异常。如果检测到文件结尾，则引发EOFError异常。

```
loads(string

)
```

与load()相同，但是从字符串读取一个对象的序列化表示形式。

下列例子显示如何使用这些函数加载数据：

```
f = open('myfile', 'rb')
x = pickle.load(f)
y = pickle.load(f)
... load more objects

 ...
f.close()
```

进行加载时，不必指定该协议或有关要加载对象类型的任何信息。此类信息都保存为序列化数据格式本身的一部分。

如果要序列化一个以上的Python对象，只需如前面的例子所示那样重复调用dump()和load()。进行多次调用时，只要确保load()调用顺序与用来写文件的dump()调用顺序一致即可。

使用涉及循环或共享引用的复杂数据结构时，使用dump()和load()可能有问题，因为它们不会保留已经被序列化或已还原的对象的任何内部状态。这样可能导致输出文件过于庞大，并且加载多个对象时不能正确地还原对象之间的关系。替代方法就是使用Pickler和Unpickler对象。

```
Pickler(file
 [, protocol
 ])
```

创建序列化对象，利用指定的序列化 *protocol* 将数据写到文件对象 *file*。Pickler的实例p有一个方法p.dump(*x*)，可将对象x转储到 *file* 。在将 *x* 转储之后，就会记住它的身份。如果接下来使用p.dump()操作写入相同的对象，则保存前面已转储对象的引用，而不是写新副本。方法p.clear_memo()清除用于跟踪前面已转储对象的内部字典。如果想要写一个前面已转储对象的新副本（也就是说，如果在最后一次dump()操作后它的值改变了），就会这样使用。

```
Unpickler(file
)
```

创建一个反序列化对象，它从文件对象 *file* 中读取数据。Unpickler的实例u有一个方法u.load()，它从 *file* 加载并返回一个新对象。Unpickler跟踪其返回的对象，因为输入源可能包含由Pickler对象创建的对象引用。在这种情况下，u.load()返回一个前面已加载对象的引用。

pickle模块适用于多种一般Python对象。包括：

- None；
  数字和字符串；

- 元组、列表和只包含可序列化对象的字典；
- 在模块顶层定义的用户定义类的实例。

序列化用户定义类的实例时，该实例数据就是被序列化的唯一部分。不会保存相应的类定义——相反，序列化后的数据只包含相关类和模块的名称。反序列化这些实例时，按照重建实例时访问类定义的顺序来导入定义了该类的模块。还应注意，还原一个实例时，不会调用类方法`__init__()`。而是通过其他方式和还原的实例数据来重建该实例。

对实例的限制是相应的类定义必须放在模块顶层（即没有嵌套类）。另外，如果该实例的类定义最初定义在`__main__`中，那么在反序列化已保存的对象之前必须手动重新加载这个类定义，因为在反序列化时，解释器无法知道如何将必要的类定义自动载回到`__main__`中。

通常不必再做什么就可以将用户定义类与pickle一起使用。但是，类可以通过实现特殊方法`__getstate()`和`__setstate__()`来定义可保存和还原其状态的自定义方法。`__getstate__()`方法必须返回表示对象状态的可序列化对象（例如字符串或元组）。`__setstate__()`方法接受已序列化的对象并还原其状态。如果没有定义这些方法，则默认行为是序列化实例的底层`__dict__`属性。应注意如果定义了这些方法，那么`copy`模块也将使用它们实现深浅复制操作。

注意

- 在Python2中，有一个称为`cPickle`的模块，它是在`pickle`模块中的函数的C实现。它比`pickle`要快得多，但制约条件是它不允许子类化`Pickler`和`Unpickle`对象。Python 3也有一个包含C实现的支持模块，但是可以更透明地使用它（`pickle`会在合适时自动使用它）。
- `pickle`使用的数据格式是Python专有的，并且不与像XML这样的任何外部标准兼容。
- 只要可能，就应当使用`pickle`模块而不是`marshal`模块，因为`pickle`更灵活，它可以将数据编码文档化并进行额外的错误检查。
- 出于安全考虑，程序不应反序列化从不信任源收到的数据。
- `pickle`模块和外部模块中定义的类型如何一起使用的相关内容比这里所述的要多。外部类型的实现应当查询联机文档，了解有关使这些对象与`pickle`一起使用时所需的底层协议——特别是关于如何实现`__reduce()`和`__reduce_ex__()`特殊方法的细节，`pickle`使用这些方法创建序列化的字节序列。

13.7 SYS

`sys`模块包含了与解释器及其环境操作有关的变量和函数。

13.7.1 变量

已定义下列变量。

api_version

表示Python解释器的C语言版本API的整数。使用扩展模块时使用该变量。

argv

传递给程序的命令行选项列表。`argv[0]` 是程序名。

builtin_module_names

包含内置于Python可执行文件中模块名的元组。

byteorder

机器的本机字节排序方式——`'little'` 表示小尾，`'big'` 表示大尾。

copyright

包含版权信息的字符串。

__displayhook__

`displayhook()` 函数的初始值。

dont_write_bytecode

导入模块时决定Python是否写字节码（.pyc 或.pyo 文件）的布尔标志。初始值是True，除非指定了解释器的-B 选项。在自己的程序中可以根据需要修改该设置。

dllhandle

针对Python DLL（Windows）的整数句柄。

__excepthook__

excepthook() 函数的初始值。

exec_prefix

安装了平台相关的Python文件的目录。

executable

包含解释器可执行文件名的字符串。

flags

表示提供给Python解释器的不同命令行选项设置的对象。下表列出了flags 的属性以及将该标志打开的对应命令行选项。这些属性都是只读的。

属　　性	命令行选项
flags.debug	-d
flags.py3k_warning	-3
flags.division_warning	-Q
flags.division_new	-Qnew
flags.inspect	-i
flags.interactive	-i
flags.optimize	-O 或-OO
flags.dont_write_bytecode	-B
flags.no_site	-S
flags.ignore_environment	-E
flags.tabcheck	-t 或-tt
flags.verbose	-v
flags.unicode	-U

float_info

保存有关浮点数内部表示信息的对象。这些属性的值从float.h 的C头文件中取得。

属　　性	说　　明
float_info.epsilon	1.0与下一个最大浮点数之差

float_info.dig	四舍五入之后无需修改就可以表示的十进制位数
float_info.mant_dig	使用float_info.radix 中指定的基数所能够表示的位数
float_info.max	最大浮点数
float_info.max_exp	在float_info.radix 中指定基数的最大指数
float_info.max_10_exp	基数为10的最大指数
float_info.min	最小正浮点值
float_info.min_exp	在float_info.radix 中指定基数的最小指数
float_info.min_10_exp	基数为10的最小指数
float_info.radix	指数所用的基数
float_info.rounds	四舍五入的方式（Ḥ1表示未定，0表示朝向0，1表示最近，2表示正无限，3表示负无限）

hexversion

对sys.version_info中包含的版本信息进行编码后使用十六进制表示的整数。该整数值总是随着更新的解释器版本而增大。

last_type, last_value, last_traceback

遇到未处理的异常以及解释器输出一条错误消息时设置这些变量。last_type 是最后一个异常的类型，last_value 是最后一个异常的值，而last_traceback 是栈追踪。注意，使用这些变量并不是线程安全的，要想保证线程安全应使用sys.exc_info() 。

maxint

整数类型支持的最大整数（仅在Python 2中使用）。

maxsize

系统中C的`size_t` 数据类型支持的最大整数值。该值决定了字符串、列表、字典和其他内置类型的最大长度。

maxunicode

表明能够表示的最大Unicode码的整数。16位UCS-2编码的默认值是65 535。如果将Python配置为使用UCS-4，就会得到更大的值。

modules

将模块名与模块对象进行匹配的字典。

path

指定模块搜索路径的字符串列表。第一项通常设为可启动Python的脚本所在的目录（如果可用的话）。参见第8章。

platform

平台标识符字符串，如'linux-i386'。

prefix

安装平台无关Python文件的目录。

ps1, ps2

包含解释器的主要和次要提示符文本的字符串。最初ps1 设为'>>>'，ps2 设为'...'。将这些值分派到任意对象的str() 方法通过求值来得到提示文本。

py3kwarning

在Python 2中，使用-3 选项运行解释器时被设为True 的标志。

stdin, stdout, stderr

与标准输入、标准输出和标准错误对应的文件对象。stdin 用于raw_input() 和input() 函数，stdout 用于print 和raw_input() 与input() 的提示。stderr 用于解释器的提示和错误消息。可以将这些变量分派给任何支持对单个字符串参数进行处理的write() 方法的对象。

__stdin__, __stdout__, __stderr__

在解释器启动时包含stdin 、stdout 和stderr 值的文件对象。

tracebacklimit

发生未处理的异常时，打印输出的回溯信息的最大级数。默认值是1 000。0值将禁止所有回溯信息，只打印输出异常类型和值。

version

版本字符串。

version_info

表示为元组(*major* , *minor* , *micro* , *releaselevel* , *serial*)的版本信息。除 *releaselevel* 外，所有值都是整数， *releaselevel* 是字符串'alpha' 、'beta' 、'candidate' 或'final' 。

warnoptions

使用命令行选项-W 提供给解释器的警告选项列表。

winver

用于在Windows上建立注册表键的版本号。

13.7.2 函数

可以使用下列函数。

```
_clear_type_cache()
```

清除内部类型缓存。为了优化方法查询，在解释器内部保留了1024项最近所用方法的缓存。该缓存加快了对重复方法的查询——特别是在具有深层继承层次结构的代码中。通常不必清空该缓存，但是如果要跟踪真正复杂的内存引用计数问题，就可以这么做。例如，如果缓存中的方法要保留准备销毁的对象的引用时。

```
_current_frames()
```

在调用时返回将线程标识符映射到执行线程的顶层栈帧的字典。编写有关线程调试的工具（即跟踪死锁）时该信息很有用。记住，该函数返回的值只代表在调用时解释器的快照。查看返回的数据时线程可能在其他地方执行。

```
displayhook([value])
```

解释器以交互模式运行时，调用该函数会打印表达式的结果。默认情况下，repr(*value*)的值打印到标准输出，而 *value* 保持在变量__builtin__ 中。如果需要就可以重新定义_.displayhook 来实现不同的处理。

```
excepthook(type
,value
,traceback
)
```

发生未捕获的异常时调用该函数。 *type* 是异常类， *value* 是raise 语句提供的值，而 *traceback* 是回溯对象。默认行为是打印该异常和回溯到标准错误。不过，可以重新定义该函数来实现另一种处理未捕获异常的方式（在诸如调试器或CGI脚本这样专门的应用程序中可能会用到）。

```
```

```
exc_clear()
```

清除有关最后一次发生异常的全部信息。它只清除调用线程特有的信息。

```
exc_info()
```

返回元组(*type* , *value* , *traceback*)，它包含当前要处理的异常信息。 *type* 是异常类型， *value* 是传递的异常参数，而 *traceback* 是回溯对象，包含了发生异常的调用栈。如果当前没有要处理的异常，则返回None 。

```
exit([n
])
```

通过引发SystemExit 异常来退出Python。 *n* 是一个表示状态码的整数退出码。0值表示正常（默认值）；非零值表示异常。如果 *n* 指定为一个非整数值，则将它打印到sys.stderr 并使用退出码1。

```
getcheckinterval()
```

返回检查间隔值，它设置解释器多久检查信号、线程开关和其他定期的事件。

```
getdefaultencoding()
```

得到按照Unicode惯例进行编码的默认字符串。返回像'ascii' 或'utf-8' 这样的值。site 模块设置默认编码。

```
getdlopenflags()
```

返回标志参数，在UNIX上加载扩展模块时将该参数提供给C函数dlopen() 。参见dl模块。

getfilesystemencoding()

返回的字符编码用于将Unicode文件名映射到底层操作系统使用的文件名。对于Windows返回'mbcs' ，或对于Macintosh OS X返回'utf-8' 。在UNIX系统上，编码取决于区域设置并将返回区域参数CODESET 的值。在使用默认编码的系统上可能返回None 。

**_getframe([*depth*

])

返回来自调用栈的帧对象。如果省略depth 或其为0，则返回顶层帧。否则，返回当前帧之下用于多次调用的帧。例如， _getframe(1) 返回调用程序的帧。如果depth 无效，则引发ValueError 。

getprofile()

返回由setprofile() 函数设置的探查函数。

getrecursionlimit()

返回函数的递归限制。

getrefcount(*object*

```
)
```

返回 *object* 的引用计数。

```
getsizeof(object
 [, default
])
```

返回 *object* 大小，以字节为单位。调用 *object* 的__sizeof__()特殊方法就可以进行该计算。如果还未定义，则生成TypeError，除非 *default* 参数指定了默认值。因为根据需要对象可以任意定义__sizeof__()，所以不能保证该函数的结果能真正测量出内存的使用量。不过，对于列表或字符串等内置类型来说，结果是准确的。

```
gettrace()
```

返回由settrace()函数设置的跟踪函数。

```
getwindowsversion()
```

返回一个元组(*major* ,*minor* ,*build* ,*platform* ,*text*)，它说明正使用的Windows版本。 *major* 是主版本号。例如，值为4则表明Windows NT 4.0，值为5则表示Windows 2000和Windows XP系列。 *minor* 是次版本号。例如，0表示Windows 2000，而1表示Windows XP。 *build* 是Windows构建号。 *Platform* 标识平台，它是一个整数，可以取以下常用值之一：0（Windows 3.1上的Win32）、1（Windows 95、98或Me）、2（Windows NT、2000、XP）或3（Windows CE）。 *text* 是包含像Service Pack 3这样额外信息的字符串。

```
setcheckinterval(n
)
```

在解释器检查诸如信号和线程上下文开关这样的定期事件之前，设置解释器必须执行的Python虚拟机指令数。默认值是10。

```
setdefaultencoding(enc
)
```

设置默认编码。 *enc* 是诸如'ascii'或'utf-8'这样的字符串。该函数只定义在site 模块内部。可以从用户可定义的sitecustomize 模块中调用它。

```
setdlopenflags(flags
)
```

设置传递到C的dlopen() 函数的标志，用于在UNIX上加载扩展模块。这将影响库与其他扩展模块之间解析符号的方式。 *flags* 是在dl 模块中可以得到的按位或值（参见第19章），如sys. setdlopenflags(dl.RTLD_NOW | dl.RTLD_GLOBAL) 。

```
setprofile(pfunc
)
```

设置系统探查函数，用于实现源代码探查程序。

```
setrecursionlimit(n
)
```

改变函数的递归限制。默认值是1 000。注意，操作系统可能严格限定栈大小，因此

设置这么高的值可能引起Python解释器进程崩溃，产生段错误（Segmentation Fault）或非法访问。

```
settrace(tfunc
)
```

设置系统跟踪函数，用于实现调试器。参见第11章，了解有关Python调试器的信息。

13.8 traceback

traceback 模块在发生异常后回收并打印程序的栈跟踪信息。该模块中的函数对诸如sys.exc_info() 函数返回的第三项这样的回溯对象进行操作。该模块主要用在需要利用非标准方式报告错误的代码中——例如，如果要运行深嵌在网络服务器内的Python程序并且希望将回溯重定向到日志文件时。

```
print_tb(traceback
 [, limit
 [, file
]])
```

从 *traceback* 将 *limit* 栈跟踪项打印到文件 *file* 。如果省略 *limit* ，则打印全部项。如果省略 *file* ，则将输出发送到sys.stderr 。

```
print_exception(type
, value
, traceback
 [, limit
 [, file
]])
```

将异常信息和栈跟踪打印到 *file* 。 *type* 是异常类型， *value* 是异常值。 *limit* 和 *file* 与print_tb()中的参数含义相同。

```
print_exc([limit
 [, file
]])
```

与print_exception()相同，适用于sys.exc_info()函数返回的信息。

```
format_exc([limit
 [, file
]])
```

返回一个字符串，它包含由print_exc()打印的相同信息。

```
print_last([limit
 [, file
]])
```

等同于print_exception(sys.last_type, sys.last_value, sys.last_traceback, *limit* , *file*)。

```
print_stack([frame
 [, limit
 [, file
]]])
```

从调用堆栈跟踪处将其打印输出。 *frame* 设置可选的从何处开始的栈帧。 *limit* 和 *file* 与print_tb() 中的参数含义相同。

```
extract_tb(traceback
 [, limit
])
```

提取print_tb() 使用的栈跟踪信息。返回值是(*filename* , *line* , *funcname* , *text*) 形式的元组列表，它包含堆栈跟踪中常出现的相同信息。limit 是要返回的项数。

```
extract_stack([frame
 [, limit
]])
```

提取print_stack() 使用的相同栈跟踪信息，不过是从栈帧 *frame* 中得到。如果省略 *frame* ，则使用调用程序的当前栈帧， *limit* 是要返回的项数。

```
format_list(list
)
```

为了进行打印而格式化栈跟踪信息。 *list* 是extrac_tb() 或extract_stack() 返回的元组列表。

```
format_exception_only(type
, value
)
```

为了进行打印而格式化异常信息。

```
format_exception(type
, value
, traceback
 [, limit
])
```

为了进行打印而格式化异常信息和栈跟踪。

```
format_tb(traceback [, limit
])
```

等同于format_list(extract_tb(*traceback* , *limit*))。

```
format_stack([frame [, limit
]])
```

等同于format_list(extract_stack(*frame* , *limit*))。

```
tb_lineno(traceback
)
```

返回回溯对象中设置的行号。

13.9 types

types模块定义了与函数、模块、生成器、栈帧和其他程序元素对应的内置类型名称。该模块的内容通常与内置的`isinstance()`函数和其他与类型有关的操作一起使用。

变　量	说　明
BuiltinFunctionType	内置函数类型
CodeType	代码对象类型
FrameType	执行帧对象类型
FunctionType	用户定义的函数和lambda类型
GeneratorType	生成器—迭代器对象类型
GetSetDescriptorType	getset描述符对象类型
LambdaType	FunctionType的另一个名称
MemberDescriptorType	成员描述符对象类型
MethodType	用户定义类方法类型
ModuleType	模块类型
TracebackType	追踪对象类型

大多数上述类型的对象都用作构造函数，可以用来创建该类型的对象。以下说明提供了用于创建函数、模块、代码对象和方法的参数。第3章包含有关创建对象的属性和参数的详细信息，这些参数将提供给下面介绍的函数。

```
FunctionType(code
, globals
 [, name
 [, defarags
 [, closure
]]])
```

```

```

创建新函数对象。

```
CodeType(argcount
, nlocals
, stacksize
, flags
, codestring
, constants
, names
,

varnames
, filename
, name
, firstlineno
, lnotab
 [, freevars
 [, cellvars
 ]])
```

创建新代码对象。

```
MethodType(function
, instance
, class
)
```

创建新边界实例方法。

```
ModuleType(name
 [, doc
])
```

创建新模块对象。

注意

- 不应使用types 模块引用诸如整数、列表或字典这样的内置对象类型。在Python 2 中，types 包含像IntType 和DictType 这样的其他名称。不过，这些名称只是int 和dict 内置类型名称的别名。在当今的代码中，应仅使用内置类型名称，因为在Python 3中，types 模块只包含前面列出的名称。

13.10 warnings

warnings 模块提供的函数用于发出和筛选警告信息。与异常不同，警告用于向用户报告潜在问题，但是不会生成异常或导致执行停止。警告模块的主要用途之一是通知用户那些Python建议不使用的特性，在Python以后的版本中可能不会支持这些特性。例如：

```
>>> import regex


__main__:1: DeprecationWarning: the regex module is deprecated; use the re
module
>>>
```

与异常类似的是，各种警告被划分为一种类层次结构，它说明警告的一般类别。下面列出目前支持的类别：

警告对象	说　　明
Warning	所有警告类型的基类
UserWarning	用户定义的警告

DeprecationWarning	对使用不支持特性的警告
SyntaxWarning	潜在的语法问题
RuntimeWarning	潜在的运行时问题
FutureWarning	警告特定功能的语义在以后的版本中将改变

在__builtin__ 模块以及exceptions 模块中可使用这些类。此外，它们也是Exception 的实例。这样就可以很容易地将警告转换为错误。

使用warn() 函数来发出警告。例如：

```
warnings.warn("feature X is deprecated.")
warnings.warn("feature Y might be broken.", RuntimeWarning)
```

如果需要，可以筛选警告。筛选过程可以用于改变警告消息的输出行为，或者将警告转换为异常。filterwarnings() 函数用于为特定的警告类型添加筛选器。例如：

```
warnings.filterwarnings(action="ignore",
                        message=".*regex.*",
                        category=DeprecationWarning)
import regex       # 警告信息消失
```

使用解释器的-W 选项也可以设置限定的筛选形式。例如：

```
% python –Wignore:the\ regex:DeprecationWarning
```

在warnings 模块中定义了以下函数：

```
warn(message
[, category
[, stacklevel
]])
```

发出一个警告。 *message* 是一个包含警告信息的字符串， *category* 是警告类（例如Deprecation-Warning），而 *stacklevel* 是一个整数，它设置应当引发该警告信息的栈帧。默认情况下， *category* 是UserWarning，而 *stacklevel* 是1。

```
warn_explicit(message
, category
, filename
, lineno
[, module
[, registry
]])
```

这是warn()函数的底层版本。 *message* 和 *category* 的含义与warn()中的相同。 *filename* 、 *lineno* 和 *module* 显式设置警告的位置。 *registry* 是表示当前所有活动筛选器的对象。如果省略 *registry* ，则不会禁止警告消息。

```
showwarning(message
, category
, filename
, lineno
[, file
])
```

将警告写到文件。如果省略 *file* ，则将警告打印到sys.stderr。

```
formatwarning(message
, category
```

```
, filename
, lineno
)
```

创建格式化字符串，发出警告时打印该字符串。

```
filterwarnings(action
[, message
[, category
[, module
[, lineno
[, append
]]]]])
```

向警告筛选器列表添加一项。 *action* 可以是'error'、'ignore'、'always'、'default'、'once'或'module'。以下列表对每一项进行说明。

操　作	说　明
'error'	将该警告转换为异常
'ignore'	忽略该警告
'always'	总是打印警告消息
'default'	针对产生警告的每个位置打印警告一次
'module'	针对发生警告的每个模块打印警告一次
'once'	无论哪里发生警告都打印警告一次

message 是可匹配警告消息的正则表达式字符串。 *category* 是诸如DeprecationError 这样的警告类。 *module* 是匹配模块名的正则表达式字符串。 *lineno* 是具体行号或0，0表示匹配所有行。 *append* 设置应当将该筛选器添加到所有筛选器的列表（检查最后一个）。默认情况下，将新筛选器添加到筛选器列表的开始位置。如果省略所有参数，则默认值为匹配所有警告。

```
resetwarnings()
```

重置所有警告筛选器。这样会丢弃所有以前对filterwarnings() 的调用以及使用-W 设置的选项。

注意

- 在变量warnings.filters 中可以找到当前活动的筛选器列表。
- 警告转换为异常时，警告类别变为异常类型。例如，在DeprecationWarning 上的错误将引发DeprecationWarning 异常。
- -W 选项可用于在命令行上设置警告筛选器。该选项的一般语法是：-Waction:message:category:module:lineno
- 其中每个部分的含义都与filterwarnings() 函数中的相同。但是在这种情况下，*message* 和 *module* 字段分别设置警告消息第一部分的子字符串（而不是正则表达式）以及要筛选的模块名。

13.11 weakref

weakref 用于支持弱引用。通常情况下，引用对象会使其引用计数增加——这可有效地保持对象处于活动状态直到引用结束。另一方面，弱引用提供了一种不增加其引用计数而引用对象的方式。某些种类的应用程序必须通过特殊方式管理对象时，可以使用这种方式。例如，在面向对象的程序中可能实现诸如Observer模式这样的关系，那么可以使用弱引用避免创建引用循环。在7.10节中展示了这样一个例子。

使用weakref.ref() 函数创建弱引用的方法如下所示：

```
>>> class A: pass

>>> a = A()

>>> ar = weakref.ref(a)
       #
 创建对a
的一个弱引用
```

```
>>> print ar


<weakref at 0x135a24; to 'instance' at 0x12ce0c>
```

在创建了弱引用之后，只要将其作为没有参数的函数来调用就可以从弱引用得到原始对象。如果底层对象仍然存在，则将其返回。否则返回None，表明原始对象不再存在。例如：

```
>>> print ar()

                #

打印原始对象
<__main__.A instance at 12ce0c>
>>> del a

                      #

删除原始对象
>>> print ar()

                # a

已经消失，所以现在返回None


None
>>>
```

在weakref 模块中定义了以下函数：

```
ref(object

[, callback

])
```

创建 *object* 的一个弱引用。 *callback* 是可选函数，准备销毁 *object* 时将调用它。如果提供了该选项，则该函数应当只接受一个与弱引用对象对应的参数。一个以上的弱引用可能引用相同的对象。在这种情况下，将按照从最新应用的引用到最早引用的顺序调用 *callback* 函数。将返回的弱引用对象作为没有参数的函数进行调用就可以从弱引用得到 *object* 。如果原始对象不再存在，则返回None 。ref() 实际定义了一个类型 ReferenceType，可用于类型检查和子类。

```
proxy(object
[, callback
])
```

使用对 *object* 的弱引用创建一个代理。返回的代理对象实际上是关于原始对象的包装器，它提供了对其属性和方法的访问。只要原始对象存在，那么对代理对象的操纵将模仿底层对象的行为。另一方面，如果原始对象已被销毁，那么对代理对象的操作将引发 weakref.ReferenceError，表明该对象不再存在。 *callback* 是一个回调函数，与ref() 函数中的回调函数含义相同。代理对象的类型既可以是ProxyType 也可以是CallableProxyType，这取决于原始对象是否可以调用。

```
getweakrefcount(object
)
```

返回引用 *object* 的弱引用数目和代理数。

```
getweakrefs(object
)
```

返回全部引用 *object* 的弱引用和代理对象的列表。

```
WeakKeyDictionary([dict
])
```

创建一个字典，对其中的键进行弱引用。对键没有更多的强引用时，会自动删除该字典中的对应项。如果提供 *dict* ，则 *dict* 中的各项最初都被添加到返回的 WeakKeyDictionary 对象。因为只能弱引用某些对象类型，所以对于可接受的键值有很多限制。特别是，内置字符串不用作弱键。但是，用户定义类的实例中定义了

__hash__() 方法就可以用作键。WeakKeyDictionary 实例有两个返回弱键引用的方法：iterkeyrefs() 和keyrefs() 。

```
WeakValueDictionary([dict

])
```

创建一个字典，对其中的值进行弱引用。对值没有更多的强引用时，将丢弃该字典中的对应项。如果提供 *dict* ，则将 *dict* 中的各项添加到返回的 WeakValueDictionary 。WeakValueDictionary 的实例有两个返回弱值引用的方法：itervaluerefs() 和valuerefs() 。

```
ProxyTypes
```

这是一个元组(ProxyType, CallableProxyType)，可用于测试对象是否为proxy() 函数创建的两种代理对象之一。例如，isinstance(*object , ProxyTypes*)。

13.11.1 示例

使用弱引用的应用程序可以创建最近计算结果的缓存。例如，如果函数要花大量时间计算一个结果，那么缓存这些结果并在应用程序的其他地方使用它们时，对其进行重用就会很有意义。例如：

```
_resultcache = { }
def foocache(x):
     if resultcache.has_key(x):
          r = _resultcache[x]()       # 获得弱引用并废除该引用
          if r is not None: return r
     r = foo(x)
     _resultcache[x] = weakref.ref(r)
     return r
```

13.11.2 注意

- 只有类实例、函数、方法、集合、冻结集合、文件、生成器、类型对象以及库模块中定义的某些对象类型（例如套接字、数组和正则表达式模型）支持弱引用。不能使用内置函数以及像列表、字典、字符串和数字这样的内置类型。
- 如果曾经对WeakKeyDictonary 或WeakValueDictionary 使用了迭代，那么必须特别注意要确保该字典不会改变大小，因为这样可能产生奇怪的副作用，例如某些项不

知什么原因从字典中神秘消失。

- 如果使用`ref()`或`proxy()`注册的回调函数在执行过程中发生异常，则将该异常打印到标准错误并予以忽略。
- 只要原始对象是可散列的，则弱引用就是可散列的。而且只要在对象仍然存在时计算了原始散列值，那么在删除原始对象之后该弱引用将保持其散列值。
- 可以测试弱引用是否相等，但是不能进行排序。如果这些对象仍然存在，那么当底层对象具有相同值时，这些引用相等。否则，只有具有相同引用时这些引用才相等。

第14章　数学运算

本章介绍可执行各种数学运算操作的模块。此外还会介绍decimal 模块，该模块提供了对十进制浮点数的一般支持。

14.1　decimal

Python使用双精度二进制浮点编码表示float 数据类型（通常与IEEE 754标准定义的一样）。该编码方式的微妙结果就是，不能准确表示像0.1这样的十进制值。而最接近的值是0.10000000000000001。这种不精确性会影响浮点数计算，有时可能导致意想不到的结果（例如，3 * 0.1 == 0.3的值为False ）。

decimal 模块实现了IBM通用十进制算法标准，它能够准确表示十进制值。它还能够精确控制运算精度、有效数位和四舍五入操作。如果与精确定义了十进制数属性的外部系统进行交互，那么这些功能就很有用。例如，在编写必须与商业应用程序进行交互的Python程序时。

decimal 模块定义了两种基本数据类型：一种表示十进制数的Decimal 类型，一种表示与精度和四舍五入错误处理等计算有关的各种参数的Context 类型。下面是一些简单的例子，说明了该模块的基本工作原理：

```
import decimal
x = decimal.Decimal('3.4')     # 创建一些十进制数
y = decimal.Decimal('4.5')

# 使用默认环境执行一些数学计算
a = x * y         #  a = decimal.Decimal('15.30')
b = x / y         #  b = decimal.Decimal('0.75555555555555555555555555556')

# 改变精度并执行计算
decimal.getcontext().prec = 3
c = x * y         #  c = decimal.Decimal('15.3')
d = x / y         #  d = decimal.Decimal('0.756')

# 只改变某个语句块的精度
with decimal.localcontext(decimal.Context(prec=10)):
    e = x * y    #  e = decimal.Decimal('15.30')
    f = x / y    #  f = decimal.Decimal('0.7555555556')
```

14.1.1　Decimal 对象

通过以下类表示Decimal 数：

```
Decimal([value

 [, context
```

```
]])
```

value 是数字的值，既可以是一个整数，也可以是一个包含像'4.5' 这样十进制值的字符串或元组(*sign* , *digits* , *exponent*)。如果给定一个元组，则正数的 *sign* 是0，而负数的是1； *digits* 是指定为整数的数字元组；而 *exponent* 是整数指数。特殊字符串'Infinity' 、'-Infinity' 、'NaN' 和'sNaN' 可用于表示正无穷和负无穷以及非数值（NaN）。'sNaN' 是另一种形式的NaN，如果在之后的计算中使用了它，则会导致异常。普通的float 对象不能用作初始值，因为该值可能不准确（违背了一开始就使用decimal 的目的）。参数context 是稍后将介绍的Context 对象。如果提供的话，context 会确定在初始值不是有效值时会发生什么情况——引起异常或返回值为NaN的十进制值。

以下例子说明了如何创建各种十进制数值：

```
a = decimal.Decimal(42)                  # 创建十进制("42")
b = decimal.Decimal("37.45")             # 创建十进制("37.45")
c = decimal.Decimal((1,(2,3,4,5),-2))   # 创建十进制("-23.45")
d = decimal.Decimal("Infinity")
e = decimal.Decimal("NaN")
```

十进制对象都是不可变对象，并且具有所有内置int 和float 类型的常见数值属性。该对象也可以用作字典键、置于集合中、排序等。对于大多数情况，使用标准Python算术运算符处理Decimal 对象。不过，以下列表中的方法可用于执行多种常用数学操作。所有操作都采用可选的context 参数来控制精度、四舍五入和计算的其他方面。如果省略该参数，则使用当前上下文。

方　法	说　明
x .exp ([*context*])	自然指数e**d
x. fma(*y, z [, context])*	x*y + z ， x*y 指数不进行四舍五入
x. ln(*[context])*	x 的自然对数（基数e）
x. log10*([context])*	基数为10的x 的对数
x. sqrt*([context])*	x 的平方根

14.1.2 Context 对象

十进制数的各种属性，如四舍五入和精度，都通过Context 对象进行控制：

```
Context(prec
=None, rounding
=None, traps
=None, flags
=None,


Emin
=None, Emax
=None, capitals
=1)
```

这将创建了一个新的十进制上下文。应当使用具有所示名称的关键字参数设置这些参数。 *prec* 是整数，它设定算术运算的精度位数， *rounding* 确定四舍五入的方式，而 *traps* 是一个信号列表，计算过程中发生某些事件时（如除0）产生Python异常。 *flags* 是表示上下文初始状态（如溢出）的信号列表。通常并不指定 *flags* 。 *Emin* 和 *Emax* 是分别表示指数最小和最大范围的整数。 *capitals* 是表示指数使用E 还是e 的布尔标志。默认是1（'E'） 。

通常不会直接创建 *Context* 对象，而是用函数 *getcontext()* 或 *localcontext()* 返回当前活动的 *Context* 对象。然后根据需要修改该对象。本节稍后会介绍这种例子。但是，为了更好地理解这些例子，有必要更详细地介绍这些环境参数。

四舍五入的方式取决于rounding 参数设置的值。

常　　量	说　　明
ROUND_CEILING	向正无穷四舍五入。例如，2.52四舍五入到2.6，而-2.58四舍五入到-2.5
ROUND_DOWN	向0四舍五入。例如，2.58四舍五入到2.5，而-2.58四舍五入到-2.5
ROUND_FLOOR	向负无穷四舍五入。例如，2.58四舍五入到2.5，而-2.52四舍五入到-2.6
	如果小数部分大于一半，则向0的反方向四舍五入，反之则向0四舍五入。例如，2.58四舍

ROUND_HALF_DOWN	五入到2.6，2.55四舍五入到2.5。-2.55四舍五入到-2.5，而-2.58四舍五入到-2.6
ROUND_HALF_EVEN	与ROUND_HALF_DOWN 相同，除了小数部分正好是一半的情况之外，如果前面的数字是偶数则结果是朝下四舍五入，如果前面的数字是奇数，则向上四舍五入。例如，2.65四舍五入到2.6，而2.55四舍五入到2.6
ROUND_HALF_UP	与ROUND_HALF_DOWN 相同，除了小数部分正好是一半之外（向0的反方向四舍五入）。例如2.55四舍五入到2.6，而-2.55四舍五入到-2.6
ROUND_UP	向0的反方向四舍五入。例如，2.52四舍五入到2.6，而-2.52四舍五入到-2.6
ROUND_05UPD	如果趋向于0之后的最后一位数字为0或5，则向0的反方向四舍五入。否则向0进行四舍五入。例如，2.54四舍五入到2.6，而2.64四舍五入到2.6

Context() 的参数 *traps* 和 *flags* 都是信号列表。信号代表可能发生在计算过程中的数学异常类型。未列在 *traps* 中的信号都将被忽略。否则引发异常。定义了以下信号：

信　　号	说　　明
Clamped	对指数进行调整以适合允许的范围
DivisionByZero	非无限数值除0
Inexact	发生四舍五入错误
InvalidOperation	执行了非法操作
Overflow	四舍五入之后指数超出 *Emax* 。也生成Inexact 和Rounded
Rounded	产生四舍五入。即使没有丢失信息也可能发生（例如，1.00四舍五入到1.0）
Subnormal	在四舍五入之前指数小于Emin
Underflow	数值下溢。结果四舍五入到0。也生成Inexact 和Rounded

这些信号名称对应于可用来进行错误检查的Python异常，例如：

```
try:
      x = a/b
```

```
except decimal.DivisionByZero:
     print "Division by zero"
```

与异常一样，这些信号都按照一种层次结构来组织：

```
ArithmeticError (built-in exception)
    DecimalException
           Clamped
           DivisionByZero
    Inexact
           Overflow
           Underflow
    InvalidOperation
    Rounded
           Overflow
           Underflow
    Subnormal
           Underflow
```

Overflow 和Underflow 在表格中出现了不止一次，这是因为这些信号也会导致父信号（例如，Underflow 也会产生Subnormal 信号）。decimal.DivisionByZero 信号也派生自内置的Division- ByZero 异常。

在很多情况下都只是忽略数学信号。例如，一次计算可能产生一个舍位错误，但是没有生成异常。在这种情况下，这些信号名称可以用于检查一组表明计算状态的sticky 标志，例如：

```
ctxt = decimal.getcontext()       # 获得当前的上下文
x = a + b
if ctxt.flags[Rounded]:
    print "Result was rounded!"
```

设置这些标志后，在使用clear_flags() 方法将其清除之前它们会一直保持不变。因此，可以执行一系列完整的计算，并且只在结束时检查错误。

通过以下属性和方法可以修改现有Context 对象 *c* 的设置：

```
c

.capitals
```

设为1或0的标志，决定是否使用 *E* 或 *e* 作为指数字符。

```
```

```
c
.Emax
```

整数，设置最大指数。

```
c
.Emin
```

整数，设置最小指数。

```
c
.prec
```

整数，设置精度位数。

```
c
.flags
```

包含与信号对应的当前标志值的字典。例如，`c.flags[Rounded]` 返回Rounded 信号的当前标志值。

```
c
.rounding
```

正在使用的四舍五入规则，如ROUND_HALF_EVEN 。

```
c
.traps
```

该字典包含可导致Python异常的信号的True 或False 设置。例如， *c* .traps[DivisionByZero] 通常为True ， 而 *c* .traps[Rounded] 为False 。

```
c
.clear_flags()
```

重置所有sticky 标志（清除 *c* .flags ）。

```
c
.copy()
```

返回上下文 *c* 的一个副本。

```
c
.create_decimal(value
)
```

使用 *c* 作为上下文创建一个新的Decimal 对象。要生成的数值的精度和四舍五入操作覆盖默认上下文的设置时，这会比较有用。

14.1.3 函数和常量

下列函数和常量都是由decimal 模块定义的。

```
getcontext()
```

返回当前的十进制上下文。每个线程具有其自己的十进制上下文，因此这会返回调用线程的上下文。

```
localcontext([c

])
```

创建上下文管理器，它将当前十进制上下文设为with 语句体内定义的语句的 *c* 副本。如果省略 *c* ，则创建当前上下文的副本。下面是使用该函数的例子，它为一组语句临时设置5个十进制位精度：

```
with localcontext() as c:
    c.prec = 5
    statements
setcontext(c

)
```

将调用线程的十进制上下文设为 *c* 。

```
BasicContext
```

具有9位精度的预制上下文。四舍五入方式为ROUND_HALF_UP 。Emin 是-999999999 ；Emax 是999999999 ；并且除Inexact 、Rounded 和Subnormal 外启用所有限制

（trap）。

```
DefaultContext
```

创建新的上下文时使用的默认上下文（这里存储的值都用作新上下文的默认值）。定义了28位精度；ROUND_HALF_EVEN 四舍五入方式；以及Overflow 、InvalidOperation 和DivisionByZero 限制。

```
ExtendedContext
```

具有9位精度的预制上下文。四舍五入方式为ROUND_HALF_EVEN 。Emin 是-999999999 。Emax 是999999999 ， 并且禁用所有限制。永不引发异常，而是将结果设为NaN 或Infinity 。

```
Inf
```

与Decimal("Infinity") 相同。

```
negInf
```

与Decimal("-Infinity") 相同。

```
NaN
```

与Decimal("NaN") 相同。

14.1.4 示例

下面的示例展示了十进制数值的基本语法：

```
>>> a = Decimal("42.5")


>>> b = Decimal("37.1")


>>> a + b


Decimal("79.6")
>>> a / b


Decimal("1.145552560646900269541778976")
>>> divmod(a,b)


(Decimal("1"), Decimal("5.4"))
>>> max(a,b)


Decimal("42.5")
>>> c = [Decimal("4.5"), Decimal("3"), Decimal("1.23e3")]


>>> sum(c)


Decimal("1237.5")
>>> [10*x for x in c]


[Decimal("45.0"), Decimal("30"), Decimal("1.230e4")]
>>> float(a)


42.5
>>> str(a)


'42.5'
```

下面是改变环境参数的例子：

```
>>> getcontext().prec = 4


>>> a = Decimal("3.4562384105")


>>> a


Decimal("3.4562384105")
```

```
>>> b = Decimal("5.6273833")


>>> getcontext().flags[Rounded]


0
>>> a + b


9.084
>>> getcontext().flags[Rounded]


1
>>> a / Decimal("0")


Traceback (most recent call last):
File "<stdin>", line 1, in ?
decimal.DivisionByZero: x / 0
>>> getcontext().traps[DivisionByZero] = False


>>> a / Decimal("0")


Decimal("Infinity")
```

14.1.5 注意

- Decimal 和Context 对象具有大量与十进制操作的表示和方式有关的底层细节方法。此处并没有涉及这些内容，因为它们不属于该模块的基本用法。如果要了解详细内容可以查询联机文档：http://docs.python.org/library/decimal.html 。
- 每个线程都有一个唯一的十进制上下文。对该上下文的改变只影响线程本身而不会影响其他线程。
- 特殊数值Decimal("sNaN") 可用作使用信号的NaN。任何内置函数都不会生成该数值。但是，如果它出现在计算过程中，则总是使用信号通知错误。可以使用它表示必须对无效的计算生成错误，而不能简单地忽略掉。例如，函数可以返回sNaN 作为结果。
- 0值可能为正也可能为负（即Decimal(0) 和Decimal("-0") ）。不同的零值相比仍然相等。
- 由于计算中涉及的开销量很高，所以该模块可能不适用于高性能要求的科学计算。此外，在这样的应用程序中使用十进制浮点数代替二进制浮点数往往没有什么实用价值。
- 从数学上对浮点表示法和错误分析进行全面讨论超出了本书范围。你可以阅读有关数值分析的图书，了解更多详情。David Goldberg在Association for Computing Machinery出版的1991年3月的*Computing Surveys* 中发表的文章“What Every Computer Scientist Should Know About Floating-Point Arithmetic”也值得一读（在网上如果搜索该标题就会很容易找到这篇文章）。
- 《IBM通用十进制计算规范》包含更多信息，而且可以利用搜索引擎轻松地找到该内

容。

14.2 fractions

fractions 模块定义了表示有理数的Fraction 类。可以通过3种不同方式使用类构造函数创建实例。

```
Fraction([numerator
 [,denominator
]])
```

创建新的有理数。 *numerator* 和 *denominator* 是整数值，默认分别为0和1。

```
Fraction(fraction
)
```

如果 *fraction* 是numbers.Rational 的实例，则创建一个与 *fraction* 相同值的新有理数。

```
Fraction(s
)
```

如果 *s* 是包含诸如3/7 或-4/7 这样分数的字符串，则创建一个具有相同值的分数。如果s 是诸如1.25这样的十进制数值，则创建具有该值的分数（如Fraction(5,4) ）。

下列类方法可以根据其他对象类型创建Fraction 实例。

```
Fraction.from_float(f
)
```

创建一个分数，表示浮点数 *f* 的精确值。

```
Fraction.from_decimal(d
)
```

创建一个分数，表示Decimal 数 *d* 的精确值。

下面是一些使用这些函数的例子：

```
>>> f = fractions.Fraction(3,4)

>>> g = fractions.Fraction("1.75")

>>> g

Fraction(7, 4)
>>> h = fractions.Fraction.from_float(3.1415926)

Fraction(3537118815677477, 1125899906842624)
>>>
```

Fraction 的实例 *f* 支持所有普通的数学运算。分子和分母分别存储在 *f* .numerator 和 *f* .denomin- ator 属性中。此外，定义了以下方法。

```
f
.limit_denominator([max_denominator
])
```

返回的分数具有最接近 *f* .max_denominator 的值，它指定了要使用的最大分母，并且默认值是1000000。

下面是一些使用Fraction 实例的例子（使用前面例子中创建的值）：

```
>>> f + g


Fraction(5, 2)
>>> f * g


Fraction(21, 16)
>>> h.limit_denominator(10)


Fraction(22, 7)
>>>
```

fractions 模块还定义了一个单独的函数：

```
gcd(a

,b

)
```

计算整数 *a* 和 *b* 的最大公约数。如果 *b* 是非零数，则结果的符号与 *b* 相同；否则其符号与 *a* 相同。

14.3 math

math 模块定义了下列标准算术运算函数。这些函数适用于整数和浮点数，但是不适用于复数（可以使用单独的模块cmath 对复数执行类似操作）。所有函数的返回值都是浮点数。所有三角函数假定都使用弧度。

函　　数	说　　明
acos(*x*)	返回 *x* 的反余弦
acosh(*x*)	返回 *x* 的双曲线反余弦
asin(*x*)	返回 *x* 的反正弦
asinh(*x*)	返回 *x* 的双曲线反正弦
atan(*x*)	返回 *x* 的反正切

atan2(y, x)	返回atan(y/x)
atanh(x)	返回 x 的双曲线反正切
ceil(x)	返回 x 的向上舍入值
copysign(x,y)	返回与 y 具有相同符号的 x
cos(x)	返回 x 的余弦
cosh(x)	返回 x 的双曲线余弦
degrees(x)	将 x 从弧度转换为角度
radians(x)	将 x 从角度转换为弧度
exp(x)	返回e ** x
fabs(x)	返回 x 的绝对值
factorial(x)	返回 x 阶乘
floor(x)	返回 x 的向下舍入值
fmod(x, y)	返回 x % y ，与C的fmod() 函数计算的结果一样
frexp(x)	返回元组形式的 x 的正尾数和指数
fsum(s)	返回可迭代序列 s 中浮点值的完整精度之和。参见下面注意里的解释
hypot(x, y)	返回欧几里德距离，sqrt(x * x + y *y)
isinf(x)	如果 x 是正无穷，则返回True
isnan(x)	如果 x 是NaN，则返回True
ldexp(x, i)	返回x * (2 ** i)

log(*x [, base]*)	返回给定 *base* 的 *x* 的对数。如果省略 *base* ，则该函数计算自然对数
log10(*x*)	返回基数为10的 *x* 的对数
log1p(*x*)	返回1+ *x* 的自然对数
modf(*x*)	返回元组形式的 *x* 的小数和整数部分。它们与 *x* 的符号相同
pow(*x, y*)	返回 *x* ** *y*
sin(*x*)	返回 *x* 的正弦
sinh(*x*)	返回 *x* 的双曲线正弦
sqrt(*x*)	返回 *x* 的平方根
tan(*x*)	返回 *x* 的正切
tanh(*x*)	返回 *x* 的双曲线正切
trunc(*x*)	将 *x* 截为最接近于零的整数

定义了以下常量。

常　　量	说　　明
pi	数学常量pi
e	数学常量e

注意

- 将字符串传入`float()` 函数就可以创建浮点值`+inf` 、`-inf` 和`nan` ，如`float("inf")` 、`float ("-inf")` 或`float("nan")` 。
- `math.fsum()` 函数比内置的`sum()` 函数更精确，因为它使用一种不同的算法，尽量避免补偿效应带来的浮点错误。例如，考虑这个序列`s = [1, le100, -le100]` 。如果使用`sum(s)` ，则得到的结果是`0.0` （因为将1添加到更大值le100时丢失了1的值）。而使用`math.sum(s)` 则会产生正确的结果`1.0` 。在1996年的*Carnegie Mellon*

University School of Computer Science Technical Report CMU-CS-96-140中Jonathan Richard Shewchuk所著的*Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates* 介绍了`math.sum()` 使用的算法。

14.4 numbers

`numbers` 模块定义了一组抽象基类，用于管理各种数值。数值类按层次结构划分，随着层次的增长逐渐增加新的功能。

`Number`

作为数值层次结构顶层的类。

`Complex`

表示复数的类。该类型的数值具有`real` 和`imag` 属性。继承自`Number` 。

`Real`

表示实数的类。继承自`Complex` 。

`Rational`

表示分数的类。该类型的数值具有`numerator` 和`denominator` 属性。继承自`Real` 。

`Integral`

表示整数的类。继承自Rational 。

该模块中的类并不用于实例化，而是用于对值进行各种类型检查。例如：

```
if isinstance(x, numbers.Number)    # x是任何种类的数值
   statements



if isinstance(x, numbers.Integral)  # x是整数值


statements
```

如果其中一个类型检查返回True ，则表明 *x* 兼容所有与该类型有关的常见数学运算，并且支持转换到诸如complex() 、float() 或int() 这样的内置类型。

抽象基类也可以作为用户定义类的基类，用于模拟数值。这样做不仅是进行类型检查的好办法，而且增加了安全性检查，确保实现了所有的必需方法。例如：

```
>>> class Foo(numbers.Real): pass


...
>>> f = Foo()


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can′ t instantiate abstract class Foo with abstract methods
 __abs__, __add__, __div__, __eq__, __float__, __floordiv__, __le__, __lt__,
 __mod__, __mul__, __neg__, __pos__, __pow__, __radd__, __rdiv__,
 ➥__rfloordiv__,
 __rmod__, __rmul__, __rpow__, __rtruediv__, __truediv__, __trunc__
>>>
```

注意

- 参考第7章，了解有关抽象基类的更多信息。
- PEP 3141（http://www.python.org/dev/peps/pep-3141）包括有关该模块的类型层次结构和使用方面的更多信息。

14.5 random

Random 模块提供各种用于生成伪随机数的函数，以及根据不同的实数分布来随机生成值的函数。该模块中的多数函数取决于函数random() ，该函数使用Mersenne Twister生成器在[0.0, 1.0)范围内生成一致分布的数值。

14.5.1 种子和初始化

以下函数用于控制基础随机数生成器的状态。

```
seed([x
])
```

初始化随机数生成器。如果省略 *x* 或 *x* 为None，则使用系统时间来设置生成器。否则，如果 *x* 是整数或长整数，则使用该值。如果 *x* 不是整数，那么它必须是可散列的对象并且将hash(*x*) 的值作为种子。

```
getstate()
```

返回表示当前生成器状态的对象。稍后将该对象传递到setstate() 来恢复状态。

```
setstate(state
)
```

从getstate() 返回的对象中恢复随机数生成器的状态。

```
jumpahead(n
)
```

如果在一行中调用了 *n* 次random()，则快速将生成器状态改为其应有的状态。 *n* 必须是非负的整数。

14.5.2 随机整数

以下函数用于操作随机整数。

```
getrandbits(k
)
```

创建包含 *k* 个随机位的长整数。

```
randint(a
,b
)
```

返回随机整数 *x* ，范围是 *a* <= *x* <= *b* 。

```
randrange(start
,stop
,[,step
])
```

返回一个范围在(*start* ,*stop* ,*step*)之间的随机整数。不包括结束值。

14.5.3 随机序列

以下函数用于产生随机序列数据。

```
choice(seq
)
```

从非空序列 *seq* 中返回一个随机元素。

```
sample(s
,len
)
```

返回长度为 *len* 的序列，它包含从序列 *s* 中随机选择的元素。结果序列中的元素按照选择它们时的顺序排列。

```
shuffle(x
 [,random
])
```

随机原地打乱列表 *x* 中的项， *random* 是可选参数，它指定随机生成函数。如果提供该参数，则该参数不能是包含参数并且返回范围在[0.0, 1.0)内的浮点数的函数。

14.5.4 实值随机分布

以下函数生成实数的随机数。分布和参数名与概率和统计中使用的标准名称一致。需要查询其他相关内容来了解更多细节。

```
random()
```

返回范围在[0.0,1.0) 之间的随机数。

```
uniform(a
,b
)
```

返回范围在[a,b) 之间的一致分布随机数。

```
betavariate(alpha
,beta
)
```

从Beta分布中返回一个在0和1之间的值，其中alpha > -1 而beta > -1 。

```
cunifvariate(mean
,arc
)
```

圆形一致分布， *mean* 是平均角，而 *arc* 是沿平均角周围居中的分布范围。这些值都必须设置在0 到pi 之间的弧度范围内。返回值范围是(*mean* – *arc* /2, *mean* + *arc* /2) 。

```
expovariate(lambd
)
```

指数分布， *lambd* 是由1.0除以预期均值。返回值范围是[0,+∞) 。

```
gammavariate(alpha
, beta
)
```

Gamma分布，其中 *alpha* > -1 ， *beta* > 0 。

```
gauss（mu
, sigma
）
```

均值为 *mu* 且标准偏差为 *sigma* 的高斯分布。比normalvariate()稍快。

```
lognormvariate(mu
, sigma
)
```

对数正态分布。取该分布的自然对数的结果是均值为 *mu* 且标准偏差为 *sigma* 的正态分布。

```
normvariate(mu
,signma
)
```

均值为 *mu* 且标准偏差为 *sigma* 的正态分布。

```
paretovariate(alpha
)
```

形状参数为 *alpha* 的帕累托分布。

```
triangular([low
```

```
 [, high

 [, mode

]]])
```

三角分布。随机数 *n* 的范围是low <= n < high，且模式为 *mode* 。默认情况下， *low* 是0， *high* 是1.0，而 *mode* 设为 *low* 和 *high* 的中点值。

```
vonmisesvariate(mu

, kappa

)
```

von Mises分布，其中 *mu* 是平均角，弧度范围在0到2 * pi之间，而 *kappa* 是非负集中因子。如果 *kappa* 为零，则该分布简化为统一随机角，范围在0到2 * pi之上。

```
weibullvariate(alpha

, beta

)
```

Weibull分布，比例参数为 *alpha* ，形状参数为 *beta* 。

14.5.5 注意

- 该模块中的函数都不是线程安全的。如果要在不同线程中生成随机数，就应当使用锁定以防止并发访问。
- 随机数生成器的区间（在数值开始重复之前）是2**19937-1。
- 该模块生成的随机数都是确定的，不应用于密码。
- 通过实现random.Random 的子类并实现random()、seed()、getstate()、getstate() 和jumpahead() 方法，就可以创建新的随机数生成器类型。实际上，该模块中的所有其他函数在内部都作为Random 的方法实现的。因此，可以将它们作为新的随机数生成器实例的方法来访问。
- 该模块提供了两种供选择的随机数生成器类——WichmannHill 和SystemRandom ，

通过实例化正确的类并将前面的函数作为方法进行调用就可以使用。`WichmannHill` 类实现了早期Python发布版中所使用的Wichmann-Hill生成器。`SystemRandom` 类使用系统随机数生成器`os.urandom()` 生成随机数。

第15章　数据结构、算法与代码简化

本章介绍的各个模块旨在解决与数据结构、算法和代码简化（包括迭代、函数编程、上下文管理器和类）相关的常见编程问题。这些模块应看作是Python的内置类型和函数的扩展。在许多情况下，它们的底层实现具有很高的效率，而且能够比内置类型和函数更好地解决某些问题。

15.1　abc

abc 模块定义了一个元类和一对装饰器，用于定义新的抽象基类。

ABCMeta

这个元类表示抽象基类。要定义抽象类，需要定义使用ABCMeta 作为元类的类。例如：

```
import abc
class Stackable:                    # 在Python 3中使用该语法
   __metaclass__ = abc.ABCMeta      # class Stackable(metaclass=abc.ABCMETA)
   ...
```

通过这种方式创建的类与普通类的区别如下所述。

- 首先，如果抽象类定义了使用abstractmethod 和abstractproperty 装饰器（稍候将介绍）装饰的方法或特性，则无法创建派生类的实例，除非这些类提供了这些方法和特性的非抽象实现。
- 其次，抽象类具有一个类方法register(subclass)，可用于将附加类型注册为逻辑子类。对于使用此函数注册的任何subclass 类型，如果 *x* 是subclass 的实例，那么操作isinstance(*x* , *AbstractClass*) 都将返回True 。
- 抽象类的最后一个特征是，它们可以定义一个特殊类方法__subclasshook__(*cls* , *subclass*)。如果类型 *subclass* 被看作子类，此方法应该返回True ； 如果 *subclass* 不是子类，则返回False ； 如果不知道该类型的任何信息，则抛出NotImplemented 异常。

**abstractmethod(*method*

)**

此装饰器声明要抽象化的 *method* ，当在抽象基类中使用时，直接通过继承定义的派生类只有在定义了该方法的非抽象实现后才能实例化。此装饰器不会对使用抽象基类的register() 方法注册的子类产生任何影响。

```
abstractproperty(fget
 [, fset
 [, fdel
[, doc
]]])
```

创建一个抽象特性。它的参数与普通property() 函数相同，当在抽象基类中使用时，直接通过继承定义的派生类只有在定义了该方法的非抽象实现时才能实例化。

下面的代码定义简单抽象类：

```
from abc import ABCMeta, abstractmethod, abstractproperty
class Stackable:                  # 在Python 3中使用该语法
    __metaclass__ = ABCMeta             # class Stackable(metaclass=ABCMeta)
    @abstractmethod
    def push(self,item):
        pass
    @abstractmethod
    def pop(self):
        pass
    @abstractproperty
    def size(self):
        pass
```

下面是一个派生自Stackable 的类的例子：

```
class Stack(Stackable):
    def __init__(self):
        self.items = []
    def push(self,item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
```

下面是在尝试创建Stack 时得到的错误消息：

```
>>> s = Stack()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Stack with abstract methods size
>>>
```

此错误可通过将size() 特性添加到Stack 来修复。可以采用两种方法添加特性：修改Stack 本身的定义，或者从其继承并添加需要的方法或特性。

```
class CompleteStack(Stack):
    @property
    def size(self):
        return len(self.items)
```

下面给出了一个使用完整栈对象的例子：

```
>>> s = CompleteStack()


>>> s.push("foo")


>>> s.size


1
>>>
```

另请参见： 第7章、14.4节和15.4节。

15.2 array

array 模块定义一个新对象类型array ，该类型的工作原理与列表极其相似，但它的内容仅限于单一类型。数组的类型在创建数组时确定，可使用表15-1中的任意一种类型的编码。

表**15-1** 类型编码

类型编码	描　　述	**C**类型	最小大小（字节）
'c'	8位字符	Char	1
'b'	8位整型	signed char	1

'B'	8位无符号整型	unsigned char	1
'u'	Unicode字符	PY_UNICODE	2或4
'h'	16位整型	short	2
'H'	16位无符号整型	unsigned short	2
'i'	整型	int	4或8
'I'	无符号整型	unsigned int	4或8
'l'	长整型	long	4或8
'L'	无符号长整型	unsigned long	4或8
'f'	单精度浮点型	float	4
'd'	双精度浮点型	double	8

整型和长整型的表示方法取决于机器的架构（32位或64位）。当返回存储为'L'或'I'的值时，在Python 2中它们将以长整型的形式返回。

此模块定义以下类型：

```
array(typecode
 [, initializer
])
```

创建类型为 *typecode* 的数组。 *initializer* 是一个字符串或值列表，用于初始化数组中的值。以下属性和方法适用于array 对象 *a* 。

项　　目	描　　述
a.typecode	用于创建数组的类型编码字符

a.itemsize	存储在数组中项的大小（以字节为单位）
a.append(x)	将x 附加到数组末尾
a.buffer_info()	返回 *(address, length)* ，提供用于存储数组的缓冲区的内存位置和长度
a.byteswap()	在大尾与小尾之间切换数组中所有项的字节顺序。仅支持整型值
a.count(x)	返回*a* 中出现*x* 的次数
a.extend(b)	将*b* 附加到数组 *a* 的末尾。 *b* 可以是一个数组，也可以是一个元素类型与 *a* 中相同的可迭代对象
a.fromfile(f, n)	从文件对象 *f* 中读取 *n* 个项（二进制格式），并附加到数组末尾。 *f* 必须是一个文件对象。如果可读取的项少于 *n* ，则抛出EOFError
a.fromlist(list)	将 *list* 中的项附加到数组末尾。 *list* 可以是任何可迭代对象
a.fromstring(s)	附加字符串 *s* 中的项，其中 *s* 是一个由二进制值组成的字符串，与使用 *fromfile()* 进行读取相同
a.index(x)	返回 *x* 在 *a* 中首次出现的位置索引。如果未找到，则抛出 *ValueError*
a.insert(i, x)	在位置 *i* 前插入 *x*
a.pop([i])	从数组中删除项 *i* 并将其返回。如果 *i* 已被删除，则删除最后一个元素
a.remove(x)	从数组中删除第一个 *x* 。如果未找到，则抛出 *ValueError*
a.reverse()	反转数组的顺序
a.tofile(f)	将所有项写入文件 *f* 。数据保存为本机二进制格式
a.tolist()	将数组转换为普通的值列表
a.tostring()	转换为由二进制数据组成的字符串，与使用 *tofile()* 写入的数据相同
a.tounicode()	将数组转换为Unicode字符串。如果数组类型不为 *'u'* ，则抛出 *ValueError*

将项插入到数组中时，如果项的类型与用于创建数组的类型不匹配，则生成TypeError 异常。

如果需要有效利用存储数据列表的空间，并且列表中的所有项都将使用相同类型，那么array 模块将很有用。例如，在一个列表中存储1000万个整数大约需要160MB的内存，而包含1000万个整数的数组仅需要40MB。尽管节省了这么多空间，但对array 的所有基本操作并不比对应的列表快，实际上反而更慢。

在执行数组计算时，要注意创建列表的操作。例如，在数组上使用列表推导会将整个数组转换为列表，这会错失任何空间节省优势。更好的处理方式是使用生成器表达式创建新数组。例如：

```
a = array.array("i", [1,2,3,4,5])
b = array.array(a.typecode, (2*x for x in a))   # 从b创建新数组
```

因为使用数组是为了节省空间，所以执行“原地”操作可能更符合需要。一种有效的方式是使用enumerate()，比如：

```
a = array.array("i", [1,2,3,4,5])
for i, x in enumerate(a):
     a[i] = 2*x
```

对于大型数组，这种原地修改的运行速度比使用生成器表达式创建新数组快大约15%。

注意

- 此模块创建的数组不适用于数字操作，如矩阵或矢量运算。例如，加运算符不会添加数组中的相应元素，而是将一个数组附加到另一个。要创建可高效存储和计算的数组，请使用numpy 扩展（可从http://numpy.sourceforge.net/ 获得）。注意，numpy API与此扩展完全不同。
- += 运算符可用于附加另一个数组的内容。*= 运算符可用于重复附加一个数组。

另请参见： 16.4节。

15.3 bisect

bisect 模块支持使列表保持已排好的顺序。它使用二分算法来执行大部分工作。

```
bisect(list
, item
 [, low
 [, high
```

```
]])
```

返回要放在 *List* 中的 *item* 的插入点索引，以便让 *List* 维持已排好的顺序。*low* 和 *high* 是两个索引，指定要检查的列表的子集。如果 *item* 已经在列表中，那么插入点将始终位于列表中现有条目的右边。

```
bisect_left(list
, item
 [, low
 [, high
]])
```

返回要放在 *List* 中的 *item* 的插入点索引，以便让 *List* 维持已排好的顺序。*low* 和 *high* 是两个索引，指定要检查的列表的子集。如果 *item* 已经在列表中，那么插入点将始终位于列表中现有条目的左边。

```
bisect_right(list
, item
 [, low
 [, high
]])
```

与bisect()相同。

```
insort(list
, item
 [, low
 [, high
```

```
]])
```

按已排好的顺序将项插入列表中。如果 *item* 已在列表中，那么新项将插到所有现有项的右边。

```
insort_left(list
, item
 [, low
 [, high
]])
```

按已排好的顺序将项插入列表中。如果 *item* 已在列表中，那么新项将插到所有现有项的左边。

```
insort_right(list
, item
 [, low
 [, high
]])
```

与insort() 相同。

15.4 collections

collections 模块包含一些有用容器类型的高性能实现、各种容器的抽象基类，以及用于创建名称元组对象的实用工具函数。下面介绍每一部分。

15.4.1 deque 和defaultdict

collections 模块中定义了两个新容器：deque 和defaultdict 。

```
deque([iterable
 [, maxlen
]])
```

表示双端队列（deque，发音为“deck”）对象的类型。iterable 是可迭代对象，用于填充deque 。双端队列 允许在队列的任意一端插入或删除项。它的实现已经优化过，所以这些操作的性能接近于O(1) 。这与列表稍有不同，对列表前端的操作可能需要移动所有后续对象。如果提供了可选maxlen 参数，生成的deque 对象就会成为具有该大小的循环缓冲区。也就是说，如果添加新项，但没有更多空间，那么会在相反的一端删除对象来腾出空间。

deque 的实例 *d* 具有以下方法：

```
d
.append(x
)
```

将 *x* 添加到 *d* 的右端。

```
d
.appendleft(x
)
```

将 *x* 添加到 *d* 的左端。

```
d
.clear()
```

从 *d* 中删除所有项。

```
d
.extend(iterable
)
```

将 *iterable* 中的所有项添加到 *d* 的右端，以扩展它。

```
d
.extendleft(iterable
)
```

将iterable 中的所有项添加到 *d* 的左端，以扩展它。由于在左侧附加的顺序不同，所以 *iterable* 中的项将按相反顺序出现在 *d* 中。

```
d
.pop()
```

返回并删除 *d* 右端的项。如果 *d* 是空的，则抛出IndexError 。

```
d
.popleft()
```

返回并删除 *d* 左端的项。如果d 是空的，则抛出IndexError 。

```
d
.remove(item
)
```

删除首次出现的 *item* 。如果未找到匹配值，则抛出ValueError 。

```
d
.rotate(n
)
```

将所有项向右旋转 *n* 步。如果 *n* 为负值，则向左旋转项。

许多Python程序员常常忽略双端队列，其实这种类型具有许多优势。首先，它的实现非常高效，甚至可以与提供出色处理器缓存的内部数据结构媲美。将项附加到后端的速度比内置 *list* 类型稍慢，而将项插入到前端则快得多。将新项添加到双端队列中也是线程安全的，所以这种类型适合于实现队列。双端队列也可以使用 *pickle* 模块来序列化。

```
defaultdict([default_factory
], ...)
```

此类型与字典基本一样，除了在对缺少键的处理上。当查找不存在的键时，将调用作为default_factory 提供的函数来提供一个默认值，然后将该值保存为关联键的值。defaultdict 的其余参数与内置dict() 函数完全相同。可以对defaultdictionary 的实例 *d* 执行与内置字典相同的操作。属性 *d* .default_factory 包含作为第一个参数传递的函数，并可以根据需要进行修改。

如果想要使用字典作为跟踪数据的容器，那么defaultdict 对象将很有用。例如，要跟踪字符串 *s* 中每个单词的位置。下面展示如何使用defaultdict 来轻松完成此任务：

```
>>> from collections import defaultdict

>>> s = "yeah but no but yeah but no but yeah"

>>> words = s.split()

>>> wordlocations = defaultdict(list)

>>> for n, w in enumerate(words):

...     wordlocations[w].append(n)

...
>>> wordlocations

defaultdict(<type 'list'>, {'yeah': [0, 4, 8], 'but': [1, 3, 5, 7], 'no': [2, 6]})
>>>
```

在这个例子中，查找函数wordlocations[w] 在遇到第一个单词时将“失败”。但是，它不会抛出KeyError 错误，而会调用作为default_factory 提供的函数list 来创建一个新值。内置字典拥有一个setdefault() 方法，可用于实现类似的结果，但它常常使代码不易理解且减慢代码运行速度。例如，前面显示的附加新项的语句可以替换为wordlocations.setdefault(w,[]).append(n) 。该语句条理不是很清晰，而且通过简单的计时测试表明，它的运行速度几乎比使用defaultdict 对象慢两倍。

15.4.2 命名元组

元组 常常用于表示简单的数据结构。例如，网络地址可以表示为元组addr = (hostname, port) 。元组的常见缺陷是，需要使用数字索引访问各个项，如addr[0] 或addr[1] 。这会生成难以理解和维护的代码，除非你可以记住所有索引值的含义（而且元组越大，此问题就越严重）。

collections 模块包含函数namedtuple() ，此函数用于创建tuple 的子类，在tuple 中可以使用属性名称来访问元组元素。

```
namedtuple(typename

, fieldnames
```

```
 [, verbose

])
```

使用名称typename 创建tuple 的子类。fieldnames 是以字符串形式指定的属性名称列表。此列表中的名称必须是有效的Python标识符，不能以下划线开头，而且需按项在元组中出现的顺序来指定，如['hostname','port'] 。也可以将fieldnames 指定为'hostname port' 或'hostname, port' 等字符串。此函数返回的值为一个类，该类的名称设置为typename 中提供的值。使用此类来创建命名元组的实例。如果verbose 标记设置为True ，将向标准输出端输出生成的类定义。

下面是一个使用此函数的例子：

```
>>> from collections import namedtuple


>>> NetworkAddress = namedtuple('NetworkAddress',['hostname','port'])


>>> a = NetworkAddress('www.python.org',80)


>>> a.hostname


'www.python.org'
>>> a.port


80
>>> host, port = a


>>> len(a)


2
>>> type(a)


<class '__main__.NetworkAddress'>
>>> isinstance(a, tuple)


True
>>>
```

在这个例子中，命名元组NetworkAddress 在各方面都与普通的tuple 相同，但它支持使用属性查找（如a.hostname 或a.port ）来访问元组组件。其底层实现是高效的

——创建的类不会使用实例字典，也不会在内置tuple 中增加任何内存开销。所有普通元组操作仍然有效。

如果定义仅用作数据结构的对象，那么使用命名元组会有帮助。例如，无需像下面这样定义一个类：

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```

可以定义命名元组来代替：

```
import collections
Stock = collections.namedtuple('Stock','name shares price')
```

两个版本的工作原理几乎相同。例如，无论采用哪种方法，都可以通过编写s.name 、s.shares 等来访问字段。但是，命名元组的优势在于：它能够更高效地使用内存，并且支持解包等元组操作。例如，如果有一个命名元组列表，则可以使用for name, shares, price in stockList 等语句将值解包到一个for 循环中。命名元组的缺点在于：属性访问没有类那么高效。例如，如果s 是命名元组的实例，而不是普通类的实例，那么访问s.shares 的速度将慢两倍。

以往，命名元组常常用在Python标准库的其他部分。现在这种用法有点过时了：在许多库模块中，元组最初被用作各种函数的返回值，这些函数可能返回关于文件、栈帧或其他底层细节的信息。使用这些元组的代码并不总是那么优美。因此，使用命名元组的目的在于：在不破坏后端兼容性的情况下消除这些用法。元组的另一个小问题是，一旦开始使用，预期的字段数量将永远锁定。例如，如果意外添加了一个新字段，那么旧代码将被破坏。命名元组的一些变体已在库中用于将新字段添加到某些函数返回的数据中。例如，对象可能支持遗留元组接口，但它提供了更多值，这些值不仅仅用作命名属性。

15.4.3 抽象基类

collections 模块定义一系列抽象基类。设计这些类是为了描述各种容器（如列表、集和字典）上的编程接口。这些类主要有两个用途。首先，它们可用作用户定义对象的基类，这些对象需要模仿内置容器类型的功能。其次，它们可用于类型检查。例如，如果需要检查 *s* 是否类似于一个序列，可以使用isinstance(*s* , collections.Sequence) 。

```
Container
```

所有容器的基类。定义抽象方法__contains__()，实现in 运算符。

Hashable

可用作散列表键的对象的基类。定义抽象方法__hash__()。

Iterable

支持迭代协议的对象的基类。定义抽象方法__iter__()。

Iterator

迭代器对象的基类。定义抽象方法next() 并继承Iterable，还提供一个不执行任何操作的默认__iter__() 实现。

Sized

可以确定其大小的容器的基类。定义抽象方法__len__()。

Callable

支持函数调用的对象的基类。定义抽象方法__call__()。

Sequence

类似于序列的对象的基类。继承自Container、Iterable和Sized，定义抽象方法__getitem__()和__len__()。还提供了__contains__()、__iter__()、__reversed__()、index()和count()的默认实现，这些方法是使用__getitem__()和__len__()方法实现的。

MutableSequence

易变序列的基类。继承自Sequence并添加了抽象方法__setitem__()和__delitem__()。还提供了append()、reverse()、extend()、pop()、remove()和__iadd__()的默认实现。

Set

类似于集的对象的基类。继承自Container、Iterable和Sized，定义抽象方法__len__()、__iter__()和__contains__()。还提供了集运算符__le__()、__lt__()、__eq__()、__ne__()、__gt__()、__ge__()、__and__()、__or__()、__xor__()、__sub__()和isdisjoint()的默认实现。

MutableSet

易变集的基类。继承自Set，添加了抽象方法add()和discard()。还提供了clear()、pop()、remove()、__ior__()、__iand__()、__ixor__()和__isub__()的默认实现。

Mapping

支持映射（字典）查找的对象的基类。继承自Sized、Iterable和Container，定义了抽象方法__getitem__()、__len__()和__iter__()。还提供了__contains__()、keys()、items()、values()、get()、__eq __()和__ne__()

的默认实现。

MutableMapping

易变映射对象的基类。继承自Mapping，添加了抽象方法__setitem__()和__delitem__()。还添加了pop()、popitem()、clear()、update()和setdefault()的实现。

MappingView

映射视图的基类。映射视图 是一种对象，用于以集的形式访问映射对象的内部。例如，键视图是一个类似于集的对象，显示映射中的键。参阅附录A了解更多细节。

KeysView

映射的键视图的基类。继承自MappingView和Set。

ItemsView

映射的项视图的基类。继承自MappingView和Set。

ValuesView

映射的(key, item)视图的基类。继承自MappingView和Set。

Python的内置类型已注册了所有这些基类。而且，通过使用这些基类，可以编写能更准确地进行类型检查的程序。下面给出一些例子。

```
# 获取序列中最后一个项
if isinstance(x, collections.Sequence):
   last = x[-1]

# 仅在一个对象上进行迭代，只要对象大小是已知的
if isinstance(x, collections.Iterable) and isinstance(x, collections.Sized):
   for item in x:
       statements

# 向集添加一个新项
if isinstance(x, collections.MutableSet):
   x.add(item)
```

另请参见： 第7章。

15.5 contextlib

contextlib 模块提供一个装饰器和一些实用工具函数，用于创建与with 语句结合使用的上下文管理器。

```
contextmanager(func

)
```

一个装饰器，根据生成器函数 *func* 创建一个上下文管理器。使用此装饰器的方式如下：

```
@contextmanager
def foo(args

):
    statements


    try:
        yield value
    except Exception as e:
        error handling (if any)


    statements
```

当语句with foo(args) as value 出现时，使用所提供的参数执行生成器函数，直到到达第一条yield 语句。yield 返回的值放在变量value 中。此时执行with 语句体。

完成之后，生成器函数将继续执行。如果with 语句体中出现任何异常，生成器函数内会抛出该异常并可被适当处理。如果错误即将被传播，生成器应该再次抛出该异常。可以在5.5节找到使用此装饰器的例子。

```
nested(mgr1
, mgr2
, ..., mgrN
)
```

此函数在一个操作中调用多个上下文管理器（mgr1 、mgr2 等）。返回一个元组，其中包含with 语句的不同返回值。语句with nested(m1,m2) as (x,y): *statements* 与语句with m1 as x: with m2 as y: *statements* 含义相同。注意，如果内部上下文管理器捕获并禁止了异常，将不会向外部管理器传送任何异常信息。

```
closing(object
)
```

创建上下文管理器，在执行过程离开with 语句体时自动执行object.close() 。with 语句返回的值与object 相同。

15.6 functools

functools 模块包含能够创建高阶函数、函数式编程和装饰器的函数和装饰器。

```
partial(function
 [, *args
 [, **kwargs
]])
```

创建一个类似函数的对象partial ，当调用该对象时，会使用位置参数 *args* 、关

键字参数 *kwargs* 和任何附加位置或关键字参数来调用 *function* 。附加的位置参数添加到 *args* 末尾，附加的关键字参数合并到 *kwargs* 中，覆盖以前定义的任何值（如果有）。当执行大量函数调用且许多参数都是固定时一般会使用partial()。例如：

```
from functools import partial
mybutton = partial(Button, root, fg="black",bg="white",font="times",size="12")
b1 = mybutton(text="Ok")        # 调用Button()，同时使用text="Ok"和上面提供给partial()
b2 = mybutton(text="Cancel")    # 的所有附加参数
b3 = mybutton(text="Restart")
```

partial 创建的对象的实例 *p* 具有以下属性：

项　　目	描　　述
p .func	调用 *p* 时调用的函数
p .args	一个元组，包含在调用 *p.func* 时提供给它的最左边的位置参数。其他位置参数串联在此值末尾
p .keywords	一个字典，包含在调用 *p.func* 时提供给它的关键字参数。其他关键字参数合并到此字典中

在使用partial 对象替代常规函数时一定要谨慎。得到的结果与使用常规函数的结果并不完全一样。例如，如果在类定义中使用partial()，那么该函数的行为类似于静态方法，而不是实例方法。

```
reduce(function
, items
 [, initial
])
```

向可迭代的items 中的项渐增地应用函数 *function* ，并返回一个值。 *function* 必须接受两个参数，并首先应用到 *items* 中的前两项。然后将此结果以类似方式与items 中的后续元素相组合，一次与一个元素组合，直到使用了 *items* 中的所有元素。 *initial* 是在首次计算时以及在 *items* 是空时使用的一个可选初始值。此函数与Python 2中内置的 *reduce()* 函数相同。为了保证以后的兼容性，请使用此版本。

```
update_wrapper(wrapper
```

```
, wrapped

 [, assigned

 [, updated

]])
```

这是一个实用工具函数，在编写装饰器时很有用。将函数 *wrapped* 的属性赋值到包装器函数 *wrapper* ，使包装的函数类似于原始函数。 *assigned* 是要复制的属性名元组，默认设置为('__name__','__module__','__doc__')。 *updated* 是一个元组，包含一些函数属性名称，这些函数属性是字典并且你需要将它们的值合并到包装器中。默认情况下，它为元组('__dict__',)。

```
wraps(function

 [, assigned

 [, updated

 ]])
```

一个装饰器，在它所应用的函数上执行与update_wrapper()相同的任务。*assigned* 和 *updated* 的含义相同。在编写其他装饰器时通常会用到这个装饰器。例如：

```
from functools import wraps
def debug(func):
    @wraps(func)
    def wrapped(*args,**kwargs):
        print("Calling %s" % func.__name__)
        r = func(*args,**kwargs)
        print("Done calling %s" % func.__name__)
    return wrapped

@debug
def add(x,y):
    return x+y
```

另请参见： 第6章。

15.7 heapq

heapq 模块使用堆实现一个优先级队列。堆就是一个有序的项列表，在其中会强制执行堆叠条件。具体来讲，对于从n=0.heap[0] 开始的所有n ， heap[n]<=heap[2*n+1] 和heap[n]<=heap[2*n+2] 始终包含最小项。

```
heapify(x
)
```

将列表x原地转换为堆。

```
heappop(heap
)
```

返回并删除 *heap* 中的最小项，保留堆条件。如果 *heap* 为空，则抛出 IndexError 。

```
heappush(heap
, item
)
```

将 *item* 添加到堆中，保留堆条件。

```
heappushpop(heap
, item
)
```

在一个操作中将 *item* 添加到堆并从 *heap* 中删除最小项。这比分别调用heappush() 和heappop() 更高效。

```
heapreplace(heap
, item
)
```

返回并删除堆中的最小 *item* 。同时添加一个新 *item* 。在此过程中会保留堆条件。此函数比依次调用heappop() 和heappush() 更高效。此外，新项的添加在获取返回值之后。因此，返回值可能比 *item* 大。如果 *heap* 为空，则抛出IndexError 。

```
merge(s1
, s2
, ...)
```

创建一个迭代器，将有序的迭代变量 *s1* 、 *s2* 等合并到一个有序序列中。此函数不需要输入，但是会返回一个迭代器，该迭代器增量式地处理数据。

```
nlargest(n
, iterable
 [, key
])
```

创建一个列表，包含 *iterable* 中最大的 *n* 个项。最大项排在返回列表的前端。*key* 是可选函数，接受一个输入参数并计算 *iterable* 中的每个项的比较键。

```
nsmallest(n
, iterable
 [, key
])
```

创建一个列表，包含 *iterable* 中最小的n 个项。最小项排在返回列表的前端。*key* 是一个可选键函数。

> **注意**
>
> 堆队列的理论和实现在大部分介绍算法的书中都可以找到。

15.8 itertools

itertools 模块包含创建高效迭代器的函数，可以用各种方式对数据执行循环操作。此模块中的所有函数返回的迭代器都可以与for 语句以及其他包含迭代器（如生成器和生成器表达式）的函数结合使用。

```
chain(iter1

, iter2

, ..., iterN

)
```

给定一组迭代器(*iter1* ,... , *iterN*)，此函数创建一个新迭代器来将所有迭代器链接起来。返回的迭代器从 *iter1* 开始生成项，直到 *iter1* 被用完，然后从 *iter2* 生成项。这一过程会持续到 *iterN* 中的所有项都被用完。

```
chain.from_iterable(iterables

)
```

一个备用链构造函数，其中的 *iterables* 是一个迭代变量，生成迭代对象序列。此操作的结果与以下生成器代码片段生成的结果相同：

```
for it in iterables:
    for x in it:
        yield x

combinations(iterable

, r
```

```
)
```

创建一个迭代器，返回 *iterable* 中所有长度为 *r* 的子序列。返回的子序列中的项按输入 *iterable* 中的顺序排序。例如，如果 *iterable* 是列表[1,2,3,4]，那么combinations([1,2,3,4], 2)生成的序列为[1,2]、[1,3]、[1,4]、[2,3]、[3,4]。

```
count([n

])
```

创建一个迭代器，生成从 *n* 开始的连续整数。如果忽略 *n* ，则从0开始计算（注意，此迭代器不支持长整数。如果超出了 *sys.maxint* ，计数器将溢出并继续从 *-sys.maxint - 1* 开始计算。）

```
cycle(iterable

)
```

创建一个迭代器，对 *iterable* 中的元素反复执行循环操作。内部会生成 *iterable* 中的元素的一个副本。此副本用于返回循环中的重复项。

```
dropwhile(predicate

, iterable

)
```

创建一个迭代器，只要函数 *predicate* (*item*)为True，就丢弃 *iterable* 中的项。如果 *predicate* 返回False，就会生成 *iterable* 中的项和所有后续项。

```
groupby(iterable
```

```
 [, key

])
```

创建一个迭代器，对 *iterable* 生成的连续项进行分组。在分组过程中会查找重复项。例如，如果 *iterable* 在多次连续迭代中生成了同一项，则会定义一个组。如果将此函数应用于一个分类列表，那么分组将定义该列表中的所有唯一项。 *key* （如果已提供）是一个函数，应用于每一项。如果此函数存在返回值，该值将用于与后续项而不是该项本身进行比较。此函数返回的迭代器生成元素(*key* , *group*)，其中 *key* 是分组的键值， *group* 是迭代器，生成组成该组的所有项。

```
ifilter(predicate

, iterable

)
```

创建一个迭代器，仅生成 *iterable* 中 *predicate* (*item*) 为True 的项。如果 *predicate* 为None，将返回 *iterable* 中所有计算为True 的项。

```
ifilterfalse(predicate

, iterable

)
```

创建一个迭代器，仅生成 *iterable* 中 *predicate* (*item*) 为False 的项。如果 *predicate* 为None，则返回 *iterable* 中所有计算为False 的项。

```
imap(function

, iter1

, iter2

, ..., iterN

)
```

创建一个迭代器，生成项 *function* (*i1 ,i2, ... iN*)，其中 *i1* 、*i2* ……*iN* 分别来自迭代器 *iter1* 、 *iter2*……*iterN* 。如果 *function* 为None，则返回(*i1 , i2 , ..., iN*)形式的元组。只要提供的一个迭代器不再生成值，迭代就会停止。

```
islice(iterable
, [start
,] stop
 [, step
])
```

创建一个迭代器，生成项的方式类似于切片返回值： *iterable* [*start* :*stop* :*step*]。将跳过前 *start* 个项，迭代在 *stop* 所指定的位置停止。 *step* 指定用于跳过项的步幅。与切片不同，负值不会用于任何 *start* 、 *stop* 或 *step* 。如果省略了 *start* ，迭代将从0开始。如果省略了 *step* ，则使用步幅1。

```
izip(iter1
, iter2
, ... iterN
)
```

创建一个迭代器，生成元组(*i1 , i2 , ..., iN*)，其中 *i1*、*i2*……*iN* 分别来自迭代器 *iter1* 、 *iter2*……*iterN* 。只要提供的某个迭代器不再生成值，迭代就会停止。此函数生成的值与内置的zip() 函数相同。

```
izip_longest(iter1
, iter2
, ..., iterN
 [,fillvalue=None])
```

与izip() 相同，但迭代过程会持续到所有输入迭代变量 *iter1* 、 *iter2* 等都耗尽为止。如果没有使用fillvalue 关键字参数指定不同的值，则使用None 来填充已经使用的迭代变量的值。

```
permutations(iterable
 [, r
])
```

创建一个迭代器，返回 *iterable* 中所有长度为 *r* 的项序列。如果省略了 *r* ，那么序列的长度与 *iterable* 中的项数量相同。

```
product(iter1
, iter2
, ... iterN
, [repeat=1])
```

创建一个迭代器，生成表示 *item1* 、 *item2* 等中的项的笛卡儿积的元组。*repeat* 是一个关键字参数，指定重复生成序列的次数。

```
repeat(object
 [, times
])
```

创建一个迭代器，重复生成 *object* 。 *times* （如果已提供）指定重复计数。如果未提供 *times* ，将无止尽地返回该对象。

```
starmap(func
```

```
 [, iterable

])
```

创建一个迭代器，生成值 *func* (**item*)，其中 *item* 来自 *iterable* 。只有当 *iterable* 生成的项适用于这种调用函数的方式时此函数才有效。

```
takewhile(predicate

 [, iterable

])
```

创建一个迭代器，生成 *iterable* 中 *predicate* (*item*)为True 的项。只要 *predicate* 计算为False ，迭代就会立即停止。

```
tee(iterable

 [, n

])
```

从 *iterable* 创建 *n* 个独立的迭代器。创建的迭代器以 *n* 元组的形式返回。 *n* 的默认值为2。此函数适用于任何可迭代的对象。但是，为了克隆原始迭代器，生成的项会被缓存，并在所有新创建的迭代器中使用。一定要注意，不要在调用tee() 之后使用原始迭代器iterable ， 否则缓存机制可能无法正常工作。

示例

以下示例演示了itertools 模块中一些函数的工作原理：

```
from itertools import *
# 在数字0, 1, …, 10, 9, 8, …, 1上执行无限循环
for i in cycle(chain(range(10),range(10,0,-1))):
    print i

# 创建a中的唯一项列表
a = [1,4,5,4,9,1,2,3,4,5,1]
a.sort()
```

```
b = [k for k,g in groupby(a)]   # b = [1,2,3,4,5,9]

# 对x和y中所有可能的值对组合进行迭代
x = [1,2,3,4,5]
y = [10,11,12]
for r in product(x,y):
    print(r)
# 生成输出(1,10),(1,11),(1,12), ... (5,10),(5,11),(5,12)
```

15.9 operator

operator 模块提供访问内置运算符和第3章中介绍的特殊解释器方法的函数。例如，add(3, 4) 相当于3 + 4 。对于还具有替代版本的操作，可以使用iadd(x,y) （等效于x += y ）这样的函数。下表列出了operator 中定义的函数以及它们与各种运算符的对应关系。

函　　数	描　　述
add(*a, b*)	返回 *a + b* （适用于数字）
sub(*a, b*)	返回 *a – b*
mul(*a, b*)	返回 *a * b* （适用于数字）
div(*a, b*)	返回 *a / b* （旧除法）
floordiv(*a, b*)	返回 *a // b*
truediv(*a, b*)	返回 *a / b* （新除法）
mod(*a, b*)	返回 *a % b*
neg(*a*)	返回 *-a*
pos(a)	返回 *+a*
abs(a)	返回 *a* 的绝对值
inv(*a*), *invert(a)*	返回 *a* 的倒数
lshift(*a, b*)	返回 *a << b*

rshift(a, b)	返回 a >> b
and_(a, b)	返回 a & b （按位AND）
or_(a, b)	返回 a | b （按位OR）
xor(a, b)	返回 a ^ b （按位XOR）
not_(a)	返回非 a
lt(a, b)	返回 a < b
le(a, b)	返回 a <= b
eq(a, b)	返回 a ==b
ne(a, b)	返回 a !=b
gt(a, b)	返回 a > b
ge(a, b)	返回 a >= b
truth(a)	如果 a 为真，返回Ture ； 否则返回False
concat(a, b)	返回 a + b （适用于数列）
repeat(a, b)	返回 a * b （适用于数列 a 和整数 b ）
contains(a, b)	返回 a 包含 b 的结果
countOf(a, b)	返回 a 中 b 出现的次数
indexOf(a, b)	返回 a 中 b 第一次出现的位置索引
getitem(a, b)	返回 a [b]
setitem(a, b, c)	a [b] = c

delitem(*a, b*)	del *a [b]*
getslice(*a, b, c*)	返回 *a[b:c]*
setslice(*a, b, c, v*)	设置 *a[b:c] = v*
delslice(*a, b, c*)	del *a[b:c]*
is_(*a, b*)	*a* 是 *b*
is_not(*a,b*)	*a* 不是 *b*

初看起来，你可能不明白为什么有人想要使用这些函数，因为它们执行的操作可通过常规语法轻松完成。这些函数适合处理使用回调函数的代码的情况，以及如果不使用这些函数，就需要使用lambda 定义匿名函数的情况。例如，考虑下面的计时基准测试代码，它使用了functools.reduce() 函数：

```
>>> from timeit import timeit


>>> timeit("reduce(operator.add,a)","import operator; a = range(100)")


12.055853843688965
>>> timeit("reduce(lambda x,y: x+y,a)","import operator; a = range(100)")


25.012306928634644
>>>
```

注意，在这个示例中，使用operator.add 作为回调的运行速度是使用lambda x,y: x+y 的版本速度的两倍。

operator 模块还定义了以下函数，它们创建针对属性访问、项查找和方法调用的包装器。

```
attrgetter(name

 [, name2

 [, ... [, nameN

]]])
```

创建可调用的对象 *f* ，其中对 *f* (obj) 的调用返回 *obj .name* 。如果提供了多个参数，则返回一个结果组。例如，调用attrgetter('name','shares') 会返回(*obj .name* , *obj .shares*)。 *name* 也可以包括附加的点查找。例如，如果 *name* 为"address.hostname"，那么 *f* (*obj*) 返回 *obj* .address. hostname 。

```
itemgetter(item
 [, item2
 [, ... [, itemN
]]])
```

创建可调用的对象 *f* ，其中 *f* (*obj*) 返回 *obj* [*item*]。如果提供了多个项作为参数，那么调用 *f* (*obj*) 将返回包含(*obj* [*item*], *obj* [*item2*], ..., *obj* [*itemN*]) 的元组。

```
methodcaller(name
 [, *args
 [, **kwargs
]])
```

创建可调用的对象 *f* ，其中 *f* (*obj*) 返回 *obj.name* (**args* ,***kwargs*)。

这些函数可以优化涉及回调函数的运算性能，特别是涉及分类等常见数据处理运算的操作。例如，如果需要对第二列上的元素 *rows* 列表进行分类，那么既可以使用sorted(*rows* , key=lambda r: r[2])，也可以使用sorted(*rows* ,key=itemgetter(2))。第二个版本的运行速度快得多，因为它避免了与lambda 相关的开销。

第16章　字符串和文本处理

本章介绍与基本字符串和文本处理相关的最常用Python模块。本章的重点是最常见的字符串操作，比如文本处理、正则表达式模式匹配以及文本格式化。

16.1　codecs

codecs 模块用于处理Unicode文本I/O中使用的各种字符编码。该模块既可用于定义新字符编码，也可以用于使用各种现有编码（如UTF-8、UTF-16等）来处理字符数据。对于程序员而言，使用一种现有编码的情况更常见，所以这里着重介绍这种情况。如果希望创建新编码，可以参考在线文档了解更多细节。

16.1.1　低级codecs 接口

每种字符编码都分配有一个通用名称，如'utf-8' 或'big5' 。下面这个函数用于查找编码名称。

```
lookup(encoding

)
```

在编码解码器注册表中查找一个编码解码器。 *encoding* 是一个字符串，如'utf-8' 。如果找不到关于所请求编码的任何信息，将会抛出LookupError 。否则，将返回CodecInfo 的实例 *c* 。

CodecInfo 实例 *c* 具有以下方法：

```
c.encode(s

 [, errors

])
```

一个无状态编码函数，编码Unicode字符串 *s* 并返回元组(*bytes* , *length_consumed*)。 *bytes* 是一个8位字符串或字节数组，包含已编码数据。 *length_consumed* 为 *s* 中已编码的字符数。 *errors* 是错误处理策略，默认设置为'strict' 。

```
c.decode(bytes
 [, errors
])
```

一个无状态编码函数，解码字节字符串 *bytes* 并返回元组(*s* , *length_consumed*)。 *s* 是一个Unicode字符串， *length_consumed* 是在解码时使用的 *bytes* 中的字节数。 *errors* 是错误处理策略，默认设置为'strict' 。

```
c
.streamreader(bytestream
 [, errors
])
```

返回一个StreamReader 实例，该实例用于读取已编码的数据。 *bytestream* 是一个类文件对象，已在二进制模式下打开。 *errors* 是错误处理策略，默认设置为'strict' 。StreamReader 的实例 *r* 支持以下低级I/O操作。

方　法	描　述
r .read([size [, *chars* [, *firstline*]]])	返回已解码文本中最多 *chars* 个字符。 *size* 是要从低级字节流读取的最大字节数，用于控制内部缓冲。 *firstline* 是一个标志，如果设置了该标志，即使在文件后面出现了解码错误，也会返回第1行
r .readline([*size* [, *keepends*]])	返回一行已解码文本。 *keepends* 是一个标志，控制是否保留行尾字符（默认设置为 *true* ）
r .readlines([*size* [, *keepends*]])	将所有行读取到一个列表中
r .reset()	重置内部缓冲区和状态

```
c.streamwriter(bytestream
[, errors
])
```

返回一个StreamWriter 实例，该实例用于写入已编码数据。 *bytestream* 是一个类文件对象，已在二进制模式下打开。 *errors* 是错误处理策略，默认设置为'strict' 。StreamWriter 的实例 *w* 支持以下低级I/O操作。

方　　法	描　　述
w .write(s)	写入字符串 *s* 的已编码表示形式
w .writelines(*lines*)	将 *lines* 中的一组字符串写入到文件
w .reset()	重置内部缓冲区和状态

```
c
.incrementalencoder([errors
])
```

返回一个IncrementalEncoder 实例，该实例可用于通过多个步骤编码字符串。*errors* 默认设置为'strict' 。IncrementalEncoder 的实例 *e* 具有以下方法。

方　　法	描　　述
e .encode(*s* [,*final*])	以字节字符串的形式返回字符串 *s* 的已编码表示形式。 *final* 是一个标志，应该在最后调用encode() 时设置为True
e .reset()	重置内部缓冲区和状态

```
c
.incrementaldecoder([errors
])
```

返回一个IncrementalDecoder 实例，该实例可用于通过多个步骤解码字节字符串。 *errors* 默认设置为'strict' 。IncrementalDecoder 的实例 *d* 具有以下方法。

方　　法	描　　述
d .decode(*bytes* [,*final*])	返回 *bytes* 中已编码字节的已解码字符串。 *final* 是一个标志，应该在最后调用decode() 时设置为True
d .reset()	重置内部缓冲区和状态

16.1.2 I/O相关函数

codecs 模块提供了一组高级函数，用于简化涉及已编码文本的I/O。大部分程序员都会使用其中的某个函数来替代前面介绍的低级编码解码器接口。

```
open(filename
, mode
[, encoding
[, errors
[, buffering
]]])
```

在给定 *mode* 下打开 *filename* ，根据 *encoding* 中指定的编码执行透明的数据编码/解码。 *errors* 可以是'strict' 、gnore' 、'replace' 、'backslashreplace' 或'xmlcharrefreplace' ，默认设置为'strict' 。 *buffering* 的含义与内置open() 函数中的含义相同。无论在 *mode* 中指定了何种模式，底层文件始终在二进制模式下打开。在Python 3中，可以使用内置的open() 函数代替codecs.open() 。

```
```

```
EncodedFile(file, inputenc[, outputenc [, errors]])
```

一个类，为现有文件对象 *file* 提供一个编码包装器。写入该文件的数据首先根据输入编码 *inputenc* 进行解释，然后使用输出编码 *outputenc* 写入文件。从该文件读取的数据会根据 *inputenc* 进行解码。如果省略了 *outputenc* ，它将默认设置为 *inputenc* 。 *errors* 的含义与open() 中的含义相同，默认设置为'strict' 。

```
iterencode(iterable
, encoding
 [, errors
])
```

生成器函数，增量式地将 *iterable* 中的所有字符串编码为指定的 *encoding* 。 *errors* 默认设置为'strict' 。

```
iterdecode(iterable
, encoding
 [, errors
])
```

生成器函数，根据指定的 *encoding* 增量式地解码 *iterable* 中的所有字节字符串。 *errors* 默认设置为'strict' 。

16.1.3 有用的常量

codecs 定义了以下字节顺序标记常量，可用于在不了解底层编码时帮助解释文件。这些字节顺序标记有时包含在文件开头，用于指示文件的字符编码，也可用于挑选要使用的适当编码解码器。

常　量	描　述

BOM	机器的本机字节顺序标记（BOM_BE 或BOM_LE ）
BOM_BE	大尾字节顺序标记（'\xfe\xff' ）
BOM_LE	小尾字节顺序标记（'\xfe\xfe' ）
BOM_UTF8	UTF-8标记（'\xef\xbb\xbf' ）
BOM_UTF16_BE	16位UTF-16大尾标记（'\xfe\xff' ）
BOM_UTF16_LE	16位UTF-16小尾标记（'\xfe\xfe' ）
BOM_UTF32_BE	32位UTF-32大尾标记（'\x00\x00\xfe\xff' ）
BOM_UTF32_LE	32位UTF-32小尾标记（'\xff\xfe\x00\x00' ）

16.1.4 标准编码

以下是一些最常用的字符编码。编码名称就是在指定编码时传递给函数（如open()或lookup() ）的参数。完整的编码列表可以参考codecs 模块的在线文档（http://docs.python.org/library/codecs ）。

编解码器名称	描　　述	编解码器名称	描　　述
ascii	7位ASCII字符	utf-16-le	UTF-16小尾
cp437	MS-DOS扩展ASCII字符集	utf-32	UTF-32
cp1252	Windows扩展ASCII字符集	utf-32-be	UTF-32大尾
latin-1,iso-8859-1	包含拉丁字符的扩展ASCII	utf-32-le	UTF-32小尾
utf-16	UTF-16	utf-8	UTF-32
utf-16-be	UTF-16大尾		

16.1.5 注意

- codecs 模块的更多用途已在第9章介绍。
- 参考在线文档，了解如何创建新的字符编码类型。
- 尤其需要注意encode() 和decode() 操作的输入。所有encode() 操作都应该处理Unicode字符串，而所有decode() 操作都应该处理字节字符串。Python 2在这一点上并不完全一致，而Python 3会严格区分不同字符串。例如，Python 2中的一些编码解码器将字节字符串映射到字节字符串（如“bz 2”编码解码器）。这些编码解码器在Python 3中不可用，而且如果重视兼容性问题，在Python 2中也不应该使用它们。

16.2 re

re 模块用于在字符串中执行正则表达式匹配和替换。它同时支持Unicode和字节字符串。正则表达式模式是以包含文本和特殊字符序列的字符串形式指定的。由于模式大量使用特殊字符和反斜杠，所以它们通常写为“原始”字符串，如r'(?P<int>\d+).(\d*)' 。提醒一下，本节中的所有正则表达式都使用原始字符串语法来表示。

16.2.1 模式语法

正则表达式模式能够识别以下特殊字符序列。

字　符	描　述
text	匹配文字字符串 *text*
.	匹配任何字符串，但换行符除外
^	匹配字符串的开始标志
$	匹配字符串的结束标志
*	匹配前面表达式的0个或多个副本，匹配尽可能多的副本
+	匹配前面表达式的1个或多个副本，匹配尽可能多的副本
?	匹配前面表达式的0个或多个副本
*?	匹配前面表达式的0个或多个副本，匹配尽可能少的副本
+?	匹配前面表达式的1个或多个副本，匹配尽可能少的副本
??	匹配前面表达式的0个或1个副本，匹配尽可能少的副本

{*m* }	准确匹配前面表达式的 *m* 个副本
{*m* , *n* }	匹配前面表达式的第 *m* 到 *n* 个副本，匹配尽可能多的副本。如果省略了 *m* ，它将默认设置为0。如果省略了 *n* ，它将默认设置为无穷大
{*m* , *n* }?	匹配前面表达式的第 *m* 到 *n* 个副本，匹配尽可能少的副本
[...]	匹配一组字符，如r'[abcdef]' 或r'[a-zA-z]' 。特殊字符（如* ）在字符集中是无效的
[^...]	匹配集合中未包含的字符，如r'[^0-9]'
A | *B*	匹配 *A* 或 *B* ，其中 *A* 和 *B* 都是正则表达式
(...)	匹配圆括号中的正则表达式（圆括号中的内容为一个分组）并保存匹配的子字符串。在匹配时，分组中的内容可以使用所获得的MatchObject 对象的group() 方法获取
(?aiLmsux)	将字符"a" 、 "i" 、 "L" 、 "m" 、 "s" 、 "u" 和"x" 解释为与提供给re.compile() 的re.A 、 re.I 、 re.L 、 re.M 、 re.S 、 re.U 、 re.X 相对应的标志设置。"a" 仅在Python 3中可用
(?:...)	匹配圆括号中的正则表达式，但丢弃匹配的子字符串
(?P<*name* >...)	匹配圆括号中的正则表达式并创建一个指定分组。分组名称必须是有效的Python标识符
(?P=*name*)	匹配一个早期指定的分组所匹配的文本
(?#...)	一个注释。圆括号中的内容将被忽略
(?=...)	只有在括号中的模式匹配时，才匹配前面的表达式。例如，'Hello(?=World)' 只有在'World' 匹配时才匹配'Hello '
(?!...)	只有在括号中的模式不匹配时，才匹配前面的表达式。例如，'Hello(?!=World)' 只有在'World' 不匹配时才匹配'Hello '
(?<=...)	如果括号后面的表达式前面的值与括号中的模式匹配，则匹配该表达式。例如，只有当'def' 前面是'abc' 时，r'(?<=abc)def' 才会与它匹配
(?<!...)	如果括号后面的表达式前面的值与括号中的模式不匹配，则匹配该表达式。例如，只有当'def' 前面不是'abc' 时，r'(?<=abc)def' 才会与它匹配
(?(*id* | *name*)*ypat* | *npat*)	检查id 或name 标识的正则表达式组是否存在。如果存在，则匹配正则表达式ypat 。否则，匹配可选的表达式npat 。例如，模式r'(Hello)?(?(1)World | Howdy)' 匹配字符串'Hello World' 或'Howdy'

在正则表达式中，标准字符转义序列（如'\n' 和'\t' ）被认为是标准字符，例如，r'\n+' 可以与一个或多个换行字符匹配。此外，要在正则表达式中指定通常拥有特殊含义的文字符号，可以在它们前面加上反斜杠。例如，r'*' 与字符* 匹配。此外，还有许多反斜杠序列与特殊的字符集对应。

字符	描　　述
\number	匹配与前面的组编号匹配的文本。组编号范围为1到99，从左侧开始
\A	仅匹配字符串的开始标志
\b	匹配单词开始或结尾处的空字符串。单词 （word）是一个字母数字混合的字符序列，以空格或任何其他非字母数字字符结束
\B	匹配不在单词开始或结尾处的空字符串
\d	匹配任何十进制数。等同于r'[0-9]'
\D	匹配任何非数字字符。等同于r'[^0-9]'
\s	匹配任何空格字符。等同于r'[\t\n\r\f\v]'
\S	匹配任何非空格字符。等同于[^ \t\n\r\f\v]'
\w	匹配任何字母数字字符
\W	匹配\w 定义的集合中不包含的字符
\z	仅匹配字符串的结束标志
\\	匹配反斜杠本身

在与Unicode字符串进行匹配时，特殊字符\d 、\D 、\s 、\S 、\w 和\W 的含义不同。在这种情况下，它们将匹配与所描述属性相匹配的所有Unicode字符。例如，\d 匹配分类为数字的所有Unicode字符，如欧洲数字、阿拉伯数字和印度数字，它们分别占据了不同的Unicode字符范围。

16.2.2 函数

以下函数用于执行模式匹配和替换。

```
compile(str
[, flags
])
```

将正则表达式模式字符串编译为正则表达式对象。此对象可以作为模式参数传递给随后的所有函数。该对象还提供了许多方法，稍后将介绍。 *flags* 是以下标志的按位OR结果。

标　志	描　述
A 或ASCII	执行仅8位ASCII字符匹配（仅适用于Python 3）
I 或IGNORECASE	执行不区分大小写的匹配
L 或LOCALE	为\w 、 \W 、 \b 和\B 使用地区设置
M 或MULTILINE	将^ 和$ 应用于包括整个字符串的开始和结尾的每一行（在正常情况下，^ 和$ 仅适用于整个字符串的开始和结尾）
S 或DOTALL	使点（. ）字符匹配所有字符，包括换行符
U 或UNICODE	使用\w 、 \W 、 \b 和\B 在Unicode字符属性数据库中的信息（仅限于Python 2。Python 3默认使用Unicode）
X 或VERBOSE	忽略模式字符串中未转义的空格和注释

```
escape(string
)
```

返回一个字符串，其中的所有非字母数字字符都带有反斜杠。

```
findall(pattern
, string
 [,flags
])
```

返回 *string* 中与 *pattern* 匹配的所有未重叠的值，包括空匹配值。如果模式包含分组，将返回与分组匹配的文本列表。如果使用了不只一个分组，那么列表中的每项都是一个元组，包含每个分组的文本。 *flags* 的含义与compile() 相同。

```
finditer(pattern
, string
, [, flags
])
```

与findall() 含义相同，但返回一个迭代器对象。迭代器返回类型为MatchObject 的项。

```
match(pattern
, string
 [, flags
])
```

检查 *string* 的开头是否有字符与 *pattern* 匹配。如果成功，则返回MatchObject ；否则返回None 。 *flags* 的含义与compile() 相同。

```
search(pattern
```

```
, string
 [, flags
])
```

在 *string* 中搜索pattern 的第一个匹配值。 *flags* 的含义与compile() 相同。如果成功，则返回MatchObject ； 如果未找到匹配值，则返回None 。

```
split(pattern
, string
 [, maxsplit
 = 0])
```

根据pattern 出现的位置拆分string 。返回字符串列表，其中包括与模式中任何分组匹配的文本。 *maxsplit* 是执行拆分的最高次数。默认情况下，将执行所有可能的拆分。

```
sub(pattern
, repl
, string
 [, count
 = 0])
```

使用替换值 *rep1* 替换 *string* 中最左侧的、未重叠的 *pattern* 的出现位置。*rep1* 可以是字符串或函数。如果它是一个函数，则使用MatchObject 调用它，并返回替换字符串。如果 *rep1* 是一个字符串，则使用反向引用（如'\6' ）来引用模式中的分组。序列'\g<*name* >' 用于引用给定名称的分组。 *count* 是执行替换的最高次数。默认情况下，将替换所有出现的位置。尽管这些函数不接受 *flags* 参数，如compile() ，但可以使用本节前面介绍的(?iLmsux) 实现相同效果。

```
subn(pattern
```

```
, repl
, string
 [, count
 = 0])
```

与sub() 相同，但返回一个元组，其中包含新字符串和替换次数。

16.2.3 正则表达式对象

由compile() 函数创建的经过编译的正则表达式对象 *r* 具有以下方法和属性。

```
r
.flags
```

在编译正则表达式时使用 *flags* 参数，如果没有指定标志则使用0。

```
r
.groupindex
```

一个字典，将r'(?P<id>)' 定义的符号分组名称映射到分组编号。

```
r
.pattern
```

一个模式字符串，正则表达式从它编译而来。

```
r
.findall(string
[, pos
 [, endpos
]])
```

等效于findall() 函数。 *pos* 和 *endpos* 指定搜索的开始和结束位置。

```
r
.finditer(string
 [, pos
 [, endpos
]])
```

等效于finditer() 函数。 *pos* 和 *endpos* 指定搜索的开始和结束位置。

```
r
.match(string
[, pos
] [, endpos
])
```

检查在string 的开头是否有匹配的字符。 *pos* 和 *endpos* 指定要搜索的 *string* 范围。如果找到匹配值，则返回MatchObject ； 否则返回None 。

```
```

```
r
.search(string
[, pos
] [, endpos
])
```

在 *string* 中搜索匹配值。 *pos* 和 *endpos* 指定搜索的开始和结束位置。如果找到匹配值，则返回 *MatchObject* ； 否则返回None 。

```
r.split(string [, maxsplit
= 0])
```

等效于split() 函数。

```
r
.sub(repl
, string
 [, count
 = 0])
```

等效于sub() 函数。

```
r
.subn(repl
, string
[, count
```

```
 = 0])
```

等效于subn() 函数。

16.2.4 匹配对象

search() 和match() 返回的MatchObject 实例包含关于分组内容的信息，以及匹配值的位置数据。MatchObject 实例 *m* 具有以下方法和属性。

```
m
.expand(template
)
```

返回一个字符串，该字符串可通过在字符串 *template* 上置换正则表达式反斜杠来获得。数字反向引用（如"\1" 和"\2" ）和命名引用（如"\g<n>" 和"\g<name>" ）可替换为对应分组的内容。注意，这些序列应该使用原始字符串指定，或使用原义反斜杠字符（如r'\1' 或'\\1' ）指定。

```
m
.group([group1
, group2
, ...])
```

返回匹配值的一个或多个子分组。其中的参数指定分组编号或分组名称。如果未给定分组名称，则将返回整个匹配值；如果仅给定了一个分组，将返回一个字符串，其中包含与该分组匹配的文本；否则返回一个元组，其中包含与所请求的每个分组匹配的文本。如果给定的分组编号或名称无效，将抛出IndexError 。

```
m
```

```
.groups([default
])
```

返回一个元组，其中包含与模式中所有分组匹配的文本。 *default* 是针对未包含在匹配值中的分组的返回值（默认值为 *None* ）。

```
m
.groupdict([default
])
```

返回一个字典，其中包含所有匹配值的所有给定名称的分组。 *default* 是针对未包含在匹配值中的分组的返回值（默认值为None ）。

```
m
.start([group
])

m
.end([group
])
```

这两个方法返回与一个分组匹配的子字符串的开始和结束索引。如果省略 *group* ，将使用相匹配的整个子字符串。如果该分组存在，但未包含在匹配值中，则返回None 。

```
m
.span([group
```

```
])
```

返回一个二元组(*m* .start(*group*), *m* .end(*group*))。如果 *group* 未包含在匹配值中，则返回(None, None)。如果省略 *group* ，则使用整个匹配的子字符串。

```
m
.pos
```

传递给search()或match()函数的 *pos* 值。

```
m
.endpos
```

传递给search()或match()函数的 *endpos* 值。

```
m
.lastindex
```

相匹配的最后一个分组的数字索引。如果没有相匹配的分组，则为None 。

```
m
.lastgroup
```

相匹配的最后一个给定名称分组的名称。如果没有相匹配的给定名称分组，或者模式中没有给定名称的分组，则为None 。

```
m
.re
```

一个正则表达式对象，它的match() 或search() 方法生成此MatchObject 实例。

```
m
.string
```

传递给match() 或search() 的字符串。

16.2.5 示例

以下示例演示了如何使用re 模块在一个字符串中搜索、提取或替换文本模式。

```
import re
text = "Guido will be out of the office from 12/15/2012 - 1/3/2013."

# 日期的正则表达式模式
datepat = re.compile('(\d+)/(\d+)/(\d+)')

# 找到并打印所有日期
for m in datepat.finditer(text):
    print(m.group())

# 找到所有日期，但以一种不同的格式打印
monthnames = [None,'Jan','Feb','Mar','Apr','May','Jun',
              'Jul','Aug','Sep','Oct','Nov','Dec']
for m in datepat.finditer(text):
    print ("%s %s, %s" % (monthnames[int(m.group(1)], m.group(2), m.group(3)))

# 将所有日期替换为欧洲日期格式（日/月/年）
def fix_date(m):
    return "%s/%s/%s" % (m.group(2),m.group(1),m.group(3))
newtext = datepat.sub(fix_date, text)

# 一种可选的替换方法
newtext = datepat.sub(r'\2/\1/\3', text)
```

16.2.6 注意

- 关于正则表达式的理论和实现的详细信息，可以在讲解编译器构造的教材中找到。也可以参考Jeffrey Friedl撰写的*Mastering Regular Expressions*（O'Reilly & Associates，1997）一书。
- 使用`re`模块的最困难的地方是编写正则表达式模式。要编写模式，可以考虑使用Kodos（http://kodos.sourceforget.net ）等工具。

16.3 `string`

`string`模块包含许多用于操作字符串的有用常量和函数。它还包含用于实现新字符串格式的类。

16.3.1 常量

以下常量定义各种字符集，它们可以处理各种字符串操作。

常　　量	描　　述
`ascii_letters`	一个字符串，包含ASCII字母的所有小写和大写形式
`ascii_lowercase`	字符串`'abcdefghijklmnopqrstuvwxyz'`
`ascii_uppercase`	字符串`'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`
`digits`	字符串`'0123456789'`
`hexdigits`	字符串`'0123456789abcdefABCDEF'`
`letters`	`lowercase`和`uppercase`的串联形式
`lowercase`	包含特定于当前地区设置的所有小写字母的字符串
`octdigits`	字符串`'01234567'`
`punctuation`	由ASCII标点字符组成的字符串
`printable`	由可打印字符组成的字符串，是`letters`、`digits`、`punctuation`和`whitespace`的组合

uppercase	包含特定于当前地区设置的所有大写字母的字符串
whitespace	包含所有空格字符的字符串。这通常包括空格、制表符、换行、回车、换页和垂直制表符

注意，一些常量（如letters 和uppercase ）将因系统的地区设置不同而不同。

16.3.2 Formatter 对象

字符串的str.format() 方法用于执行高级字符串格式操作。第3章和第4章都已介绍，此方法可以访问序列或映射中的项、对象的属性和其他类型的相关操作。string 模块定义了一个Formatter 类，用于实现自定义格式操作。这个类公开了实现字符串格式操作的部分代码，允许对它们进行自定义。

```
Formatter()
```

创建一个新Formatter 实例。Formatter 的实例 *f* 支持以下操作。

```
f
.format(format_string
, *args
, **kwargs
)
```

格式化字符串format_string 。默认情况下，输出与调用 *format_string* .format(*_args_ , **_kwargs_) 的输出相同。例如， *f* .format("{name} is {0:d} years old", 39,name="Dave") 创建字符串"Dave is 39 years old" 。

```
f
.vformat(format_string
, args
```

```
, kwargs
)
```

此方法实际执行 *f* .format() 的工作。 *args* 是可能的参数元组， *kwargs* 是关键字参数字典。如果已经获取了元组或字典中的参数信息，那么这是一种更快的方法。

```
f
.parse(format_string
)
```

此函数创建一个迭代器来解析格式字符串 *format_string* 的内容。该迭代器扫描整个格式字符串，并生成格式元组(*literal _text , field_name , format_spec , conversion*)。literal_text 是大括号{ ... } 中下一个格式标识符之前的文字文本。如果没有前导文本，它是一个空字符串。 *field_name* 是一个字符串，指定格式标识符中的字段名称。例如，如果标识符为'{0:d}' ，那么字段名称为'0' 。 *format_spec* 是出现在冒号之后的格式标识符，在前面的例子中为'd' 。如果没有指定它，那么它将是一个空字符串。 *conversion* 是一个字符串，包含转换标识符（如果有）。在上面的例子中，它为None ，但是如果标识符为'{0!s:d}' ，它将会被设置为's' 。对于格式字符串的最后一个片段， *field_name* 、 *format_spec* 和 *conversion* 都将为None 。

```
f
.get_field(fieldname, args, kwargs
)
```

从 *args* 和 *kwargs* 中提取与给定 *fieldname* 关联的值。 *fieldname* 是一个字符串，如前面的parse() 方法返回的"0" 或"name" 。返回元组(*value , key*)，其中 *value* 是字段值， *key* 用于在 *args* 和 *kwargs* 中定位该值。如果 *key* 为整数，则表示 *args* 中的一个索引。如果它为一个字符串，则表示 *kwargs* 中使用的键。字段名可能包括附加索引和属性查找，如'0.name' 或'0[name]' 。在这种情况下，此方法将执

行额外的查找并返回适当的值。但是，返回的元组中 *key* 的值将设置为'0'。

```
f
.get_value(key, args, kwargs
)
```

从 *args* 或 *kwargs* 提取与 *key* 对应的对象。如果 *key* 为整数，将从 *args* 中提取对象。如果它为字符串，将从 *kwargs* 中提取。

```
f
.check_unused_args(used_args, args, kwargs
)
```

检查format()操作中未使用的参数。 *used_args* 是格式字符串中所有已使用的参数键集（参见get_field()）。 *args* 和 *kwargs* 是传递给format()的位置和关键字参数。默认行为是为未使用的参数抛出TypeError。

```
f
.format_value(value
, format_spec
)
```

根据给定的格式规范格式化一个字符串值。默认情况下，只是简单地执行内置函数format(value, format_spec)。

```
f
```

```
.convert_field(value, conversion
)
```

根据指定的转换编码，转换get_field() 返回的 *value* 。如果 *conversion* 为None，将不经修改地返回 *value* 。如果 *conversion* 为's' 或'r'，则分别使用str() 或repr() 将 *value* 转换为一个字符串。

如果希望创建自定义字符串格式，那么可以创建Formatter 对象，并使用默认方法来执行期望的格式操作。也可以定义一个继承自Formatter 的新类，重新实现前面介绍的任意方法。

关于格式标识符的语法和高级字符串格式的详细信息，请参阅第3章和第4章。

16.3.3 Template 字符串

string 模块定义了一种新字符串类型Template，简化了特定的字符串置换操作。可以在第9章找到一个示例。

下面的函数创建新的模板字符串对象。

```
Template(s
)
```

这里 *s* 是一个字符串，Template 定义为一个类。

Template 对象 *t* 支持以下方法。

```
t
.substitute(m [, **kwargs
])
```

此方法接受一个映射对象 *m* （如一个字典）或一个关键字参数列表，在字符串 *t* 上执行关键字置换。这一置换操作将字符串'$$' 替换为单个'$'，如果提供了关键字参

数，则分别将字符串'$*key* '或'${*key* }'替换为m['*key* ']或kwargs['*key* ']。key必须是有效的Python标识符。如果最终的字符串包含无法解析的'$*key* '模式，那么将抛出KeyError异常。

```
t
.safe_substitute(m
 [, **kwargs
])
```

与substitute()相似，但它不会生成任何异常或错误。相反，未解析的$*key* 引用将按原样保留在字符串中。

```
t
.template
```

包含传递给Template()的原始字符串。

可以修改Template类的行为，方法是构建它的子类并重新定义属性delimiter和idpattern。例如，下面这段代码将转义字符$更改为@，并将键名称限制为字母。

```
class MyTemplate(string.Template):
    delimiter = '@'           # 转义序列的文字字符
    idpattern = '[A-Z]*'       # 标识符正则表达式模式
```

16.3.4 实用工具函数

string模块还定义了另外两个用于操作字符串的函数，但是未定义为字符串对象的方法。

```
capwords(s
)
```

```

```

将 *s* 中每个单词的第一个字母变为大写形式，将重复的空格字符替换为一个空格，删除前导和结尾的空格。

```
maketrans(from, to

)
```

创建一个转换表，将 *from* 中的每个字符映射到 *to* 中相同位置的字符。 *from* 和 *to* 必须具有相同的长度。此函数用于创建适合与字符串的translate() 方法结合使用的参数。

16.4 struct

struct 用于在Python与二进制数据结构（表示为Python字节字符串）之间转换数据。这些数据结构通常在与使用C编写的函数、二进制文件格式、网络协议进行交互或通过串行端口进行二进制通信时使用。

16.4.1 打包和解包函数

以下模块级函数用于打包和解包字节字符串中的数据。如果程序需要反复执行这类操作，可以考虑使用下一节介绍的Struct 对象。

```
pack(fmt, v1, v2,

 ...)
```

根据 *fmt* 中的格式字符串，将值v1 、v2 等打包到字节字符串中。

```
pack_into(fmt, buffer, offset, v1, v2

 ...)
```

将值 *v1* 、 *v2* 等打包到可写入缓冲区对象 *buffer* 中从字节偏移量 *offset* 开

始的部分。此函数仅适用于支持缓冲区接口的对象。示例包括 *array.array* 和 bytearray 对象。

```
unpack(fmt, string
)
```

根据 *fmt* 中的格式字符串，解包字节字符串 *string* 中的内容。返回一个解包值元组。 *string* 的长度必须与calcsize() 函数确定的格式大小完全相同。

```
unpack_from(fmt, buffer, offset
)
```

根据 *fmt* 中的格式字符串解包 *buffer* 对象中从偏移量 *offset* 开始的内容。返回一个解包值元组。

```
calcsize(fmt
)
```

根据格式字符串 *fmt* 计算结构的字节数。

16.4.2 Struct 对象

struct 模块定义一个类Struct ，该类提供了另一种打包和解包接口。使用此类效率较高，因为只需解释一次格式字符串。

```
Struct(fmt
)
```

创建一个Struct 实例，表示根据给定格式编码解包的数据。Struct 的实例 *s* 具有

以下方法，它们的功能与上一节中介绍的对应方法相同。

方　　法	描　　述
s .pack(*v1, v2* , ...)	将值打包到字节字符串中
s .pack_into(*buffer,offset,v1,v2,* ...)	将值打包到缓冲区对象中
s .unpack(*bytes*)	解包字节字符串中的值
s .unpack_from(*buffer, offset*)	解包缓冲区对象中的值
s .format	使用的格式编码
s .size	格式的字节数

16.4.3 格式编码

struct 模块中使用的格式字符串是一系列字符，其含义如下：

格　　式	**C**类型	**Python**类型
'x'	pad byte	没有值
'c'	char	长度为1的字符串
'b'	signed char	整型
'B'	unsigned char	整型
'?'	_Bool (C99)	布尔值
'h'	short	整型
'H'	unsigned short	整型
'i'	int	整型

'I'	unsigned int	整型
'l'	long	整型
'L'	unsigned long	整型
'q'	long long	长整型
'Q'	unsigned long long	长整型
'f'	float	浮点型
'd'	double	浮点型
's'	char[]	字符串
'p'	char[]	长度为第一个字节中的值的字符串
'P'	void *	整型

每个格式字符都可以在前面添加上一个整数，以表示重复次数（例如，'4i' 等效于'iiii'）。对于's' 格式，该次数表示字符串的最大长度，所以'10s' 表示一个10字节长的字符串。格式'0s' 表示长度为0的字符串。'p' 格式用于编码一个字符串，其中的第一个字节表示字符串长度，后面才是字符串数据。这在处理Pascal编码时很有用，在Macintosh上有时必须使用这种编码。注意，在这种情况下，字符串的长度不得超过255个字符。

当使用'I' 和'L' 格式来解包值时，返回值是一个Python长整型值。此外，'P' 格式既可以返回整型值，也可以返回长整型值，具体取决于机器使用的字长。

每个格式字符串的第一个字符还可以指定字节顺序和打包数据的对齐方式，如下所示。

格　　式	字节顺序	大小和对齐方式
'@'	本机	本机
'='	本机	标准
'<'	小尾	标准

'>'	大尾	标准
'!'	网络（大尾）	标准

本机字节顺序既可以是小尾，也可以是大尾，具体取决于机器体系结构。本机大小和对齐方式与C编译器使用的值对应，与具体实现密切相关。标准对齐方式假设不需要进行任何类型的对齐。标准大小假设`short` 为2字节，`int` 为4字节，`long` 为4字节，`float` 为32位，`double` 为64位。`'P'` 格式仅能使用本机字节顺序。

16.4.4 注意

- 有时候，根据特定类型的对齐要求，必须对齐结构的末尾。为此，使用该类型的代码来结束结构格式字符串，其中重复次数为0。例如，格式`'llh0l'` 指定在4字节边界处结束的结构（假设长整型值在4字节边界处对齐）。在这种情况下，将在`'h'` 编码指定的短值后插入两个填充字节。这仅适用于使用本机大小和对齐方式，标准大小和对齐方式不会强制实施对齐规则。
- 如果用于编译Python的C编译器支持`long long` 数据类型，那么`'q'` 和`'Q'` 格式仅适用于“本机”模式。

另请参见： 15.2节和26.3节。

16.5 unicodedata

`unicodedata` 模块提供了访问Unicode字符数据库的方式，它包含所有Unicode字符的字符属性。

```
bidirectional(unichr
)
```

以字符串形式返回分配给 *unichr* 的双向类别，如果没有定义这类值，则返回一个空字符串。返回以下值之一。

值	描　　述	值	描　　述
L	从左到右	ET	欧洲数字终止符
LRE	从左到右嵌入	AN	阿拉伯数字

LRO	从左到右覆盖	CS	通用数字分隔符
R	从右到左	NSM	非间距标记
AL	从右到左（阿拉伯字符）	BN	边界不确定
RLE	从右到左嵌入	B	段落分隔符
RLO	从右到左覆盖	S	部分分隔符
PDF	弹出方向格式	WS	空格
EN	欧洲数字	ON	其他不确定性
ES	欧洲数字分隔符		

category(*unichr*

)

返回一个字符串，描述 *unichr* 的一般类别。返回的字符串为以下值之一。

值	描　　述	值	描　　述
Lu	字母，大写	Cf	其他，格式
Ll	字母，小写	Cs	其他，替代
Lt	字母，标题大写形式	Lm	字母，修饰符
Mn	标记，非空格	Lo	字母，其他
Mc	标记，空格组合	Pc	标点符号，连字符
Me	标记，封闭类型	Pd	标点符号，破折号

Nd	数字，十进制数	Ps	标点符号，左侧标点
Nl	数字，字母	Pe	标点符号，右侧标点
No	数字，其他形式	Pi	标点符号，左侧引号
Zs	分隔符，空格	Pf	标点符号，右侧引号
Zl	分隔符，行	Po	标点符号，其他
Zp	分隔符，段落	Sm	符号，数学
Cc	其他，控制	Sc	符号，货币
Co	其他，专用	Sk	符号，修饰符
Cn	其他，未分配	So	符号，其他

combining(*unichr*

)

返回一个整数，描述 *unichr* 的组合类型，如果未定义组合类型，则返回0。返回以下值之一。

值	描　　述	值	描　　述
0	间距、拆分、围绕、环绕（reordrant）和附加的藏文	214 216	附加到上侧 附加到右上侧
1	覆盖和内部组合	218	左下侧
7	雷布查文（Nuktas）	220	下侧
8	平假名/片假名音标	222	右下侧

9	Viramas	224	左侧
10 - 199	固定位置类型	226	右侧
200	附加到左下侧	228	左上侧
202	附加到下侧	230	上侧
204	附加到右下侧	232	右上侧
208	附加到左侧	233	下侧两端
210	附加到右侧	234	上侧两端
212	附加到左上侧	240	下侧（下标）

```
decimal(unichr
[, default
])
```

返回分配给 *unichr* 的十进制整数值。如果 *unichr* 不是十进制数，则返回 *default* 或抛出ValueError 。

```
decomposition(unichr
)
```

返回包含 *unichr* 的分解映射的字符串，如果没有定义此类映射，则返回空字符串。通常，包含重音符号的字符可以被分解为多字符序列。例如，decomposition(u"\u00fc")("ü") 返回字符串"0075 0308" ，表示字母u 和变音符号(¨) 重音符号。此函数也可能返回以下字符串。

值	描　述

<font>	字体变形（如黑体形式）
<noBreak>	空格或连字符的非断开版本
<initial>	原始显示形式（阿拉伯字符）
<medial>	中间显示形式（阿拉伯字符）
<final>	最终显示形式（阿拉伯字符）
<isolated>	单独显示形式（阿拉伯字符）
<circle>	环绕形式
<super>	上标形式
<sub>	下标形式
<vertical>	垂直布局显示形式
<wide>	广义（或zenkaku）兼容字符串
<narrow>	狭义（或hankaku）兼容字符
<small>	小型变形形式（CNS兼容性）
<square>	CJK宋体变形
<fraction>	普通分数形式
<compat>	其他未指定的兼容字符

digit(*unichr*

[, *default*

])

返回分配给 *unichr* 的整数值。如果 *unichr* 不是数字，则返回 *default* 或抛出 ValueError 。此函数与decimal() 不同，因为它处理可能表示数字但并不是十进制数字的字符。

```
east_asian_width(unichr
)
```

返回分配给 *unichr* 的东亚宽度字符。

```
lookup(name
)
```

按名称查找字符。例如，lookup('COPYRIGHT SIGN') 返回相应的Unicode字符。常见名称可以在http://www.unicode.org/charts 上找到。

```
mirrored(unichr
)
```

如果 *unichr* 是双向文本中的一个“镜像”字符，则返回1 ；否则返回0 。如果文本按相反顺序显示，那么为了正确显示，镜像字符的外观可能会改变。例如，字符'(' 是镜像字符，因为在从右向左输出文本时，应该将其翻转为')' 才合理。

```
name(unichr
 [, default
])
```

返回一个Unicode字符 *unichr* 的名称。如果没有定义任何名称，则抛出ValueError；如果提供了名称，则返回 *default* 。例如，name(u'\xfc')返回'LATIN SMALL LETTER U WITH DIAERESIS'。

```
normalize(form
, unistr
)
```

根据规范形式 *form* 规范化Unicode字符串 *unichr* 。 *form* 是'NFC'、'NFKC'、'NFD'和'NFKD'之一。字符串的规范化在一定程度上可归类于某些字符的组合和分解。例如，单词“resumé”的Unicode字符串可以表示为u'resum\u00e9'或u'resume\u0301'。在第一个字符串中，重音字符é表示为一个字符。在第二个字符串中，重音字符表示为字母e和一个组合重音符号（´）。使用'NFC'规范化转换字符串 *unichr* 时，每个字符都是一个整体（例如，é是一个字符）。使用'NFD'规范化转换 *unichr* 时，字符会被分解（é表示为字母e和重音符号）。'NFKC'和'NFKD'的作用与'NFC'和'NFD'相似，但前者会将某些可表示为多个Unicode字符值的字符转换为一个标准值。例如，罗马数字具有自己的Unicode字符值，但它们也可以使用拉丁字母*I*、*V*、*M*等表示。'NFKC'和'NFKD'会将特殊的罗马数字字符转换为对应的拉丁字母。

```
numeric(unichr
[, default
])
```

以浮点数字形式返回分配给Unicode字符 *unichr* 的值。如果没有定义数字值，则返回 *default* 或抛出ValueError。例如，U+2155（分数“1/5”的字符形式）的数字值为0.2。

```
unidata_version
```

一个字符串，包含所用的Unicode数据库版本（如'5.1.0'）。

注意

关于Unicode字符数据库的更多细节，请访问http://www.unicode.org 。

第17章　Python数据库访问

本章描述用于与关系型和散列表型数据库交互的Python编程界面。与描述特定库模块的其他章节不同，本章内容会部分涉及第三方扩展。例如，如果你希望Python与MySQL或Oracle数据库交互，必须先下载一个第三方扩展模块。这个模块也必须遵守下述基本规定。

17.1　关系数据库API规范

对于关系数据库的访问，Python社区已经制定出一个标准，称为Python Database API Specification V2.0，简称PEP 249（正式描述参见http://www.python.org/dev/peps/pep-249/）。MySQL、Oracle等特定数据库模块遵从这一规范，而且可添加更多特性。本节介绍将其用于大多数应用程序所需的基本要素。

数据库API定义了一组用于连接数据库服务器、执行SQL查询并获得结果的高级函数和对象。其中有两个主要的对象：一个是用于管理数据库连接的`Connection` 对象，另一个是用于执行查询的`Cursor` 对象。

17.1.1　连接

为了连接到数据库，每个数据库模块都提供了一个模块级函数`connect(`*`parameters`*`)`。其中实际使用的参数因数据库不同而不同，但是通常包含数据源名称、用户名、密码、主机名称和数据库名称等信息。通常情况下，这些信息分别通过关键字参数`dsn`、`user`、`password`、`host` 和`database` 提供。所以，可以这样调用`connect()`：

```
connect(dsn="hostname:DBNAME",user="michael",password="peekaboo")
```

如果成功，返回`Connection` 对象。`Connection` 的实例 *`c`* 有如下方法：

```
c
.close()
```

关闭与服务器的连接。

```
```

```
c
.commit()
```

将所有未完成的事务提交到数据库。如果数据库支持事务处理，那么要使任何变更生效都必须调用这一方法。如果底层数据库不支持事务处理，这一方法没有任何作用。

```
c
.rollback()
```

将数据库回滚到未完成事务的开始状态。有时，这一方法可用于不支持事务处理的数据库，以撤销对数据库做出的更改。例如，如果在更新数据库的过程中代码发生异常，可以使用这个方法来在异常出现之前撤销对数据库做出的更改。

```
c
.cursor()
```

创建一个使用连接的新的Cursor 对象。游标 （cursor）是一个对象，你可以使用它执行SQL 查询并获得结果。这将在下一部分中介绍。

17.1.2 Cursor

为了在数据库上执行操作，首先必须创建一个连接 *c* ，然后调用 *c* .cursor() 方法创建Cursor 对象。Cursor 的实例 *cur* 有一些用于执行查询的标准方法和属性：

```
cur
.callproc(procname
 [, parameters
])
```

调用一个名为 *procname* 的存储过程， *parameter* 是一个序列的值，用作该过程的参数。函数的结果也是一个序列，项数与 *parameters* 相同。它本来是 *parameters* 的副本，函数执行后，每一个输出参数的值都被修改值替换。如果该过程还生成一组输出，可以使用fetch*() 方法来读取。这将在后面介绍。

```
cur
.close()
```

关闭游标，防止再对其进行任何操作。

```
cur
.execute(query
 [, parameters
])
```

在数据库上执行查询或 *query* 命令。 *query* 是一个包含命令（通常是SQL）的字符串， *parameters* 是一个序列或映射，用于为查询字符串中的变量赋值（下一部分介绍）。

```
cur
.executemany(query [, parametersequence
])
```

重复执行查询或命令。 *query* 是一个查询字符串， *parametersquence* 是一个参数序列。这一序列的每一项都是一个序列或映射对象，它们均可以用于上面提到的 execute() 方法。

```
cur
.fetchone()
```

返回由execute() 或executemany() 生成的下一行结果集。这一结果通常是列表或元组，包含结果中的不同列的值。如果没有更多的行，返回None 。如果没有未处理完的结果或者如果以前执行的操作没有生成结果集，就会提示异常。

```
cur
.fetchmany([size
])
```

返回结果行的序列（如元组列表）。 *size* 是要返回的行数。如果省略，cur.arraysize 的值就会作为默认值使用。实际返回的行数可能比请求的少。如果没有更多的行，就会返回空的序列。

```
cur
.fetchall()
```

返回全部剩余结果行的序列（如元组列表）

```
cur
.nextset()
```

放弃当前结果集中的所有剩余行，跳至下一个结果集（如果有）。如果没有更多的结果集，返回None ；否则返回True ，接着通过fetch*() 操作从新集合中返回数据。

```
cur
.setinputsize(sizes
)
```

给游标一个提示，说明要在接下来的execute*()方法中传递的参数。 *sizes* 是一个简短描述类型对象的序列或者是代表每一参数预期最高字符串长度的整数序列。在内部，这用于预定义内存缓冲区，来创建发送至数据库的查询和命令。使用这个方法能够加速接下来的execute*()操作。

```
cur
.setoutputsize(size
 [, column
])
```

为特定的列设定缓冲区容量。column是结果行的整数索引，而size是字节的数目。这种方法通常用于在调用execute*()之前，在一个大型的数据库列（如字符串、BLOB和LONG）上设定上限。如果 *column* 省略，则为结果中的所有列设定的上限。

游标有一系列属性，用来描述当前的结果集，并提供关于游标本身的信息。

```
cur
.arraysize
```

为fetchmany()操作提供默认的一个整数值。该值因数据库模块的不同而有所不同，其初始值可能设置为模块认为的“最佳”值。

```
cur
```

.description

提供当前结果集中的每一列的信息的一个元组序列。每一个元组的形式为(*name, type_code , display_size , internal_size , precision , scale, null_ok*)。第一个字段肯定会有定义并使之与列名称相对应。 *type_code* 可以用于涉及类型对象（在17.7.4节介绍）的比较。其他字段如果不适用该列，可以设置为None 。

cur

.rowcount

表示由一种execute*() 方法生成的最后结果中的行数。如果设置为-1，意味着没有结果集，或者行数不能确定。

虽然没有在规范中要求，但大多数数据库模块中的Cursor 对象也会实现迭代协议。在这种情况下，像for *row* in *cur* ：这样的一个语句将迭代由最后一次execute*() 方法生成的结果集的行。

这里有一个简单的例子，告诉你如何通过sqlite3 数据库模块使用这些操作，sqlite3 数据库模块是一个内置库：

```
import sqlite3
conn = sqlite3.connect("dbfile")
cur = conn.cursor()

# 简单查询示例
cur.execute("select name, shares, price from portfolio where account=12345")

# 循环结果
while True:
    row = cur.fetchone()
          if not row: break
          # 处理行
          name, shares, price = row
          ...

# 一种替代方法（使用迭代）
cur.execute("select name, shares, price from portfolio where account=12345")
for name, shares, price in cur:
        # 处理行
        ...
```

17.1.3 生成查询

使用数据库API的关键是生成SQL查询字符串将其传递到游标对象的execute*()方法中。问题是，你需要根据用户提供的参数填充查询字符串的各部分。例如，可以这样编写代码：

```
symbol = "AIG"
account = 12345

cur.execute("select shares from portfolio where name='%s' and account=%d" %
            (symbol, account))
```

虽然这“能运行”，但决不能这样使用Python字符串操作手动生成查询。如果这样做，代码将可能受到SQL注入攻击。攻击者可利用这个漏洞在数据库服务器上随意执行语句。例如，在上面的代码中，有些人可能为symbol提供了类似于"EVIL LAUGH'; drop table portfolio;--"的值，而这可能得到出乎意料的结果。

所有的数据库模块在值替换上都有各自的机制。例如，下面这段代码，与生成整个查询不同，你可能会这样做：

```
  symbol = "AIG"
account = 12345

cur.execute("select shares from portfolio where name=? and account=?",
            (symbol, account))
```

这里，占位符‘?’后来被元组(symbol, account)中的值替换。

然而，各个数据库模块实现中没有关于占位符的标准规则。但是，每一个模块都定义了一个变量paramstyle，它描述了将在查询中使用的值替换格式。这一变量可能的值如下。

参数类型	描　　述
'qmark'	问号类型，查询中的每一个?都被序列中连续的项代替。例如， *cur* .execute("...where name=? and account=?", (symbol, account))。参数规定为一个元组
'numeric'	数字类型，:*n* 由索引 *n* 中的参数值填充。例如， *cur*. execute("... where name=:0 and account=:1",(symbol, account))
'named'	命名类型，:*name* 由命名值填充，这个数据类型的参数必须作为映射给出。例如，cur.execute ("... where name=:symbol andaccount=:account", {'symbol':symbol, 'account': account})
'format'	Printf-style格式代码，如%s、%d等。例如， *cur* .execute("... where name=%s and account=%d",(symbol, account))

'pyformat'	Python扩展格式代码，如%(name)s 。与'named' 类型相似。参数必须规定为映射而不是元组

17.1.4 类型对象

当使用数据库的数据时，内置类型（如整数和字符串）通常以相应类型映射到数据库中。然而，对于日期、二进制数据和其他特殊类型，数据管理变得棘手。为了辅助这一映射，数据库模块实现了一组构造函数，用于创建各种类型的对象。

```
Date(year, month, day
)
```

创建表示日期的对象。

```
Time(hour, minute, second
)
```

创建表示时间的对象。

```
Timestamp(year, month, day, hour, minute, second
)
```

创建表示时间戳的对象。

```
DateFromTicks(ticks
)
```

根据系统时间创建日期对象。 *ticks* 是秒数，就像函数time.time() 返回的一样。

```
TimeFromTicks(ticks
)
```

根据系统时间创建时间对象。

```
TimestampFromTicks(ticks
)
```

根据系统时间创建时间戳对象。

```
Binary(s
)
```

根据字节字符串 *s* 创建二进制对象。

除了这些构造函数之外，可能还定义了如下类型的对象。这些代码的目的是对 cur.description 的type_code 字段执行类型检查。该字段描述了当前结果集的内容。

类型对象	描　　述
STRING	字符或文本数据
BINARY	二进制数据，如BLOB
NUMBER	数字数据
DATETIME	日期和时间数据
ROWID	行ID数据

17.1.5 错误处理

数据库模块定义了一个高级异常Error，作为所有其他错误的基类。下列的异常是更加具体的数据库错误。

异　　常	描　　述
InterfaceError	与数据库接口相关的错误。但是不是数据库本身
DatabaseError	与数据库本身相关的错误
DataError	与处理的数据相关的错误。例如，类型转换错误、除零等
OperationalError	与数据库本身的运行相关的错误。例如，丢失连接
IntegrityError	当数据库的关系完整性被破坏时出现的错误
InternalError	数据库内部错误。例如，如果是一个失效游标
ProgrammingError	SQL查询中的错误
NotSupportedError	不受底层数据库支持的数据库API方法导致的错误

模块也可定义一个Warning 异常，由数据库模块使用，就更新过程中出现的数据截断等事件发出警告。

17.1.6 多线程

如果将数据库访问与多线程混合，底层的数据库模块可能是线程安全的，也可能不安全。下列变量已在每一个模块中定义，以提供更多的信息。

```
threadsafety
```

这是一个说明模块线程安全性的整数。其值可能是以下几个。

0　没有线程安全。线程不能共享模块的任何部分。

1　该模块是线程安全的。但是连接是不能共享的。

2　模块和连接都是线程安全的，但是游标是不能共享的。

3　模块、连接和游标都是线程安全的。

17.1.7　将结果映射到字典中

关于数据库结果的一个常见问题是：要将元组或列表映射到命名字段字典中。例如，如果查询的结果集包含大量的列，使用描述性字段名称来处理数据就更加容易，而不用通过硬编码元组中特定字段的数字索引来实现。

有许多方式解决这个问题，但是处理结果数据最好的方式是使用生成器函数。例如：

```
def generate_dicts(cur):
    import itertools
    fieldnames = [d[0].lower() for d in cur.description ]
    while True:
         rows = cur.fetchmany()
        if not row: return
         for row in rows:
             yield dict(itertools.izip(fieldnames,row))

# 使用样例
cur.execute("select name, shares, price from portfolio")
for r in generate_dicts(cur):
    print r['name'],r['shares'],r['price']
```

注意，列的名称在不同的数据库中并不完全一致——特别是否区分大小写之类的事情。所以，你在使用这一技巧来编写与多个不同的数据库模块一起使用的代码时必须小心。

17.1.8　数据库API扩展

最后，许多扩展功能和高级功能可以添加到特定的数据库模块。例如，对两阶段提交和扩展的错误处理的支持。PEP-249包含更多有关这些功能的建议接口的信息，供高级用户查阅。第三方库模块也可简化关系数据库界面的使用。

17.2　sqlite3 模块

sqlite3 模块提供了访问SQLite 数据库存储库的Python接口（http://www.sqlite.org ）。SQLite 是一个在文件或内存中实现自我包含关系数据库的C库。虽然简单，但是由于各种原因这个库很有吸引力。首先，它不需要独立的数据库服务器，也不要求任何特定的配置，只要与数据库文件连接，就可以在程序中立即开始使用（如果它不存在，会生成一个新文件）。这个数据库也支持用于改进可靠性的事务处理（甚至在系统崩溃期间也可以），还支持lock ，允许多个进程同时访问同一数据库文件。

该库的编程接口同样遵守上一节介绍的数据库API中的规定，细节在这里就不一一赘述了。这一部分重点介绍使用这一模块的技术细节，以及sqlite3 模块特有的功能。

17.2.1 模块级函数

sqlite3 模块定义了下列函数：

```
connect(database
 [, timeout
 [, isolation_level
[, detect_types
]]])
```

创建到SQLite 数据库的连接。 *database* 是指定数据库文件名称的字符串。也可以是一个字符串":memory:"，此时使用内存中的数据库（注意这种数据库仅在Python进程运行时存在，并在程序退出的时候丢失）。 *timeout* 参数指明，当其他连接在更新数据库时，等待内部的读写锁（reader-writer lock ）释放的时间。默认情况下，*timeout* 是5秒。当使用INSERT 或UPDATE 等SQL语句时，如果一个事务没有生效，则自动开始一个新的事务。 *isolation_level* 参数是一个字符串，为用于开始这一事务的底层SQL BEGIN 语句提供可选修饰符，可选值是"" （默认）、"DEFERRED" 、"EXCLUSIVE" 或"IMMEDIATE" 。这些设置的含义与底层数据库锁相关，如下所示。

隔离级别	描　　述
"" （空字符串）	使用默认设置（DEFERRED ）
"DEFERRED"	开始一个新的事务，但在第一次数据库操作执行之前，不能获得锁
"EXCLUSIVE"	开始一个新的事务，并保证其他连接不能读取或写入数据库，直到变更提交
"IMMEDIATE"	开始一个新的事务，并保证其他连接不能对数据库做出修改，直到变更提交。但是其他连接仍然可以读取数据库

当返回结果时， *detect_types* 参数可实现某些额外类型的检测（通过对SQL查询的额外解析实现）。默认值是*0* （意味着没有额外检测）。可以将其设置为PARSE_DECLTYPES 和PARSE_COLNAMES 的按位或（OR）。如果PARSE_DECLTYPES 启用，审核查询SQL类型名称（如"integer" 或"number(8)" ）以便确定结果列的类型。如果PARSE_COLNAMES 启用，"*colname* [*typename*]" 形式的特定字符串（含双引号）可以嵌入查询， *colname* 是列名称， *typename* 是通过register_converter() 函数

注册的类型名称，该函数将在下面介绍。这些字符串在传递到SQLite引擎时仅转换成 *colname* ，但是当转换查询结果中的值时，将使用额外类型的说明符。例如，'select price as "price [decimal]" from portfolio' 查询被解释成'select price as price from portfolio'，其结果将根据“decimal”转换规则转换。

```
register_converter(typename, func
)
```

注册新的类型名称，供connect() 的detect_types 选项使用。typename 是将在查询中使用的包含类型名的字符串，func 是一个函数，它接受单一的字节字符串作为输入，结果返回一个Python数据类型。

例如，如果调用sqlite3.register_converter('decimal', decimal.Decimal)，那么可以通过写入查询（如'select price as "price [decimal]" from stocks'）将查询中的值转换成Decimal 对象。

```
register_adapter(type, func
)
```

为Python类型 *type* （用于在数据类型中存储该类型的值）注册一个适配器函数。*func* 是一个函数，接受 *type* 类型的实例作为输入，结果返回一个int 、float 、UTF-8编码的字节串、Unicode字符串或者buffer。例如，如果要存储Decimal 对象，可以使用sqlite3.register_adapter(decimal.Decimal, float) 。

```
complete_statement(s
)
```

如果字符串 *s* 代表由分号分隔的一个或一个以上的完整SQL语句，则返回True 。这也许会在编写读取来自用户的查询的交互式程序时用到。

```
enable_callback
_tracebacks(flag
```

```
)
```

处理在转换器和适配器等用户定义回调函数中出现的异常。默认情况下，忽略异常。如果 *flag* 设置为True，回溯消息将输出到sys.stderr。

17.2.2 连接对象

由connect()函数返回的Connection对象 *c* 支持在数据库API中描述的标准操作。此外，针对sqlite3模块提供下列方法。

```
c.create_function(name, num_params, func
)
```

创建能够在SQL语句中使用的用户定义函数。 *name* 是包含函数名称的字符串，*num_params* 是指明参数数量的整数， *func* 是提供实现的Python函数，这里有一个简单的例子：

```
def toupper(s):
    return s.upper()
c.create_function("toupper",1,toupper)
# 查询使用样例
c.execute("select toupper(name),foo,bar from sometable")
```

虽然定义了一个Python函数，但函数的参数和函数的输入只能是int、float、str、unicode、buffer或None。

```
c
.create_aggregate(name, num_params, aggregate_class
)
```

创建可在SQL语句中使用的用户定义聚合函数。 *name* 是包含函数名称的字符串，*num_params* 是一个整数，说明输入的参数数量。 *aggregate_class* 是实现聚合操作

的类。这个类必须支持不带参数的初始化，实现接受与 *num_params* 中给出的参数数量相同的step(*params*)方法，并实现返回最后结果的finalize()方法。这里有一个简单的例子：

```
class Averager(object):
    def __init__(self):
        self.total = 0.0
        self.count = 0
    def step(self,value):
        self.total += value
        self.count += 1
    def finalize(self):
        return self.total / self.count

c.create_aggregate("myavg",1,Averager)
# 查询使用样例

c.execute("select myavg(num) from sometable")
```

通过重复调用带输入值的step()方法进行聚合，然后调用finalize()获得最终值。

```
c
.create_collation(name, func
)
```

注册可在SQL语句中使用的用户定义校验函数。 *name* 是包含校验函数名称的字符串， *func* 是一个函数，它接受两个输入，并根据第一个输入是小于、等于还是大于第二个输入来返回-1、0、1。你可以通过SQL表达式（如"select * from *table* order by *colname* collate *name* "）来使用这个用户定义函数。

```
c
.execute(sql
[, params
])
```

使用 *c* .cursor() 创建游标对象、并通过 *params* 参数和 *sql* 中的SQL语句执行游标execute() 方法的快捷方法。

```
c
.executemany(sql
 [, params
])
```

使用 *c* .cursor() 创建游标对象、并通过 *params* 参数和 *sql* 中的SQL语句执行游标executemany() 方法的快捷方法。

```
c
.executescript(sql
)
```

使用 *c* .cursor() 创建游标对象、并通过 *params* 参数和 *sql* 中的SQL语句执行游标的execute-script() 方法的快捷方法。

```
c
.interrupt()
```

终止正在连接上执行的查询。应该从另一个线程调用。

```
c
.iterdump()
```

返回一个迭代器，以便将整个数据库的内容转储到一系列SQL语句中，执行这些语句能够重建数据库。这可以用于任何需要导出数据库、或者需要将内存数据库转储到一个文件以便稍后进行恢复的情况。

```
c
.set_authorizer(auth_callback
)
```

注册一个授权回调函数，它在每次访问数据中的一列数据时执行。这个回调函数必须接受5个参数，即 *auth_callback* (*code, arg1, arg2, dbname, innername*)。如果允许访问，这个回调函数返回的值是SQLITE_OK，如果SQL语句由于出错而失败，返回值是SQLITE_DENY，如果列被处理成Null 值而忽略，返回值是SQLITE_IGNORE 。第一个参数 *code* 是整数操作代码。 *arg1* 和 *arg2* 是参数，它们的值取决于 *code* 。 *dbname* 是包含数据库名的字符串（通常是"main" ）， *innername* 是正在尝试访问的最内部视图或触发器的名称，如没有视图或触发器处于激活状态时为None 。下面的表格列出了 *code* 的值和 *arg1* 和 *arg2* 参数的涵义。

code	arg1	arg2
SQLITE_CREATE_INDEX	索引名	表名
SQLITE_CREATE_TABLE	表名	None
SQLITE_CREATE_TEMP_INDEX	索引名	表名
SQLITE_CREATE_TEMP_TABLE	表名	None
SQLITE_CREATE_TEMP_TRIGGER	触发器名	表名
SQLITE_CREATE_TEMP_VIEW	视图名	None
SQLITE_CREATE_TRIGGER	触发器名	表名
SQLITE_CREATE_VIEW	视图名	None

SQLITE_DELETE	表名	None
SQLITE_DROP_INDEX	索引名	表名
SQLITE_DROP_TABLE	表名	None
SQLITE_DROP_TEMP_INDEX	索引名	表名
SQLITE_DROP_TEMP_TABLE	表名	None
SQLITE_DROP_TEMP_TRIGGER	触发器名	表名
SQLITE_DROP_TEMP_VIEW	视图名	None
SQLITE_DROP_TRIGGER	触发器名	表名
SQLITE_DROP_VIEW	视图名	None
SQLITE_INSERT	表名	None
SQLITE_PRAGMA	Pragma 名	None
SQLITE_READ	表名	列名
SQLITE_SELECT	None	None
SQLITE_TRANSACTION	None	None
SQLITE_UPDATE	表名	列名
SQLITE_ATTACH	文件名	None
SQLITE_DETACH	数据库名	None
SQLITE_ALTER_TABLE	数据库名	表名
SQLITE_REINDEX	索引名	None
SQLITE_ANALYZE	表名	None

SQLITE_CREATE_VTABLE	表名	模块名
SQLITE_DROP_VTABLE	表名	模块名
SQLITE_FUNCTION	函数名	None

```
c
.set_progress_handler(handler, n
)
```

注册回调函数，每n 条SQLite虚拟机指令执行一次。 *handler* 是一个没有参数的函数。

连接对象上也定义了下列属性。

```
c
.row_factory
```

调用这个函数来创建代表每一个结果行的内容的对象。这个函数接受两个参数：用来获得结果的游标对象和带有原始结果行的元组。

```
c
.text_factory
```

调用这个函数来创建代表数据库中文本值的对象。这个函数只能接受UTF-8编码的字节字符串参数。返回值应该是某种类型的字符串。默认情况下返回Unicode字符串。

```
c

.total_changes
```

一个整数，代表自数据库连接打开后发生更改的行数。

连接对象的最后一个功能是，它们可以与上下文管理器协议一起使用，以便自动处理事务。例如：

```
conn = sqlite.connect("somedb")
with conn:
     conn.execute("insert into sometable values (?,?)", ("foo","bar")))
```

在这个例子中，commit() 是在所有with 数据块中的语句执行完毕并且没有错误之后自动执行的，如果出现任何异常，将执行rollback() 操作，再次提示异常。

17.2.3 游标和基本操作

为了在sqlite3 数据库上执行基本操作，首先必须使用cursor() 方法在一个连接上创建游标对象。然后使用该游标的execute() 、executemany() 或executescript() 方法执行SQL语句。有关这些方法常用操作的更多信息，请参阅API部分。此处不再赘述。这一部分主要是展示一组常用的数据库用例和编码样例。目的是为那些需要简要了解语法的编程人员展示游标对象的操作和一些通用SQL操作。

1. 创建新的数据库表

下面代码显示如何打开一个数据库并创建一个新的表：

```
import sqlite3
conn = sqlite3.connect("mydb")
cur = conn.cursor()
cur.execute("create table stocks (symbol text, shares integer, price real)")
conn.commit()
```

当定义表的时候，应使用一些原始的SQLite数据类型：text 、integer 、Real 和blob 。blob 类型是一个字节字符串，而text 类型是UTF-8编码的Unicode 。

2. 向表中插入新的值

下列代码显示如何向表中插入新的项：

```
import sqlite3
conn = sqlite3.connect("mydb")
```

```
cur = conn.cursor()
cur.execute("insert into stocks values (?,?,?)",('IBM',50,91.10))
cur.execute("insert into stocks values (?,?,?)",('AAPL',100,123.45))
conn.commit()
```

当插入值时，应当一直像上面那样使用**?** 替代。每一个**?** 用参数元组中的值来代替。

如果需要插入一系列数据，可以像这样使用游标的executemany() 方法：

```
stocks = [ ('GOOG',75,380.13),
          ('AA',60,14.20),
          ('AIG',125, 0.99) ]
cur.executemany("insert into stocks values (?,?,?)",stocks)
```

3. 更新现有行

下面的代码说明如何为现有行更新列：

```
cur.execute("update stocks set shares=? where symbol=?",(50,'IBM'))
```

同样，当需要向SQL语句中插入值时，确保使用**?** 占位符并且提供一个元组值作为参数。

4. 删除行

下面的代码显示如何删除行：

```
cur.execute("delete from stocks where symbol=?",('SCOX',))
```

5. 执行基本查询

下面的代码显示如何执行基本查询并获得结果：

```
# 选择表中所有的列
for row in cur.execute("select * from stocks"):
    statements



# 选择若干列
for shares, price in cur.execute("select shares,price from stocks"):
    statements



# 选择匹配行
for row in cur.execute("select * from stocks where symbol=?",('IBM',))
    statements
```

```
# 按照顺序选择匹配行
for row in cur.execute("select * from stocks order by shares"):
    statements



# 逆向选择匹配行
for row in cur.execute("select * from stocks order by shares desc"):
    statements



# 以通用列名(符号)联接表
for row in cur.execute("""select s.symbol, s.shares, p.price
                          from stocks as s, prices as p using(symbol)"""):
    statements
```

17.3 DBM风格的数据库模块

Python内置了一系列支持UNIX DBM风格的数据库文件的库模块。Python支持这些数据库的几种标准类型。dbm 模块用于读取标准的UNIX-dbm数据库文件。gdbm 模块用于读取GNU dbm数据库文件（http://www.gnu.org/software/gdbm ）。dbhash 模块用于读取由Berkeley DB库创建的数据库文件（http://www.oracle.com/database/berkeley-db/index.html ）。dumbdbm 模块是纯Python模块，独立实现一个简单的DBM风格数据库。

所有这些模块提供一个对象实现基于字符串的持久性字典。也就是说，除了所有关键字和值只能是字符串之外，它和Python字典一样。数据库文件通常使用各种open() 函数打开。

```
open(
filename



 [,
flag



 [,
mode



]])
```

这个函数打开数据库文件 *filename* 并且返回一个数据库对象。当 *flag* 为'r' 时表示只读访问，'w' 表示读写访问，'c' 表示如果数据库不存在便生成数据库，'n' 表示强制性生成新的数据库。 *mode* 是创建数据库时使用的整数文件访问模式（在UNIX上默认是0666 ）。

这个由open() 函数返回的对象至少支持下列字典类操作。

操　　作	描　　述
d [*key*] = *value*	插入 *value* 到数据库
value = *d* [*key*]	从数据库中获得数据
del *d* [*key*]	移除数据库条目
d .close()	关闭数据库
key in *d*	测试密钥
d .sync()	写出随数据库做出的所有更改

特定的实现可能也会添加额外的特性（参考相应的的模块了解详细信息）。

应用多种DBM风格的数据库模块存在一个问题：并不是每一个模块都安装在所有平台上。例如，如果在Windows上使用Python，dbm 和gdbm 模块通常是不可用的。但是程序可能仍然需要创建一个DBM风格的数据库来供自己使用。为了解决这个问题，Python提供了一个anydbm 模块，用于打开并创建DBM风格的数据库文件。这个模块提供了一个如上面所述的open() 函数，但是它能够在所有平台上工作。它通过寻找一组可用的DBM模块并选择最高级的可用库做到这一点（如果dbhash 已安装，通常是dbhash ）。如果不行，就使用dumbdbm 模块，这个模块一直可用。

另一个模块是whichdb ，它有一个函数whichdb(*filename*) ，用于研究一个文件是由哪种DBM数据库创建的。

一般来说，最好不要将这些底层模块用于可移植性很重要的应用程序。例如，如果你在一台机器上创建了一个DBM数据库，然后将数据库文件转移到另一台机器上，那么要是底层DBM数据库模块没有安装，Python有可能无法读取它。如果你正在使用数据库模块来存储大量的数据，可能出现多个Python程序同时打开同一个数据库文件的情况，或者出现需要高可靠性和事务处理能力的情况（sqlite3 模块可能是一个比较安全的选择），也需要特别注意这个问题。

17.4 shelve 模块

shelve 模块通过一个特殊的“架子”对象为持久性对象提供支持。这个对象的行为类似于字典，但是不同的是，它包含的所有对象都通过基于散列表的数据库（如dbhash、dbm 或gdbm ）存储在磁盘上。然而与那些模块不同，在架子上存储的值并不仅限于字符串。相反，任何能够与pickle 模块兼容的对象都可以被存储。使用shelve.open() 函数创建架子。

```
open(filename
 [,flag
='c' [, protocol
 [, writeback
]]])
```

打开一个架子文件。如果文件不存在，就创建它。 *filename* 应该是数据库文件名并且不应该包含后缀。 *flag* 含义与本章引言中的相同，是'r' 、'w' 、'c' 或'n' 。如果数据库文件不存在，就创建它。 *protocol* 指明了用于序列化存储在数据库中的对象的协议。它与在pickle 模块中的含义相同。 *writeback* 控制数据库对象的缓存行为。如果为True ，所有被访问的条目都缓存在内存中，而且只有当架子关闭之后才写回。默认值是False 。返回一个架子对象。

打开一个架子以后，可以对其执行下列字典操作：

操　　作	描　　述
d [key] = data	将数据存储在 *key* ，否则覆盖现有数据
data = *d* [*key*]	在 *key* 上检索数据
del *d* [*key*]	在 *key* 上删除数据
d .has_key(*key*)	测试 *key* 是否存在
d .keys()	返回所有关键字
d .close()	关闭架子

d .sync()	将未保存的数据写入磁盘

架子的key 值必须是字符串。存储在架子上的对象必须能够使用pickle 模块进行序列化。

```
Shelf(dict
 [, protocol
 [, writeback
]])
```

在字典对象 *dict* 上实现架子功能的混合类。当使用这个方法的时候，存储在返回架子对象上的对象将会被序列化并存储在底层字典 *dict* 上。 *protocol* 和 *writeback* 的含义都与shelve.open() 中的相同。

shelve 模块使用anydbm 模块来选择合适的DBM模块使用。在大多数标准Python安装中，它有可能是基于Berkeley库的dbhash 。

第18章　文件和目录处理

本章讲述高级文件和目录处理的Python模块。主题包括具有以下作用的模块：处理各种基本文件编码（如gzip和bzip2文件）、zip和tar等压缩文件解压、操作文件系统本身（如目录清单、移动、重命名、复制等）。与文件相关的底层操作系统调用将在第19章介绍。XML和HTML等文件内容的解析模块主要在第24章中介绍。

18.1　bz2

bz2 模块用于读取和写入根据bzip2压缩算法压缩的数据。

```
BZ2File(filename
 [, mode
[, buffering
 [, compresslevel
]]])
```

打开一个名为 *filename* 的.bz2 文件，返回一个类文件对象。当 *mode* 为'r' 时表示读取，为'w' 时表示写入，设置为'rU' 时，还可以获得通用换行支持。 *buffering* 规定了缓冲大小（字节数），默认值为0（无缓冲）。 *compresslevel* 为1至9之间的一个数字，值为9（默认）时压缩程度最大，但占用的压缩时间也最长。返回的对象支持全部常见文件操作，包括close() 、read() 、readline() 、readlines() 、seek() 、tell() 、write() 和writelines() 。

```
BZ2Compressor([compresslevel
])
```

创建用于顺序压缩数据块序列的压缩器对象。compresslevel 为指定压缩级别，用1至9的数字表示，默认值为9。

BZ2Compressor 的实例 *c* 有如下两种方法。

```
c
.compress(data
)
```

将一串新的数据添加到压缩器对象 *c* 中。如果可能，返回一串压缩的数据。因为压缩涉及多个数据块，所以返回的字符串可能没有涵盖全部数据，可能包含上次调用compress() 返回的数据。在输入所有数据之后，使用flush() 方法返回存储在压缩器中的所有剩余数据。

```
c
.flush()
```

刷新内部缓冲区并返回一串包含全部剩余数据的压缩字符串。执行该操作以后，不应再对此对象调用compress() 。

```
BZ2Decompressor()
```

创建一个解压缩器对象。

BZ2Decompressor 的实例 *d* 仅支持一种方法：

```
d
.decompress(data
)
```

假设在字符串 *data* 中有一个压缩的数据块，该方法可以返回解压的数据。因为数据是成块处理的，所以返回的字符串可能包含解压 *data* 中的全部信息，也可能不包含。重复调用这一方法，继续解压，直到输入中出现数据流结束标记为止。但如果在这之后试图继续执行解压数据操作，就会提示EOFError 异常。

```
compress(data
 [, compresslevel
])
```

返回字符串 *data* 中数据的一个压缩版。 *compresslevel* 是1至9之间的一个数字，9为默认值。

```
decompress(data
)
```

返回一串包含字符串 *data* 解压数据的字符串。

18.2 filecmp

filecmp 模块提供了以下函数，用于比较文件和目录。

```
cmp(file1
, file2
[, shallow
])
```

比较文件 *file1* 和 *file2* ，如果它们是相同的，则返回True ，否则返回False 。默认情况下，如果os.stat() 返回的文件属性相同，就认为文件相等。如果 *shallow* 参数为False ，将通过比较两个文件的内容来确定这两个文件是否相等。

```
```

```
cmpfiles(dir1
, dir2
, common
 [, shallow
])
```

比较 *common* 列表中位于 *dir1* 和 *dir2* 两个目录中的文件内容。返回一个包含三个文件名列表的元组（ *match* 、mismatch 、errors ）。 *match* 列出的是两个目录中相同的文件， *mismatch* 列出的是不同的文件， *errors* 列出的是由于某种原因不能进行比较的文件。 *shallow* 参数与cmp() 函数含义相同。

```
dircmp(dir1
, dir2
 [, ignore
[, hide
]])
```

创建一个目录比较对象，用于执行目录 *dir1* 与 *dir2* 的各种比较操作。 *ignore* 是要忽略的文件名列表，默认值为['RCS','CVS','tags'] 。 *hide* 是要隐藏的文件名列表，它默认的是[os.curdir, os.pardir] 列表（在UNIX中是['.', '..'] ）。

由dircmp() 返回的一个目录对象d ，具有下列方法和属性。

```
d
.report()
```

比较目录 *dir1* 和 *dir2* ，将报告输出到sys.stdout 。

```
d
.report_partial_closure()
```

比较 *dir1* 和 *dir2* 以及共同的直接子目录。结果输出到sys.stdout 。

```
d
.report_full_closure()
```

递归比较 *dir1* 和 *dir2* 以及所有子目录。结果输出到sys.stdout 。

```
d
.left_list
```

列出 *dir1* 中的文件和子目录。内容根据 *hide* 和 *ignore* 筛选。

```
d
.right_list
```

列出 *dir2* 中的文件和子目录。内容根据 *hide* 和 *ignore* 筛选。

```
d
.common
```

列出在 *dir1* 和 *dir2* 中都能找到的子目录。

```
d
.left_only
```

列出仅能在 *dir1* 中找到的文件和子目录。

```
d
.right_only
```

列出仅能在 *dir2* 中找到的文件和子目录。

```
d
.common_dirs
```

列出 *dir1* 和 *dir2* 共同的子目录。

```
d
.common_files
```

列出 *dir1* 和 *dir2* 共同的文件。

```
d
```

.common_funny

列出在 ***dir1*** 和 ***dir2*** 中类型不同的文件，或者是因为类型不同而无法通过 `os.stat()` 函数获得信息的文件。

d

.same_files

列出在 ***dir1*** 和 ***dir2*** 中内容相同的文件。

d

.diff_files

列出在 ***dir1*** 和 ***dir2*** 中内容不同的文件。

d

.funny_files

列出在 ***dir1*** 和 ***dir2*** 中都存在、但是由于某种原因不能进行比较（例如，访问权限不足）的文件。

d

.subdirs

将 *d.common_dirs* 中的文件名映射到其他dircmp 对象中的字典。

注意

dircmp 对象的属性是惰性求值的，在dircmp 对象首次创建时还未确定。因此，如果仅对其中某些属性感兴趣，则不会损失与其他未使用属性相关的性能。

18.3 fnmatch

fnmatch 模块支持使用UNIX shell型通配符提供文件名匹配。本模块只执行文件名匹配，而glob 模块则用于实际获取文件列表。fnmatch 模式的语法如下。

字　　符	描　　述
*	完全匹配
?	匹配任意单一字符
[*seq*]	匹配 *seq* 中的任意单一字符
[*!seq*]	匹配不在 *seq* 中的任意字符

以下函数用于测试通配符是否匹配：

```
fnmatch(filename, pattern
)
```

返回True 或False ，具体取决于 *filename* 与 *pattern* 是否匹配。是否区分大小写取决于操作系统（在Windows这样的特定平台上不区分大小写）。

```
fnmatchcase(filename, pattern
)
```

比较 *filename* 与 *pattern* （区分大小写）。

```
filter(names

, pattern

)
```

对于 *names* 序列中的所有名称应用fnmatch()函数，返回所有与 *pattern* 匹配的名称列表。

示例

```
fnmatch('foo.gif', '*.gif')                # 返回True
fnmatch('part37.html', 'part3[0-5].html') # 返回False

# 利用os.walk()、fnmatch和generators
# 在整个目录树种找到文件的示例
def findall(topdir, pattern):
    for path, files, dirs in os.walk(topdir):
        for name in files:
            if fnmatch.fnmatch(name,pattern):
                yield os.path.join(path,name)
# 找出所有.py文件
for pyfile in findall(".","*.py"):
    print pyfile
```

18.4 glob

glob 模块返回一个目录中与UNIX shell规则所指定的模式相匹配的全部文件名（如在fnmatch 模块中所述）。

```
glob

(pattern

)
```

返回与 *pattern* 相匹配的路径名列表。

```
iglob
(pattern
)
```

返回与glob() 相同的结果，但使用迭代器。

示例

```
htmlfile = glob('*.html')
imgfiles = glob('image[0-5]*.gif')
```

> 注意
>
> 不执行波形符（~）和shell变量扩展。在调用glob() 之前，分别使用os.path.expanduser() 和os.path.expandvars() 执行这些扩展。

18.5 gzip

gzip 模块提供了一个类，即Gzipfile 。它可以用来读取和写入与GNUgzip 程序兼容的文件。Gzipfile 对象与普通文件一样，只不过数据是自动压缩或解压缩的。

```
GzipFile([filename
 [, mode
 [, compresslevel
 [, fileobj
]]]])
```

打开GzipFile 。 *filename* 是文件的名称， *mode* 是'r' 、'rb' 、'a' 、'ab' 、'w' 、'wb' 之一。默认选项是'rb' 。 *compresslevel* 是1至9的整数，控制压缩级别。1速度最快，压缩程度最小；9速度最慢，压缩程度最高（默认）。 *fileobj* 是使用的现有文件对象。如果有，会使用它代替由 *filename* 命名的文件。

```
open(filename
 [, mode
```

```
 [, compresslevel

]])
```

与GzipFile(*filename,mode ,ompresslevel*) 相同。默认的 *mode* 是'rb' 。默认的 *compresslevel* 是9。

注意

- GzipFile 对象调用close() 方法时，并没有关闭在fileobj 中传递的文件，即允许在已经压缩的数据后面添加新的信息。
- UNIX compress 程序生成的文件不受支持。
- 这一模块需要zlib 模块。

18.6 shutil

shutil 模块用于执行高级文件操作，如复制、移动和重命名等。本模块的函数仅用于一般的文件和目录。特别需要注意的是，它不能处理命名管道、块设备等文件系统中的特殊文件类型。另外，注意这些函数并不总是能够正确执行高级的文件元数据处理（如资源派生、创建者守则等）。

```
copy(src, dst

)
```

将文件 *src* 复制到文件或目录 *dst* ，保留文件许可。 *src* 和 *dst* 是字符串。

```
copy2(src, dst

)
```

与copy() 类似，但同时复制了最后访问时间和修改时间。

```
copyfile(src, dst
```

```
)
```

将 *src* 的内容复制到 *dst* 。 *src* 和 *dst* 是字符串。

```
copyfileobj(f1, f2
 [, length
])
```

将打开的文件对象 *f1* 中的所有数据复制到打开的文件对象 *f2* 。length 指定可以使用的最大缓冲容量。如果 *length* 为负数，将在一次操作中复制全部数据（也就是说，全部数据将作为单一的数据块来读取并写入）。

```
copymode(src, dst
)
```

将许可位从 *src* 复制到 *dst* 。

```
copystat(src, dst
)
```

将许可位、最后访问时间、最后修改时间从 *src* 复制到 *dst* 。

内容、所有人、 *dst* 组将保持不变。

```
copytree(src, dst, symlinks
 [,ignore
]))
```

递归地复制 *src* 下的整个目录树。创建目标目录 *dst* （且不应该已存在）。使用copy2() 复制个别文件。如果 *symlinks* 为真，源目录树中的符号链接由新树中的符号链接所代表。如果 *symlinks* 为假或忽略，则将链接的文件的内容复制到新目录树。*ignore* 是一个可选函数，可用于筛选出特定的文件。作为输入，这个功能应该接受一个目录名称和目录内容列表。作为返回值，它应该返回一个将被忽略的文件名列表。如果在复制过程中发生错误，将收集错误，在处理结束时提示Error 异常。异常的参数是一份包含了所有错误的元组列表(*srcname* 、 *dstname* 、 *exception*)。

```
ignore_pattern(pattern1
, pattern2
, ...)
```

创建一个函数，用于忽略所有由 *pattern1* 、 *pattern2* 等给出的通配符样式（glob-style）模式。返回的函数接受两个参数的输入，一个是目录名称，另一个是目录内容列表。结果返回要忽略的文件名列表。返回的函数的主要用途是作为前述copytree() 函数的 *ignore* 参数。不过，这个结果函数也可用于与os.walk() 函数相关的操作。

```
move(src
, dst
)
```

将文件或目录从 *src* 移动到 *dst* 。如果 *src* 移动到了不同的文件系统中，将递归地复制 *src* 。

```
rmtree(path
 [, ignore_errors
[, onerror
]])
```

删除整个目录树。如果 *ignore_errors* 为真，错误将被忽略。否则，错误由onerror 函数（如果提供）处理。这个函数必须接受三个参数（ *func* 、 *path* 和 *excinfo* ）。其中 *func* 是导致错误的函数（os.remove() 或os.rmdir() ）， *path* 是传递到函数的路径名， *excinfo* 是由sys.exc_info() 返回的异常信息。如果错误发生而 *onerror* 被省略，那么系统将提示异常。

18.7 tarfile

tarfile 模块用于操作 *tar* 归档文件。利用这一模块，无论tar文件压缩与否，都可以读取和写入tar文件。

```
is_tarfile(name

)
```

如果name 为可以利用这一模块读取的有效tar文件，则返回True 。

```
open([name [, mode

[, fileobj

[, bufsize

]]]])
```

用路径名 *name* 创建一个新的TarFile 对象。 *mode* 是规定tar文件如何打开的字符串。 *mode* 字符串是文件模式和压缩方法（格式为'*filemode* [:*compression*]' ）的组合。有效的组合如下。

模　式	描　述
'r'	为读打开。如果文件是压缩的，将被透明地解压缩。这是默认模式
'r:'	为读打开，不压缩文件
'r:gz'	为读打开，使用gzip压缩文件

'r:bz2'	为读打开，使用bzip2压缩文件
'a','a:'	为继续添加文件而打开，不压缩文件
'w','w:'	为写打开，不压缩文件
'w:gz'	为写打开，使用gzip压缩文件
'w:bz2'	为写打开，使用bzip2压缩文件

创建一个仅允许序列I/O访问（没有随机要求）的TarFile 对象，使用下面的模式：

模　　式	描　　述
'r|'	为读打开一个解压块流
'r|gz'	为读打开一个gzip压缩流
'r|bz2'	为读打开一个bzip2压缩流
'w|'	为写打开一个解压流
'w|gz'	为写打开一个gzip压缩流
'w|bz2'	为写打开一个bzip2压缩流

如果指定了参数 *fileobj* ，它必须是一个打开的文件对象。在这种情况下，该文件覆盖了所有通过 *name* 规定的文件名， *bufsize* 规定了在tar文件中使用的数据块的大小。默认值是20*512字节。

由open() 返回的一个TarFile 的实例 *t* ，支持下列方法和属性：

```
t
.add(name
 [, arcname
```

```
[, recursive

]])
```

添加新文件到tar归档文件。 *name* 可以是任何类型的文件的名称（目录、符号链接等）。 *arcname* 是档案内部文件的备用名称。 *recursive* 是说明是否递归地添加目录内容的布尔标志（Boolean flag）。默认的设置为`True` 。

```
t

.addfile(tarinfo

 [, fileobj

])
```

添加新对象到tar归档文件。 *tarinfo* 是包含归档成员信息的`TarInfo` 结构。 *fileobj* 是一个打开的文件对象，其中的数据将会被读取并保存在归档文件中。读取的数据量取决于 *tarinfo* 属性的大小。

```
t

.close()
```

关闭tar归档文件，如果为进行写操作而打开归档，则向末尾写入两个零块。

```
t

.debug
```

控制生成调试信息的量，0无生成，3生成全部调试信息。信息写在`sys.stderr` 中。

```
t
.dereference
```

如果这一属性设置为True，则符号和硬链接都不能引用，引用的文件的全部内容都被添加到归档文件中。如果设置是False，只添加链接。

```
t
.errorlevel
```

当提取归档文件成员时，决定如何处理错误。如果这一属性设置为0，那么错误将被忽略。如果设置为1，错误将导致OSError 或IOError 异常。如果设置为2，那么非致命性错误还会导致TarError 异常。

```
t
.extract(member [, path
])
```

从归档文件中提取一个成员，保存到当前目录下，member 或是归档文件成员名称，或是TarInfo 实例。path 用于指定一个不同的目标目录。

```
t
.extractfile(member
)
```

从归档文件中提取成员，返回一个只读类文件对象，内容能够通过read()、readline()、readlines()、seek()和tell()操作读取。 *member* 可以是文档成员的名称，也可以是一个TarInfo对象。如果 *member* 引用了链接，将打开链接的目标。

```
t
.getmember(name
)
```

查找归档成员 *name* ，返回包含相关信息的TarInfo对象。如果没有这样的归档文件成员存在，将提示KeyError。如果成员 *name* 在归档文件中不止一次出现，将返回其最后一项的信息（假定它比较新）。

```
t
.getmembers()
```

返回所有归档文件成员的TarInfo对象列表。

```
t
.getnames()
```

返回所有归档文件成员的名称列表。

```
t
.gettarinfo([name
[, arcname
[, fileobj
```

```
]]])
```

在文件系统或打开的文件对象 *fileobj* 上返回与文件 *name* 对应的TarInfo 对象。 *arcname* 是归档文件中对象的备用名称。这一函数的主要用途是创建合适的TarInfo 对象，用于在add() 等方法中使用。

```
t
.ignore_zeros
```

如果这一属性设置为True ， 读取归档文件时将会跳过空块。如果这一设置为False （默认），空块表示归档文件结束。将这一方法设置为True ， 有助于读取损坏的归档文件。

```
t
.list([verbose
])
```

将文档的内容列出到sys.stdout 。 *verbose* 决定信息的详细程度。如果这一方法设置为False ， 则仅打印归档文件名。否则将打印全部详细信息（默认）。

```
t
.next()
```

用于迭代归档文件成员的方法。返回下一个归档文件成员的TarInfo 结构或None 。

```
t

.posix
```

如果这一属性设置为True，将按照POSIX 1003.1-1990标准创建tar文件。此标准对文件名长度和文件大小作出限制（文件名必须少于256字符，文件必须小于8GB）。如果这一属性设置为False，则使用GNU扩展版创建归档文件，没有这些限制。默认值是False。

前面的许多方法可以操作TarInfo 实例。下表显示TarInfo 实例ti 的方法和属性。

属　　性	描　　述
ti .gid	组ID
ti .gname	组名称
ti .isblk()	如果对象是块设备，则返回True
ti .ischr()	如果对象是字符设备，则返回True
ti .isdev()	如果对象是设备（字符、数据块或者FIFO），则返回True
ti .isdir()	如果对象是目录，则返回True
ti .isfifo()	如果对象是FIFO，则返回True
ti .isfile()	如果对象是普通文件，则返回True
ti .islnk()	如果对象是硬链接，则返回True
ti .isreg()	与isfile() 相同
ti .issym()	如果对象是符号链接，则返回True
ti .linkname	硬链接或符号链接的目标文件名
ti .mode	许可位

ti .mtime	最后修改时间
ti .name	归档成员名称
ti .size	大小（字节）
ti .type	常量REGTYPE 、 AREGTYPE 、 LNKTYPE 、 SYMTYPE 、 DIRTYPE 、 FIFOTYPE 、 CONTTYPE 、 CHRTYPE 、 BLKTYPE 或GNUTYPE_SPARSE 中的文件类型
ti .uid	用户ID
ti .uname	用户名

18.7.1 异常

下列异常由tarfile 模块定义：

TarError

所有其他异常的基类。

ReadError

打开tar文件出错时提示（例如，当打开一个无效文件时）。

CompressionError

数据不能被解压时提示。

StreamError

在类似流的TarFile 对象上执行不受支持的操作时提示（例如，要求随机访问的操作）。

ExtractError

在提取过程中发生非致命性错误时提示（仅当errorlevel 设置为2时）。

18.7.2 示例

```
# 打开一个tar文件，向其中放入一些文件
t = tarfile.open("foo.tar","w")
t.add("README")
import glob
for pyfile in glob.glob("*.py"):
    t.add(pyfile)
t.close()

# 打开tar文件，迭代其所有成员
t = tarfile.open("foo.tar")
for f in t:
     print("%s %d" % (f.name, f.size))

# 扫描tar文件，打印“README”文件的内容
t = tarfile.open("foo.tar")
for f in t:
     if os.path.basename(f.name) == "README":
          data = t.extractfile(f).read()
          print("**** %s ****" % f.name)
```

18.8 tempfile

tempfile 模块用于生成临时性文件名和文件。

mkdtemp([*suffix*

[,*prefix*

[, *dir*

]]])

创建只有调用过程的拥有者能够进入的临时性目录，返回其绝对路径。 *suffix* 是可选后缀，可以附加到目录名称末尾。 *prefix* 是可选前缀，可以插入到目录名称开头， *dir* 是创建临时文件的目录。

```
mkstemp([suffix
[,prefix
[, dir
 [,text
]]]])
```

创建临时文件，返回元组（fd，pathname ），其中 *fd* 是由os.open() 返回的整数文件描述符， *pathname* 是文件的绝对路径。 *suffix* 是可选后缀，可以附加到文件名末尾， *prefix* 是可选前缀，可以放在文件名开头， *dir* 是创建文件的目录， *text* 是布尔标志，规定文件的打开方式是文本模式或者是二进制模式（默认）。如果系统支持os.open() 的O_EXCL 标志，那么文件的创建保证是原子式（atomic ）的（并且是安全的）。

```
mktemp([suffix
 [, prefix
 [,dir
]]])
```

返回唯一的临时文件名。 *suffix* 是可选后缀，可以附加到文件名末尾， *prefix* 是一个可选前缀，可以插入到文件名开头， *dir* 是创建文件的目录。这个函数只生成一个唯一的文件名，并不实际创建或打开临时文件。因为这个函数在文件实际打开之前生成名称，可引发潜在的安全问题。为了解决这个问题，可考虑改用mkstemp() 。

```
gettempdir()
```

返回创建临时文件的目录。

```
gettempprefix()
```

返回用于生成临时文件的前缀。不包含文件将要保存到的目录。

```
TemporaryFile([mode
 [, bufsize
 [, suffix
 [,prefix
[, dir
]]]]])
```

使用msktemp()创建临时文件，然后返回类文件对象，该对象支持与普通文件对象相同的方法。 *mode* 是文件模式，默认为'w+b'。 *bufsize* 规定了缓冲行为，与open()函数同义。 *suffix* 、 *prefix* 和 *dir* 与mkstemp()函数同义。该函数返回的对象只是内置文件对象（可在文件属性中找到）的包装器。当临时文件对象被销毁时，该函数创建的文件也会被自动销毁。

```
NamedTemporaryFile([mode
[, bufsize
 [, suffix
 [,prefix
 [, dir
[, delete
 ]]]]]])
```

创建与TemporaryFile()相似的临时文件，但保证文件名在文件系统中可见。访问返回的文件对象的 *name* 属性，就能得到这个文件名。注意，在临时文件关闭之前，某

些系统可能会阻止文件以这个名字重新打开。如果 *delete* 参数设置为True （默认），那么在临时文件关闭之后，就会删除它。

```
SpooledTemporaryFile([max
                      
                      
                      
                      
_size
[, mode
 [, bufsize
 [, suffix
 [, prefix
[,dir
]]]]]])
```

创建临时文件，如TemporaryFile ，让文件的内容全部保留在内存中，直到超过 *max_size* 中给出的大小。这个内部后台操作的实现方式如下：先将文件内容保存到StringIO 对象中，直到实际需要进入文件系统为止。正如TemproryFile 对象定义的一样，只要执行包含fileno() 方法的低级别文件I/O操作，内存的内容就会被立即写入临时文件中。由SpooledTemproryFile 返回的文件对象还有一个方法rollover() ，可用于强制将内容写入文件系统上。

下面两个全局变量用于构建临时性的名称。如果需要，可以赋予它们新值。默认值取决于系统。

变　　量	描　　述
tempdir	由mktemp() 返回的文件名所在的目录
template	由mktemp() 生成的文件名前缀。添加一串十进制数字到template ，生成唯一的文件名

注意

默认情况下，tempfile 模块通过检查几个标准的位置创建文件。例如，在UNIX中，文件在/tmp、/var/tmp或/usr/tmp中的某个位置创建。在Windows中，文件在C:\TEMP、C:\TMP、\TEMP或\TMP中的某个位置创建。通过设置一个或更多TMPDIR、TEMP和TMP环境变量可以覆盖这些目录。如果出于某种原因，临时文件不能在常见位置创建，那么它们将在当前工作目录中创建。

18.9 zipfile

zipfile 模块用于操作流行的zip格式编码的文件（起初称为PKZIP，不过现在受许多程序支持），zip文件在Python中广为使用，主要用于程序打包。例如，如果把包含Python源代码的zip文件添加到sys.path，那么包含在zip文件内的文件就能够通过import 载入（zipimport 库模块实现了这项功能，但是没必要直接使用这个库）。以.egg 文件（由安装工具扩展创建）分发的程序包实际上也是伪装的zip文件（.egg 文件实际上是zip文件，其中添加了一些新的元数据）。

下面的函数和类都是由zipfile 模块定义的：

```
is_zipfile(filename
)
```

测试 *filename* ，查看它是否是一个有效的zip文件。如果 *filename* 是zip文件，则返回True，否则返回False。

```
ZipFile(filename
[, mode
 [, compression
 [,allowZip64
]]])
```

打开zip文件 *filename* ，返回ZipFile 实例。当mode 为'r' 时，读取现有文件内容，为'w' 时，写入新文件并删除原文件，为'a' 时，向现有文件添加文件。在'a' 模式下，如果 *filename* 为现有zip文件，那么新文件就会添加到其中。如果 *filename* 不是zip文件，那么归档文件就仅附加到文件末尾。 *compression* 是在写入归档文件时使用的zip 压缩方法，它或者是ZIP_STORED，或者是ZIP_DEFLATED。默认是ZIP_STORED。 *allowZip64* 参数支持ZIP64扩展，后者能够用于创建大小超过2GB的zip文件。默认情况下，这项设置为False。

```
PyZipFile(filename
 [, mode
[, compression
```

```
 [,allowZip64

]]])
```

与ZipFile() 类似，打开一个zip文件，但是返回一个特殊的PyZipFile 实例，其包括一个额外的方法writepy() 将Python源文件添加到归档文件中。

```
ZipInfo([filename

[, date

_time

]])
```

手动创建新的ZipInfo 实例，用于包含归档文件成员的信息。一般情况下，没有必要调用这个函数，除了在使用ZipFile 实例（后面将介绍）的z.writestr() 方法时。*filename* 和 *date_time* 参数提供下列关于 *filename* 和 *date_time* 属性的值。

ZipFile 或PyZipFile 的实例 *z* ，支持下面的方法和属性。

```
z

.close()
```

关闭归档文件。调用这一函数是为了在程序终止之前刷新zip文件的记录。

```
z

.debug
```

调试级别，范围从0（没有输出）到3（最大输出）。

```
z
.extract(name
[, path
[, pwd
]])
```

从归档文件中提取一个文件，把它放在当前工作目录中。 *name* 可以是说明归档文件的成员，也可以是ZipInfo 实例的字符串。 *Path* 指定了提取文件的另一个目录，其中 *pwd* 是密码，供归档文件加密使用。

```
z
.extractall([path
[members
 [, pwd
]]])
```

将所有归档文件成员提取到当前工作目录。 *path* 指定了另一个目录， *pwd* 是密码，供归档文件加密使用。 *members* 是要提取的成员列表，这些成员必须是使用namelist() 方法（下面将介绍）返回的列表的正确子集。

```
z
.getinfo(name)
```

返回归档文件成员名的信息，作为ZipInfo 实例（稍后介绍）。

```
z
.infolist()
```

返回归档文件所有成员的ZipInfo 对象列表。

```
z
.namelist()
```

返回归档文件成员名称列表。

```
z.open(name
 [, mode
 [, pwd
]])
```

打开名为 *name* 的文档成员，返回一个类文件对象，用来读取内容。 *name* 可以是字符串，也可以是描述归档文件成员之一的ZipInfo 实例。mode 是文件模式，必须是'r' 、'rU' 、'U' 等几种只读文件模式之一。 *pwd* 是供加密归档文件成员使用的密码。返回的文件对象支持read() 、readline() 和readlines() 方法，也支持for 语句的迭代。

```
z
.printdir()
```

将归档文件目录输出到sys.stdout 。

```
z
.read(name [,pwd
])
```

读取归档文件成员 *name* 的内容，返回的数据是一个字符串。 *name* 或者是字符串，或者是描述归档文件成员的ZipInfo 实例。 *pwd* 是供加密归档文件成员使用的密码。

```
z
.setpassword(pwd
)
```

设置默认的密码，用于从归档文件提取加密文件。

```
z
.testzip()
```

读取归档文档中的所有文件，并验证它们的CRC校验和。返回第一个损坏文件的名称，如果所有文件都是完整的，则返回None 。

```
z
.write(filename
[, arcname
[, compress_type
]])
```

将文件 *filename* 写入到归档名为 *arcname* 的归档文件中， *compress_type* 是压缩参数，要么是ZIP_STORED，要么是ZIP_DEFLATED。在默认情况下，使用ZipFile() 或PyZipFile() 函数中的压缩参数。归档文件必须以'w' 或'a' 模式打开，以便能够写入。

```
z
.writepy(pathname
)
```

这个方法仅对PyZipFile 实例可用，此方法将Python源文件（*.py 文件）写入到zip归档文件中，而且能够轻松打包Python应用程序以便进行分发。如果 *pathname* 是文件，那么它必须以.py 结尾。在这种情况下，将会添加对应的.pyo 、.pyc 或.py 文件（按照这个顺序）。如果 *pathname* 是目录，并且这个目录不是Python包目录，那么所有对应的.pyo 、.pyc 或.py 文件都添加至包的顶层。如果目录是一个包，那么添加的文件将以包名称作为文件路径。如果子目录也是包目录，那么它们会被递归地添加。

```
z
.writestr(arcinfo
, s
)
```

将字符串s 写入zip文件中。 *arcinfo* 要么是存储该数据的归档文件内部的文件名，要么是ZipInfo 实例，包含文件名、日期和时间。

ZipInfo 实例i 由ZipInfo() 、z.getinfo() 和z.infolist() 函数返回，具有如下属性。

属　　性	描　　述
i .filename	归档成员名

i .date_time	元组（year 、 month 、 day 、 hours,minutes,seconds ）包含最后修改时间， month 和day 分别是1～12和1～31的数字。所有其他值都从0开始
i .compress_type	归档文件成员的压缩类型。当前仅ZIP_STORED 和ZIP_DEFLATED 受这种模式支持
i .comment	归档文件成员评论
i .extra	扩展字段数据，用于包含新的文件属性。这里存储的数据取决于创建文件的系统
i .create_system	描述创建归档文件系统的整数编码。常见的值包括0（MS-DOS FAT）、3（UNIX）、7（Macintosh）和10（Windows NTFS）
i .create_version	创建zip归档文件的PKZIP版编码
i .extract_version	提取归档文件所需的最低版本
i .reserved	保留字段。当前的设置是0
i .flag_bits	描述包含加密和压缩的数据编码的zip标志位
i .volume	文件头的卷号
i .internal_attr	描述文档内容的内部结构。如果最低位是1，那么数据是ASCII文本，否则假定为二进制数据。
i .external_attr	依赖操作系统的外部文件属性
i .header_offset	朝向文件头的字节偏移
i .file_offset	朝向文件数据开头的字节偏移
i .CRC	非压缩文件的CRC校验和
i .compress_size	压缩文件数据大小
i .file_size	非压缩文件的大小

注意

关于zip文件的内部结构的详细文档，请参阅http://www.pkware.com/appnote.html 上的PKZIP应用程序说明。

18.10 zlib

zlib 模块通过提供对zlib 库的访问支持数据压缩。

```
adler32(string
 [, value
])
```

计算Adler-32字符串校验和。 *value* 用作初始值（用来通过串联字符串来计算校验和）。否则，使用固定的默认值。

```
compress(string
 [, level
])
```

将数据压缩到 *string* 中，压缩级别是从1到9的整数，控制压缩程度。1表示压缩程度最小（最快），9代表压缩程度最大（最慢）。默认值为6。返回包含压缩数据的字符串，发生错误会弹出错误提示。

```
compressobj([level
])
```

返回压缩对象。 *level* 与 *compress()* 的含义相同。

```
crc32(string [, value
])
```

计算 *string* 的CRC校验和。如果值存在，则用它作为校验和的初始值，否则使用固定值。

```
decompress(string
 [, wbits
 [, buffsize
]])
```

将数据解压到 *string* 。 *wbits* 控制窗口缓冲区的大小， *buffsize* 是输出缓冲区的起始大小。如果发生错误则弹出提示。

```
decompressobj([wbits])
```

返回压缩对象。 *wbits* 参数控制窗口缓冲区的大小。

压缩对象 *c* ，有如下的方法：

```
c
.compress(string
)
```

压缩 *string* 。至少针对 *string* 中的部分数据返回包含压缩数据的字符串。这些数据应该连接到先前调用 *c* .compress() 函数时生成的输出以创建输出流。一些输入数据可以存储在内部缓冲区中待后续处理。

```
c
```

```
.flush([mode
])
```

压缩所有待定输入，返回包含剩余压缩输出的字符串。 *mode* 是Z_SYNC_FLUSH、Z_FULL_FLUSH 或Z_FINISH （默认）。Z_SYNC_FLUSH 和Z_FULL_FLUSH 可以进行进一步压缩，用于在解压缩时进行部分错误恢复。Z_FINISH 终止压缩流。

解压缩对象 *d* 有如下的方法和属性：

```
d
.decompress(string
 [,max_length
])
```

解压 *string* ，至少针对 *string* 中的部分数据返回包含解压数据的字符串。该数据应该连接到先前调用函数decompress() 时生成的数据来生成输出流。一些输入数据可以储存在内部缓冲区中，待后续处理。 *max_length* 指定最大的返回数据量。如果超出这一数量，未处理的数据就会被放在 *d.unconsumed_tail* 属性中。

```
d
.flush()
```

处理所有待定输入，返回包含剩余解压输出的字符串。调用这个函数以后，解压对象不能再次使用。

```
d
.unconsumed_tail
```

包含未经上一次decompress()函数处理的字符串中的数据。如果因为缓冲区大小限制而需要按阶段进行解压的话，那么它将包含数据。在本例中，这个变量将会传递给后续decompress()函数。

```
d
.unused_data
```

包含压缩数据末尾额外字节的字符串。

> **注意**
>
> zlib库可在http://www.zlib.net 获得。

第19章　操作系统服务

本章介绍的模块与访问各种操作系统服务有关，主要涉及底层I/O、进程管理和操作环境。此外，本章还会介绍与编写系统程序有关的模式，如用于读取配置文件的模块、写入日志文件的模块等。第18章介绍了与文件和文件系统操作有关的高级模块，本章介绍的模块比第18章介绍的更加底层。

Python的大多数操作系统模块都是以POSIX接口为基础的。POSIX是一个标准，它定义了一组核心的操作系统接口。大多数UNIX系统都支持POSIX，而包括Windows在内的其他平台也支持此接口的大部分模块。本章，我们将注明仅适用于特定平台的函数和模块。UNIX系统包括Linux和Mac OS X。除非另有说明，Windows系统包括Windows的所有版本。

读者可能需要阅读更多参考资料，以此补充在本章中学习到的知识。Brian W. Kernighan和Dennis M. Ritchie撰写的*The C Programming Language, Second Edition* ① （Prentice Hall，1989）概述了文件、文件描述符和本章所述的大多数模块所基于的底层接口。更高水平的读者可以阅读W. Richard Stevens和Stephen Rago编著的*Advcanced Programming in the Unix Environment* , Second Edition ② （Addison Wesley，2005）。关于一般概念的概述，可以参考关于操作系统的大学课本。但这些书籍的价格不菲，而且在日常工作中的实用功能性有限，更经济的方法是请身边计算机科学系的学生在周末把他们的课本借给你。

19.1　Commands

`Commands` 模块用于执行以字符串形式指定的简单系统命令，并将其输出以字符串形式返回。此模块仅在UNIX系统上有效。这个模块提供的功能与在UNIX shell脚本中使用反引号（` ）有几分相似。例如，键入`x = commands.getoutput('ls -l')` 类似于声明`x='ls -l'` 。

getoutput(*cmd*)

在Shell中执行 *cmd* ，返回包含命令标准输出和标准错误流的字符串。

getstatusoutput(*cmd*)

与getoutput() 相似，差别在于返回的是二元组(*status* , *output*)，其中的 *status* 是退出代码，由os.wait() 函数返回， *output* 是getoutput() 返回的字符串。

注意

- 此模块仅在Python 2中可用。在Python 3中，上述函数可在subprocess 模块中找到。
- 尽管此模块可用于简单的shell操作，但最好使用subprocess 模块启动子进程并收集其输出。

另请参见： 19.14节。

19.2 ConfigParser、configparser

ConfigParser 模块（在Python 3中称为configparser ）用于读取基于Windows INI格式的.ini 格式配置文件。这些文件由命名段（named section）组成，每个命名段都有自己的变量赋值，如下所示：

```
# 注释
; A comment
[section1]
name1 = value1


name2 = value2



[section2]
; Alternative syntax for assigning values
name1: value1


name2: value2


...
```

19.2.1 ConfigParser 类

下面这个类用于管理配置变量：

```
ConfigParser([defaults

 [, dict_type

]])
```

创建新的ConfigParser 实例。 *defaults* 是可选的值字典，可通过包含字符串格式说明符在配置变量中引用，如'%(key)s'，其中 *key* 是 *defaults* 的键，dict_type 指定在内部用于存储配置变量的字典的类型。默认情况下是dict （内置字典）。

ConfigParser 的实例c 包含以下操作。

```
c
.add_section(section
)
```

为存储的配置参数添加一个新段。 *section* 是带有段名称的字符串。

```
c
.defaults()
```

返回默认值字典。

```
c.
get(section
, option
 [, raw
[, vars
]])
```

将 *section* 段中的 *option* 选项的值以字符串形式返回。默认情况下，所返回的字符串经过插值处理，扩展'%(option)s' 等格式字符串。在本例中， *option* 可能是相同段中的另一个配置选项的名称，也可能是ConfigParser 的 *defaults* 参数中提供的默认值之一。 *raw* 是一个布尔标记，用于禁用此插值功能并返回未被修改的选项。*vars* 是可选字典，包含可在'%' 扩展中使用的更多值。

```
c
.getboolean(section
, option
)
```

从 *section* 段返回 *option* 的值，并将其转换为布尔值。"0" 、"true" 、"yes" 、"no" 、"on" 和"off" 等值均可被理解，而且不区分大小写，此方法始终会执行变量插值（参见c.get() ）。

```
c
.getfloat(section
, option
)
```

从 *section* 段返回 *option* 的值，并通过变量插值将其转换为浮点值。

```
c
.getint(section
, option
)
```

从 *section* 段返回 *option* 的值，并通过变量插值将其转换为整型值。

```
c
.has_option(section
, option
)
```

若 *section* 段有名为 *option* 的选项，则返回True 。

```
c
.has_section(section
)
```

若存在名为 *section* 的段，则返回True 。

```
c
.items(section
 [, raw
[, vars
]])
```

从 *section* 段返回(*option* , *value*)对。 *raw* 是一个布尔标记，若设置为True ， 则禁用插值功能。 *vars* 是可在'%' 扩展中使用的额外值的字典。

```
c
```

```
.options(section
)
```

返回 *section* 段中所有选项的列表。

```
c
.optionxform(option
)
```

将选项名称 *option* 转换为字符串，用于引用选项。默认转换为小写字母形式。

```
c
.read(filenames
)
```

从文件名列表读取并存储配置选项。 *filenames* 可以是一个字符串（表示所读取的是文件名），也可以是文件名列表。如果无法找到给定文件名，则被忽略。如果需要从多个位置读取配置文件，但此类文件可能已定义，也可能未定义，那么这种方法比较有用。返回成功解析的文件名列表。

```
c
.readfp(fp
 [, filename
])
```

从 *fp* 中已打开的类文件对象中读取配置选项。 *filename* 指定与 *fp* 关联的文件名（若存在）。默认情况下，文件名是从 *fp.name* 中获取的，若未定义此类属性，则将其设为'<???>'。

```
c
.remove_option(section
, option
)
```

从 *section* 段中删除 *option* 。

```
c
.remove_section(section
)
```

删除 *section* 段。

```
c
.sections()
```

返回所有段名称的列表。

```
c
.set(section
, option
, value
```

```
)
```

将配置选项 *option* 设为段 *section* 中的 *value* 。 *value* 应为字符串。

```
c
.write(file
)
```

将当前持有的所有配置数据写入 *file* 。 *file* 是一个已经打开的类文件对象。

19.2.2 示例

ConfigParser 模块往往被忽视，但它是一个极其有用的工具，它可以控制包含极度复杂的用户配置或运行时环境的程序。例如，若编写了必须在大型框架内运行的组件，那么配置文件往往是提供运行时参数的理想方式。类似地，使用配置文件也比使用optparse 模块读取大量命令行参数的程序合理。在使用配置文件和简单地从Python源脚本读取配置数据之间存在着一些细微而重要的差别。

下面几个示例展示了ConfigParser 模块一些较为有趣的功能。首先，请考虑示例.ini 文件：

```
# appconfig.ini
# mondo应用程序的配置文件

[output]
LOGFILE=%(LOGDIR)s/app.log
LOGGING=on
LOGDIR=%(BASEDIR)s/logs

[input]
INFILE=%(INDIR)s/initial.dat
INDIR=%(BASEDIR)s/input
```

以下代码说明如何读取配置文件并给一些变量提供默认值：

```
from configparser import ConfigParser # Python 2中应使用from ConfigParser

# 默认变量设置的字典
```

```
defaults = {
       'basedir' : '/Users/beazley/app'
}

# 创建ConfigParser对象并读取.ini文件
cfg = ConfigParser(defaults)
cfg.read('appconfig.ini')
```

读取配置文件后，使用get() 方法获取选项值。例如：

```
>>> cfg.get('output','logfile')


'/Users/beazley/app/logs/app.log'
>>> cfg.get('input','infile')


'/Users/beazley/app/input/initial.dat'
>>> cfg.getboolean('output','logging')


True
>>>
```

这里有一些有趣的特性。首先，配置参数不区分大小写。因此，如果程序要读取参数'logfile' ，那么无论配置文件使用'logfile' 、'LOGFILE' 还是'LogFile' 都可以被读取。其次，配置参数可以包括变量替换，如文件中的'%(BASEDIR)' 和'%(LOGDIR)' 。这些替换也不区分大小写。此外，配置参数的定义顺序在这些替换中无关紧要。例如，在appconfig.ini 中，LOGFILE 参数引用了LOGDIR 参数，而后者在文件中是稍后定义的。最后，配置文件中的值一般都能得到正确解释，即使它们不完全匹配Python语法或数据类型。例如，LOGGING 参数的'on' 值被cfg.getboolean() 方法解释为True 。

可以将配置文件合并在一起。例如，假定用户自己的配置文件具有以下自定义设置：

```
; userconfig.ini
;
; Per-user settings

[output]
logging=off

[input]
BASEDIR=/tmp
```

可以把这个文件的内容合并到已加载的配置参数中，例如：

```
```

```
>>> cfg.read('userconfig.ini')


['userconfig.ini']
>>> cfg.get('output','logfile')


'/Users/beazley/app/logs/app.log'
>>> cfg.get('output','logging')


'off'
>>> cfg.get('input','infile')


'/tmp/input/initial.dat'
>>>
```

在这里可以注意到，新加载的配置有选择地替换了已经定义好的参数。此外，如果修改其他配置参数的变量代替使用中的配置参数，改动也会正确地替换。例如，input 部分中BASEDIR 的新设置影响到了这部分中前面定义的配置参数，如INFILE 。这种行为是使用配置文件与在Python脚本中定义一组程序参数之间的一个重大且微妙的差异。

19.2.3 注意

可以使用两个其他的类来代替ConfigParser 。类RawConfigParser 类提供了ConfigParser 的所有功能，但不执行任何变量插值。SafeConfigParser 类提供的功能与ConfigParser 相同，但它解决了一些微妙问题。例如，当配置值本身在字面上包含插值功能使用的特殊格式化字符（如'%' ）时出现的问题。

19.3 datetime

datetime 模块提供表示和处理日期和时间的一些类。此模块的大部分功能是关于创建和输出日期与时间信息的各种不同方式。其他主要的功能包括数学运算，如时间增量的比较和计算。日期处理是一个复杂的主题，我们强烈建议读者参考Python的在线文档，从中了解关于此模块设计的背景资料。

19.3.1 date 对象

date 对象代表由年、月、日组成的简单日期。以下4个函数用于创建日期。

```
date(year, month, day

)
```

此函数创建一个新的日期对象。year 是范围在datetime.MINYEAR 和 datetime.MAXYEAR 之间的整数。month 是一个范围在1到12之间的整数，而day 是一个范围在1到指定月份天数之间的整数。返回的date 对象是不可变的，它的属性year 、month 和day 分别对应于所提供参数的值。

```
date.today()
```

此类方法返回对应当前日期的date 对象。

```
date.fromtimestamp(timestamp)
```

此类方法返回对应时间戳timestamp 的date 对象。timestamp 是time.time() 函数的返回值。

```
date.fromordinal(ordinal)
```

此类方法返回对应从允许的最小日期开始的ordinal 天的date 对象（例如，1年1月1日的序数值为1，而2006年1月1日的序数值为732312）。

以下类属性描述了date 实例的最大边界和解析程度。

```
date.min
```

此类属性表示能够表示的最早日期（datetime.date(1, 1, 1)）。

```
date.max
```

此类属性表示可能的最晚日期（datetime.date(9999, 12, 31)）。

```
date.resolution
```

不相等的日期对象之间最小可解析的差值（datetime.timedelta(1)）。

date 实例 *d* 具有只读属性 *d* .year、 *d* .month 和 *d* .day，另外还提供了以下方法。

```
d
.ctime()
```

返回一个字符串，表示的日期格式与time.ctime() 函数一般使用的格式相同。

```
d
.isocalendar()
```

以元组(*iso_year* , *iso_week* , *iso_weekday*)的形式返回日期，这里的 *iso_week* 范围在1到53之间， *iso_weekday* 范围在1（星期一）到7（星期日）之间。第一个 *iso_week* 是当年包含第一个星期四的一周。这三个元组组件的值范围由ISO 8601标准决定。

```
d
.isoformat()
```

返回表示日期的ISO 8601格式字符串，形式为“ *YYYY-MM-DD* ”。

```
d
.isoweekday()
```

返回一周的一天，范围在1（星期一）到7（星期日）之间。

```
d
.replace([year
[, month
[, day
]]])
```

返回一个新的date 对象，并使用新值替代其中一个或多个提供的参数。例如，d.replace(month=4) 返回一个新的日期，并将其中的月份值替换为4。

```
d
.strftime(format
)
```

返回一个字符串，表示日期的格式与time.strftime() 函数相同。此函数只能用于1900年以后的日期。此外，禁止对date 对象中没有的参数（如小时、分等）使用格式代码。

```
d
.timetuple()
```

返回适合函数在time 模块中使用的time.struct_time 对象。与一天中时间（小时，分，秒）相关的值将被设为0。

```
d
.toordinal()
```

将 *d* 转换为一个序数值。例如，1年1月1日的序数值为1。

```
d
.weekday()
```

返回一周内的时间，范围在0（星期一）到6（星期日）之间。

19.3.2 time 对象

time 对象用于表示包含小时、分、秒和微秒的时间。使用以下类构造函数创建time 对象：

```
time(hour
 [, minute
 [, second
[, microsecond
 [, tzinfo
]]]])
```

在使用此方法创建的time 对象中，各个部分的范围分别是：0 <= *hour* < 24，0 <= *minute* < 60，0 <= second < 60，0 <= microsecond < 1000000。 *tzinfo*

提供时区信息，是本节稍后要讲到的 *tzinfo* 类的一个实例。返回的time 对象具有属性hour 、minute 、second 、microsecond 和tzinfo ，分别保存对应于所提供参数的值。

time 的以下类属性描述了time 实例的允许值和解析程度的范围：

```
time
.min
```

此类属性表示可表示的最小时间（datetime.time(0,0) ）。

```
time
.max
```

此类属性表示可表示的最大时间（datetime.time(23, 59, 59, 999999) ）。

```
time.resolution
```

不等的time 对象之间最小可解析的差值（datetime.timedelta(0, 0, 1) ）。

time 对象的实例 *t* 具有属性 *t* .hour 、 *t* .minute 、 *t* .second 、 *t* .microsecond 和 *t* .tzinfo ，并提供以下方法：

```
t
.dst()
```

返回 *t* .tzinfo.dst(None) 的值。返回的对象是一个timedelta 对象。如果未设置时区，则返回None 。

```
t
.isoformat()
```

返回一个字符串，表示时间的格式为"HH:MM:SS.mmmmmm" 。如果微秒为0，就省略字符串的微秒部分。如果提供了时区信息，时间上可能会添加一个偏移量（如"HH:MM:SS.mmmmmm+HH:MM" ）。

```
t
.replace([hour
 [, minute
 [, second
 [, microsecond
 [, tzinfo
]]]]])
```

返回一个新的time 对象，并使用提供的值替代一个或多个部分。例如， *t* .replace(second=30) 将秒字段修改为30 ，并返回一个新的time 对象。参数的含义与前面提供给time() 函数的参数相同。

```
t
.strftime(format
)
```

返回一个字符串，格式化规则与time 模块中的time.strftime() 函数相同。因为日期信息不可用，只能对时间信息使用格式化代码。

```
t
.tzname()
```

返回 *t* .tzinfo.tzname() 的值。如果未设置时区，则返回None 。

```
t
.utcoffset()
```

返回 *t* .tzinfo.utcoffset(None) 的值。返回对象是一个timedelta 对象。如果未设置时区，则返回None 。

19.3.3 datetime 对象

datetime 对象用于表示日期和时间。创建datetime 实例有多种途径：

```
datetime(year, month, day
 [, hour [, minute
 [, second
[, microsecond
 [,tzinfo
]]]]])
```

此函数用于创建一个新的datetime 对象，它同时包含date 和time 对象的所有特性。其中参数的意义与提供给date() 和time() 的参数相同。

```
datetime.combine(date
,time)
```

此类方法通过组合 *date* 对象 *date* 和time 对象 *time* 的内容，创建出一个datetime 对象。

```
datetime.fromordinal(ordinal
)
```

此类方法创建指定序数天数（自datetime.min 以来的整数天数）的datetime 对象。时间组件均被置为0，而tzinfo 被置为None 。

```
datetime.fromtimestamp(timestamp
 [, tz
])
```

此类方法基于time.time() 函数返回的时间戳创建一个datetime 对象。 *tz* 提供可选时区信息，是tzinfo 的一个实例。

```
datetime.now([tz
])
```

此类方法基于当前的本地日期和时间创建一个datetime 对象。 *tz* 提供可选时区信息，是tzinfod 的一个实例。

```
datetime.strptime(datestring, format
```

```
)
```

此类方法根据format 中的日期格式解析datestring 中的日期字符串，从而创建一个datetime 对象。解析使用time 模块中的strptime() 函数执行。

```
datetime.utcfromtimestamp(timestamp
)
```

此类方法基于通常由time.gmtime() 返回的时间戳来创建一个datetime 对象。

```
datetime.utcnow()
```

此类方法基于当前的UTC日期和时间创建datetime 对象。

以下类属性描述了所允许的日期和解析度的范围。

```
datetime.min
```

可以表示的最早的日期和时间（datetime.datetime(1, 1, 1, 0, 0)）。

```
datetime.max
```

可以表示的最晚的日期和时间（datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)）。

```
datetime.resolution
```

不等datetime对象之间的最小可解析差值（`datetime.timedelta(0, 0, 1)`）。

datetime对象的实例 *d* 具有的方法由`date` 和`time` 对象的方法组合而成：

```
d
.astimezone(tz
)
```

返回一个位于不同时区 *tz* 中的新datetime对象。新对象的成员将被调整为相同的UTC时间，但位于时区tz中。

```
d
.date()
```

返回一个具有相同日期的`date` 对象。

```
d
.replace([year
 [, month
 [, day
[, hour
 [, minute
 [, second
 [, microsecond
 [, tzinfo
```

```
]]]]]]]]
```

返回一个新的datetime 对象，并将列出参数中的一个或多个替换为新值。使用关键字参数替换单个值。

```
d
.time()
```

使用相同时间返回time 对象。得到的time 对象没有时区信息集合。

```
d
.timetz()
```

使用相同的时间和时区信息返回time 对象。

```
d
.utctimetuple()
```

返回time.struct_time 对象，其中包括规格化为UTC时间的日期和时间信息。

19.3.4 timedelta 对象

timedelta 对象表示两个日期或时间之间的差值。这些对象通常是使用—运算符计算两个datetime 实例之间的差值时的结果。不过，使用下面的类可以手动构造它们：

```
timedelta([days [, seconds
```

```
 [, microseconds

 [, milliseconds

 [, minutes

 [, hours

[,


weeks

]]]]]]])
```

此类方法创建表示两个日期和时间之间差值的timedelta 对象。唯一有意义的参数是days 、seconds 和microseconds ， 它们用于在内部表示差值。如果提供其他参数，它们将转换为天、秒和微秒。返回的timedelta 对象的属性days 、seconds 和 microseconds 包含这些值。

以下类属性描述了timedelta 实例的最大范围和解析度。

```
timedelta.min
```

可以表示的最大负数timedelta （timedelta(-999999999) ）。

```
timedelta.max
```

可以表示的最大正数timedelta （timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999) ）

```
timedelta.resolution
```

表示不等timedelta 对象之间的最小可解析差值的timedelta 对象

（timedelta(microseconds=1)）。

19.3.5 涉及日期的数学运算

datetime 模块的一个重要特性是：它支持涉及日期的数学运算。date 和datetime 对象都支持以下操作。

操　作	描　述	操　作	描　述
td = date1 - date2	返回timedelta 对象	date1 == date2	
date2 = date1 + td	给date 添加timedelta	date1 != date2	
date2 = date1 – td	从date 减去timedelta	date1 > date2	
date1 < date2	日期比较	date1 >= date2	
date1 <= date2			

比较日期时，如果提供了时区信息，必须十分当心。如果日期包括tzinfo 信息，那么该日期只能与包含tzinfo 信息的其他日期进行比较，否则就会生成TypeError 。当比较两个位于不同时区的日期时，比较之前会先把它们调整为UTC时间。

timedelta 对象还支持各种数学运算。

操　作	描　述	操　作	描　述
td3 = td2 + td1	将两个时间增量相加	abs(td)	绝对值
td3 = td2 - td1	将两个时间增量想减	td1 < td2	比较
td2 = td1 * i	乘以整数	td1 <= td2	
td2 = i * td2		td1 == td2	
td2 = td1 // i	对整数 i 进行地板除（floor division）	td1 != td2	
td2 = -td1	一元减法，加法	td1 > td2	
td2 = +td1		td1 >= td2	

下面是一些例子：

```
>>> today = datetime.datetime.now()

>>> today.ctime()

'Thu Oct 20 11:10:10 2005'
>>> oneday = datetime.timedelta(days=1)

>>> tomorrow = today + oneday

>>> tomorrow.ctime()

'Fri Oct 21 11:10:10 2005'
>>>
```

除了这些操作之外，所有date 、datetime 、time 和timedelta 对象都是不可变的。也就是说，它们可以用作字典关键字，放在集合中，还能用在各种其他操作中。

19.3.6 tzinfo 对象

datetime 模块中的很多方法可以操作表示时区信息的特殊tzinfo 对象。tzinfo 仅仅是一个基类。通过继承tzinfo 类和实现以下方法创建一个时区。

```
tz

.dst(dt)
```

返回一个timedelta 对象，此对象代表夏令时（Daylight Saving Time, DST）调整（如果有的话）。如果没有关于夏令时的信息，则返回None 。参数 *dt* 是datetime 对象或None 。

```
tz

.fromutc(dt
```

```
)
```

将datetime 对象 *dt* 从UTC时间转换为本地时区，并返回一个新的datetime 对象。此方法由datetime 对象上的astimezone() 方法调用。tzinfo 已经提供了默认实现，因此一般没有必要重新定义此方法。

```
tz
.tzname(dt
)
```

返回代表时区名称的字符串，如"US/Central" 。 *dt* 是一个datetime 对象或None 。

```
tz
.utcoffset(dt
)
```

返回timedelta 对象，表示本地时间与UTC东区时间的偏移，单位是分钟。偏移结合了构成本地时间的所有元素，包括夏令时（如果有的话）。参数 *dt* 是一个datetime 对象或None 。

下面的例子显示了如何定义一个时区的基本原型：

```
# 必须要定义的变量
# TZOFFSET – 与UTC的时区偏移，单位是小时。例如，US/CST是-6小时
# DSTNAME  – DST有效时的时区名称
# STDNAME  – DST无效时的时区名称

class SomeZone(datetime.tzinfo):
        def utcoffset(self,dt):
                return datetime.timedelta(hours=TZOFFSET) + self.dst(dt)
        def dst(self,dt):
                                # is_dst()是必须实现的函数
                # 无论本地时区规则中DST是否有效。
```

```
            if is_dst(dt):
                    return datetime.timedelta(hours=1)
            else:
                    return datetime.timedelta(0)


    def tzname(self,dt):

            if is_dst(dt):

                    return DSTNAME

            else:

                    return STDNAME
```

在datetime 的在线文档中还可以找到定义时区的大量例子。

19.3.7 日期与时间解析

日期处理的一个常见问题是如何将不同种类的时间与日期字符串解析为正确的datetime 对象。datetime 模块提供的唯一解析函数是datetime.strptime() 。然而，为了使用此函数，需要使用格式代码的各种组合来指定精确的日期格式（参见time.strptime() ）。例如，要解析日期字符串s="Aug 23, 2008" ，必须使用d = datetime.datetime.strptime(s, "%b %d, %Y") 。

对于自动理解大量常见日期格式的“模糊”日期解析，必须求助于第三方模块。访问Python Package Index（http://pypi.python.org ）并搜索"datetime" ，可以找到各种实用工具模块来扩展datetime 模块的功能集合。

另请参见： 有关time 对象的内容。

19.4 errno

errno 模块为各种操作系统调用返回的整数错误代码定义了符号名称，特别是os 和socket 模块中的那些代码。这些代码通常可在OSError 或IOError 异常的errno 属性中找到。os.strerror() 函数可用于将错误代码转换为字符串错误消息。下面的字典还可以用于将整数错误代码转换为符号名称：

```
errorcode
```

此字典将errno 整数映射为符号名称，如'EPERM' 。

19.4.1 POSIX错误代码

下面列出了常见系统错误代码的POSIX符号名称。几乎每个版本的UNIX、Macintosh OS-X和Windows都支持这里列出的错误代码。不同的UNIX系统可能另外提供不那么常见的、这里没有列出的错误代码。如果出现这类错误，可以参考errorcode 字典，以便在程序中使用正确的符号名称。

错误代码	描　　述	错误代码	描　　述
E2BIG	参数列表过长	ENETUNREACH	网络不可到达
EACCES	访问被拒绝	ENFILE	文件表溢出
EADDRINUSE	地址已经使用	ENOBUFS	无可用缓存空间
EADDRNOTAVAIL	无法分配请求的地址	ENODEV	无此类设备
EAFNOSUPPORT	协议不支持地址族	ENOENT	文件或目录不存在
EAGAIN	再试	ENOEXEC	可执行文件格式错误
EALREADY	操作已经在进行中	ENOLCK	无可用记录锁定
EBADF	错误的文件编号	ENOMEM	内存不足
EBUSY	设备或资源繁忙	ENOPROTOOPT	协议不可用
ECHILD	无子进程	ENOSPC	设备上无剩余空间
ECONNABORTED	软件导致连接中断	ENOSYS	函数无法实现
ECONNREFUSED	连接被拒绝	ENOTCONN	传输端点未连接
ECONNRESET	对等端已将连接重置	ENOTDIR	不是一个目录
EDEADLK	将出现资源死锁	ENOTEMPTY	目录不为空

EDEADLOCK	文件锁定死锁错误	ENOTSOCK	非套接字上的套接字操作
EDESTADDRREQ	需要目的地址	ENOTTY	不是一个终端
EDOM	数学参数在函数作用域之外	ENXIO	设备或地址不存在
EDQUOT	超出配额	EOPNOTSUPP	传输端点上不支持操作
EEXIST	文件存在	EPERM	操作未得到许可
EFAULT	错误的地址	EPFNOSUPPORT	不支持协议族
EFBIG	文件过大	EPIPE	管道已损坏
EHOSTDOWN	主机已关闭	EPROTONOSUPPORT	不支持协议
EHOSTUNREACH	无路由通向主机	EPROTOTYPE	套接字的协议类型错误
EILSEQ	非法的字节序列	ERANGE	无法表示的数学结果
EINPROGRESS	操作正在进行	EREMOTE	对象是远程的
EINTR	系统调用被中断	EROFS	只读文件系统
EINVAL	无效参数	ESHUTDOWN	无法在传输端点关闭后发送
EIO	I/O错误	ESOCKTNOSUPPORT	套接字类型不受支持
EISCONN	传输端点已经连接	ESPIPE	非法寻址
EISDIR	是一个目录	ESRCH	进程不存在
ELOOP	遇到过多的符号链接	ESTALE	失效的NFS文件句柄
EMFILE	打开文件过多	ETIMEDOUT	连接超时
EMLINK	链接过多	ETOOMANYREFS	引用过多：无法连接

EMSGSIZE	消息过长	EUSERS	用户过多
ENETDOWN	网络已关闭	EWOULDBLOCK	操作将阻塞
ENETRESET	网络由于重置中断连接	EXDEV	跨设备链接

19.4.2 Windows错误代码

下表中的错误代码只在Windows上可用。

错误代码	描　　述	错误代码	描　　述
WSAEACCES	访问被拒绝	WSAENETRESET	网络由于重置中断连接
WSAEADDRINUSE	地址已被使用	WSAENETUNREACH	网络不可达
WSAEADDRNOTAVAIL	无法分配请求的地址	WSAENOBUFS	无可用缓存空间
WSAEAFNOSUPPORT	协议不支持地址族	WSAENOPROTOOPT	错误的协议选项
WSAEALREADY	操作正在进行	WSAENOTCONN	套接字未连接
WSAEBADF	无效的文件句柄	WSAENOTEMPTY	无法删除非空目录
WSAECONNABORTED	软件导致连接中断	WSAENOTSOCK	非套接字上的套接字操作
WSAECONNREFUSED	连接被拒绝	WSAEOPNOTSUPP	操作不支持
WSAECONNRESET	对等端已将连接重置	WSAEPFNOSUPPORT	协议族不受支持
WSAEDESTADDRREQ	需要目的地址	WSAEPROCLIM	进程过多
WSAEDISCON	远程关闭	WSAEPROTONOSUPPORT	协议不支持
WSAEDQUOT	超出磁盘配额	WSAEPROTOTYPE	套接字的协议类型错误
WSAEFAULT	错误的地址	WSAEREMOTE	本地内容不可用

WSAEHOSTDOWN	主机关闭	WSAESHUTDOWN	无法在套接字关闭后发送
WSAEHOSTUNREACH	无路由通向主机	WSAESOCKTNOSUPPORT	套接字类型不支持
WSAEINPROGRESS	操作正在进行	WSAESTALE	文件句柄不再可用
WSAEINTR	系统调用被中断	WSAETIMEDOUT	连接超时
WSAEINVAL	无效参数	WSAETOOMANYREFS	对一个内核对象的引用过多
WSAEISCONN	套接字已连接	WSAEUSERS	超出配额
WSAELOOP	无法转换名称	WSAEWOULDBLOCK	资源暂时不可用
WSAEMFILE	打开文件过多	WSANOTINITIALISED	启动WSA没有成功
WSAEMSGSIZE	消息过长	WSASYSNOTREADY	网络子系统不可用
WSAENAMETOOLONG	名称过程	WSAVERNOTSUPPORTED	Winsock.dll版本超出范围
WSAENETDOWN	网络已关闭		

19.5 fcntl

fcntl 模块对UNIX文件描述符执行文件和I/O控制。使用文件或套接字对象的 fileno() 方法可以获得文件描述符。

```
fcntl(fd
, cmd
 [, arg
])
```

在打开的文件描述符 *fd* 上执行命令 *cmd* 。 *cmd* 是整数命令代码。 *arg* 是可选参数，是整数或字符串。如果 *arg* 传递时是整数，那么此函数的返回值就是整数。如果 *arg* 是字符串，那么它将被解释为二进制数据结构，而调用的返回值就是被转换回字符

串对象的缓存内容。在这种情况下，所提供的参数和返回值应该小于1 024字节，从而避免潜在的数据损坏。可用的命令如下。

命　令	描　述
F_DUPFD	复制一个文件描述符。 *arg* 是新文件描述符能够假定的最小数字，类似于os.dup() 系统调用
F_SETFD	将close-on-exec 标志置为 *arg* （0或1）。如果设置，在exec() 系统调用时会关闭文件
F_GETFD	返回close-on-exec 标志
F_SETFL	将状态标志置为arg，它是以下内容的按位OR： O_NDELAY ——非阻塞I/O（System V） O_APPEND ——挂起模式（System V） O_SYNC ——同步写入（System V） FNDELAY ——非阻塞I/O（BSD） FAPPEND ——挂起模式（BSD） FASYNC ——当I/O可用时发送SIGIO 信号给进程组（BSD）
F_GETFL	获得由F_SETFL 设定的状态标志
F_GETOWN	获得设定为接收SIGIO 和SIGURG 信号的进程ID或进程组ID（BSD）
F_SETOWN	设置进程ID或进程组ID以接收SIGIO 和SIGURG 信号（BSD）
F_GETLK	返回文件锁定操作中使用的群结构
F_SETLK	锁定文件，如果文件已经被锁定，则返回-1
F_SETLKW	锁定文件，但如果无法获得锁定则会等待

如果fcntl() 函数失败，就会引发IOError 异常。F_GETLK 和F_SETLK 命令是由lockf() 函数支持的。

```
ioctl(fd
, op
, arg
[, mutate_flag
])
```

此函数很像fcntl() 函数，区别是 *op* 中提供的操作一般定义在库模块termios中。当传递可变的缓存对象作为参数时，额外的 *mutate_flag* 将控制此函数的行为。关于这个主题的详细内容可以参考在线文档。因为ioctl() 函数的首要用途是与设备驱动程序和操作系统的其他底层组件进行交互，所以它的使用高度依赖于底层平台。在需要可移植的代码中不应使用此函数。

```
flock(fd
, op
)
```

在文件描述符 *fd* 上执行锁定操作。op 是以下内容的按位或，这些内容可以在fnctl 中找到。

项　　目	描　　述
LOCK_EX	排他锁。获得锁定的所有进一步尝试都将被阻塞，除非锁定被释放
LOCK_NB	非阻塞模式。如果锁定已经在使用，就会立即返回IOError
LOCK_SH	共享锁。阻塞任何获取排它锁（LOCK_EX ）的尝试，但仍然可以获取共享锁
LOCK_UN	解锁。释放所有以前加上的锁

在非阻塞模式中，如果无法获取锁定，就会引发一个IOError 异常。在某些系统上，打开和锁定文件的过程可以通过一种操作来完成，即给os.open() 操作添加特殊标志。更多细节请参考os 模块。

```
lockf(fd
, op
 [, len
 [, start
```

```
 [, whence

]]])
```

对部分文件执行记录或范围锁定。 *op* 同flock()函数。 *len* 是要锁定的字节数。 *start* 是相对于 *whence* 值的锁定起始位置。 *whence* 是0代表文件开始，1代表当前位置，而2代表文件结尾。

19.5.1 示例

```
import fcntl

# 打开一个文件
f = open("foo","w")

# 为文件对象f设置close-on-exec位
fcntl.fcntl(f.fileno(), fcntl.F_SETFD, 1)

# 锁定文件(阻塞)
fcntl.flock(f.fileno(), fcntl.LOCK_EX)

# 锁定文件的前8192个字节(非阻塞)
try:
    fcntl.lockf(f.fileno(), fcntl.LOCK_EX | fcntl.LOCK_NB, 8192, 0, 0)
except IOError,e:
    print "Unable to acquire lock", e
```

19.5.2 注意

- 可用的fcntl()命令和选项集合取决于系统。在某些平台上，Fcntl模块包含的常量可能超过100个。
- 尽管定义在其他模块中锁定操作通常使用上下文管理器协议，但文件锁定并不是这样。如果要获取文件锁定，确保所写的代码中正确释放了锁定。
- 此模块中的很多函数还可以用于套接字的文件描述符。

19.6 io

io模块实现了各种形式的I/O类，以及Python 3中使用的内置open()函数。此模块在Python 2.6中也可用。

io模块所解决的中心问题是无缝地处理不同形式的基本I/O。例如，由于与换行和字符编码相关的问题，处理文本与处理二进制数据略有不同。为了处理这些差异，此模块由一系列代码层组成，其中每层依次增加更多功能。

19.6.1 基本I/O接口

io 模块定义了所有类文件对象都会实现的基本I/O编程接口。此接口由基类IOBase定义。IOBase 的实例 *f* 支持如下基本操作。

属　　性	描　　述
f .closed	指示文件是否关闭的标志
f .close()	关闭文件
f .fileno()	返回整数形式的文件描述符
f .flush()	清空I/O缓存（如果有的话）
f .isatty()	如果 *f* 是终端则返回True
f .readable()	如果 *f* 已打开可供读取，则返回True
f .readline([*limit*])	从流中读取一行。 *limit* 是要读取字节的最大数量
f .readlines([*limit*])	读取 *f* 的所有行，然后以列表的形式返回。如果提供 *limit* 参数，它表示在停止之前能够读取字节的最大数量。实际读取的字节数将稍大以容纳保持原样的最后一行
f .seek(*offset* , [*whence*])	移动文件指针到相对于whence 中指定位置的新位置。 *offset* 代表字节数。 *whence* 是0代表文件开始位置，1代表当前位置，2代表文件结尾
f .seekable()	如果f 是可拖动的则返回True
f .tell()	返回文件指针的当前值
f .truncate([*size*])	将文件大小截取为 *size* 指定大小的字节。如果未指定 *size* 参数，它将把文件截取为0个字
f .writable()	如果 *f* 已打开并可写入，则返回True
f .writelines(*lines*)	将一系列行写入 *f* 。不额外添加行尾，因为它们已经是每行的组成部分

19.6.2　原始I/O

I/O系统的最低层次与涉及原始字节的直接I/O相关。这个层次的核心对象是FileIO，它提供了通向低级别系统调用（如read()和write()）的直接接口。

```
FileIO(name
 [, mode
[, closefd
]])
```

此类用于在文件或系统文件描述符上执行原始的低级别I/O操作。name是由os.open()函数或其他文件对象的fileno()方法返回的文件名或整数文件描述符。*mode*可取的值包括'r'（默认值）、'w'或'a'，分别代表读取、写入或附加。mode中还可加入'+'代表更新，同时支持读取和写入。*closefd*是一个标志，用于确定close()方法是否关闭了底层文件，默认情况下，它的值为True，但如果使用FileIO给已经在其他地方打开的文件加上包装，那么它的值将被置为False。如果指定了文件名，就会直接调用操作系统的open()打开作为结果得到的文件对象。不存在内部缓存，而且所有数据都将作为原始字节字符串进行处理。FileIO的实例*f*具有前面描述过的所有基本I/O操作，再加上以下属性和方法。

属　　性	描　　述
f.closefd	一个标志，用于确定底层文件描述符是否由*f*.close()方法（只读）关闭
f.mode	打开时使用的文件模式（只读）
f.name	文件名（只读）
f.read([*size*])	使用一次系统调用读取的最大字节数。如果省略*size*参数，就会使用*f*.readall()方法返回尽可能多的数据。这种操作返回的字节数可能比请求的要少，因此必须使用len()函数进行检查。如果非锁定模式中不存在可用数据，则返回None
f.readall()	读取尽可能多的可用数据，并以一个字节字符串的形式返回。遇到EOF时返回一个空字符串。在非锁定模式中，只会立即返回尽可能多的数据
f.write(*bytes*)	使用一次系统调用写入一个字节字符串或字节数组。返回实际写入的字节数——这可能小于所提供的字节数

有一点需要着重强调，即FileIO对象是特别底层的对象，它基于read()和write()

之类的操作系统调用提供了一个十分薄的层。特别地，此对象的用户需要经常检查返回代码，因为无法保证 *f* .read() 或 *f* .write() 操作是否会读取或写入所有请求的数据。fcntl 模块可用于改变文件的底层操作，如文件锁定、阻塞行为等。

禁止将FileIO 对象用于面向行的数据，如文本。尽管定义了像 *f* .readline() 和 *f* .readlines() 这样的方法，但这些方法来自IOBase 基类，都完全使用Python实现，而且工作方式是一次对一个字节进行 *f* .read() 操作。例如，与在由Python 2.6中的 open() 函数创建的标准文件对象上使用 *f* .readline() 方法相比，在FileIO 对象 *f* 上使用 *f* .readline() 方法的速度要慢上750倍。

19.6.3 缓存二进制I/O

缓存I/O层包含一些用于读取和写入原始二进制数据的文件对象，但采用的是内存中缓存机制。作为输入，这些对象都需要文件对象来实现原始I/O，如前面提过的FileIO 对象。本节中的所有类均继承自BufferedIOBase 类。

```
BufferedReader(raw
 [, buffer_size
])
```

此类让缓存二进制读取raw 中指定的原始文件。 *buffer_size* 用于指定要使用的缓存大小，单位为字节。如果省略此参数，就会使用DEFAULT_BUFFER_SIZE 的值（到本书撰写之际为8 192个字节）。BufferedReader 的实例 *f* 不仅支持IOBase 上提供的所有操作，而且还支持以下操作。

```
BufferedWriter(raw
 [, buffer_size
 [, max_buffer_size
]])
```

方　法	描　述
f .peek([*n*])	在不移动文件指针的情况下，从I/O缓存返回最多 *n* 个字节的数据。如果省略 *n* ，就会返回单个字节。如果有必要，当缓存目前为空时，就会执行读取操作来填充缓存。这项操作返回的字节数绝不会大于当前的缓存大小，因此结果可能小于所请求的字节数 *n*

f .read([*n*])	读取 *n* 个字节并以字节字符串的形式返回。如果省略 *n* ，就会读取并返回所有的可用数据（到EOF为止）。如果底层文件是非阻塞的，就会读取并返回任意可用数据。如果读取非阻塞文件时无可用数据，就会引发BlockingIOError 异常
f .read1([*n*])	最多读取 *n* 个字节，并使用单次系统调用以字节字符串的形式返回。如果缓存中已经加载了数据，就会返回这些数据。否则，就会对原始文件进行read() 操作以返回数据。和 *f* .read() 不同，即使底层文件没到EOF处，这种操作返回的数据也可能少于所请求的数量
f .readinto(*b*)	从文件读取len(*b*) 个字节到现有bytearray 对象 *b* 中。返回值是所读取的实际字节数。当底层文件处于非阻塞模式，如果无可用数据时将引发一个BlockingIOError 异常

此类让缓存二进制写入raw 中指定的一个原始文件。 *buffer_size* 用于指定在数据被写入到底层I/O流中之前，缓存中能够保存的字节数。默认值是DEFAULT_BUFFER_SIZE 。 *max_buffer_size* 指定用于保存要写入到非阻塞流中的输出数据的最大缓存大小，默认为 *buffer_size* 值的两倍。这个值对于持续写入来说足够大，因为操作系统会不断将以前的缓存内容写入到I/O流中。BufferedWriter 的实例支持以下操作。

方　　法	描　　述
f .flush()	将缓存中保存的所有字节写入底层I/O流。当文件处于非阻塞模式时，如果操作阻塞（即如果流当时无法接受任何新数据），则会引发BlockingIOError异常
f .write(*bytes*)	将bytes 中的字节写入I/O流，然后返回实际写入的字节数。当底层流是非阻塞时，如果写操作阻塞，则引发BlockingIOError异常

```
BufferedRWPair(reader, writer
 [, buffer_size
[, max_buffer_size
]])
```

此类用于在一对原始I/O流上进行缓存二进制式的读取和写入。reader 是支持读取的原始文件，而writer 是支持写入的原始文件。这些文件可能有所差别，这对于涉及管道和套接字的某些类型的通信可能有用。 *buffer_size* 参数同BufferedWriter 。BufferedRWPair 的实例 *f* 支持BufferedReader 和BufferedWriter 的所有操作。

```
BufferedRandom(raw
 [, buffer_size
[, max_buffer_size
]])
```

此类用于在支持随机访问（如寻址）的原始I/O流上进行缓存二进制式的读取和写入。raw 必须是同时支持读取、写入和寻址操作的原始文件。 *buffer_size* 参数同BufferedWriter 。BufferedRandom 的实例 *f* 支持BufferedReader 和BufferedWriter 的所有操作。

```
BytesIO([bytes
])
```

实现缓存I/O流功能的内存中文件。 *byte* 是指定文件初始内容的字节字符串。BytesIO 的实例 *b* 支持BufferedReader 和BufferedWriter 对象的所有操作。另外，方法 *b* .getvalue() 可用于以字节字符串的形式返回文件的当前内容。

和FileIO 对象一样，本节中的所有文件对象都禁止用于面向行的数据，如文本。尽管由于缓存机制的存在，结果并不算很坏，但性能仍然很糟（比使用Python 2.6内置函数open() 读取文件行的速度慢上50倍）。此外，因为内部缓存的原因，写入时需要注意管理flush() 操作。例如，如果使用 *f* .seek() 将文件指针移动到一个新位置，应该首先使用 *f* .flush() 清除掉以前写入的数据（如果有的话）。

此外，要清楚缓存大小参数只能指定写入发生时的缓存大小，而不能对内部资源的使用进行限制。例如，对一个缓存文件 *f* 进行 *f* .write(*data*) 操作时， *data* 中的所有字节将被首先复制到内部缓存中。如果 *data* 表示一个非常大的字节数组，这种复制将显著增加程序的内存使用。因此，最好将数量较大的数据分为多个大小合适的数据块，而不是通过write() 操作一次性写入。应该注意，因为io 模块相对较新，这种行为在以后的版本中可能会出现变化。

19.6.4 文本I/O

文本I/O层用于处理面向行的字符数据。本节中定义的类基于缓存I/O流构建，并且增加了面向行处理和Unicode字符编码。这里所有的类均继承自TextIOBase 类。

```
TextIOWrapper(buffered
```

```
[, encoding
 [, errors
[, newline
 [, line_buffering
]]]])
```

此类用于缓存文本流。 *buffered* 是前面提到过的缓存I/O。 *encoding* 是一个字符串，如指定文本编码的'ascii'或'utf-8'。 *errors* 指定Unicode错误处理策略，默认为'strict'（请参阅第9章中的描述）。 *newline* 是表示换行的字符序列，可为None、''、'\n'、'\r'或'\r\n'。如果指定为None，就会启用通用换行模式，在这种模式中，当读取并在输出时使用os.linesep作为换行时，将把任意其他的行结束转换为'\n'。如果 *newline* 是其他值，那么输出时所有的'\n'字符都将被转换为指定的换行。 *line_buffering* 是一个标志，用于控制当任意写入操作包含换行字符时，是否执行flush()操作。它的默认值是False。TextIOWrapper的实例 *f* 支持IOBase类上定义的所有操作，以及以下操作。

方　法	描　述
f .encoding	所使用的文本编码名称
f .errors	编码与解码错误处理策略
f .line_buffering	确定行缓存行为的标志
f .newlines	None、字符串或者元组，用于给出转换后各种形式的换行
f .read([*n*])	从底层流读取最多 *n* 个字符，并以字符串的形式返回。如果省略参数 *n*，将读取所有可用数据直到文件结尾。在EOF处返回空字符串。返回的字符串将根据 *f* .encoding 中的编码设置进行编码
f .readline([*limit*])	读取文本的一行，并以字符串的形式返回。在EOF处返回空字符串。 *limit* 是要读取的最大字节数
f.write(*s*)	使用 *f* .encoding 中的文本编码将字符串 *s* 写入底层流

```
StringIO([initial
```

```
 [, encoding
 [, errors
[, newline
]]]])
```

内存中的文件对象，行为与TextIOWrapper 相同。 *initial* 是一个字符串，用于指定文件的初始内容。其他参数的意义同TextIOWrapper 。StringIO 的实例s 支持所有常用的文件操作，以及返回内存缓冲区当前内容的 *s* .getvalue() 方法。

19.6.5 open() 函数

io 模块定义了下面的open() 函数，它与Python 3中内置的open() 函数相同。

```
open(file
 [, mode
 [, buffering
 [, encoding
 [, errors
 [, newline
 [, closefd
]]]]]])
```

此函数用于打开 *file* 并返回相应的I/O对象。 *file* 是指定文件名称的字符串，或者是已经打开的I/O流的整数文件描述符。根据 *mode* 和 *buffering* 的设置，此函数的结果是io 模块中定义的I/O类之一。如果 *mode* 是文本模式之一，如'r' 、'w' 、'a' 或'U' ，那么返回TextIOWrapper 的一个实例。如果 *mode* 是二进制模式，如'rb' 和'wb' ，那么返回的结果取决于 *buffering* 的设置。如果 *buffering* 是0，就会返回FileIO 的实例，用于执行原始的未缓存I/O。如果 *buffering* 是其他值，将根据文件模式分别返回BufferReader 、BufferedWriter 或BufferedRandom 的实例。 *encoding* 、 *errors* 和 *newline* 参数只适用于以文本模式打开的文件，它们将被传递给TextIOWrapper 构造函数。只有当 *file* 是整数描述符时才能使用 *closefd* 参数，它将被传递给FileIO 构造函数。

19.6.6 抽象基类

io 模块定义了以下抽象基类，可用于进行类型检查和定义新的I/O类。

抽 象 类	描 述
IOBase	所有I/O类的基类
RawIOBase	支持原始二进制I/O的对象的基类。继承自IOBase 类
BufferedIOBase	支持缓存二进制I/O的对象的基类。继承自IOBase 类
TextIOBase	支持文本流的对象的基类。继承自IOBase 类

大多数程序员极少会直接使用这些类。关于它们用法和定义的细节请参考在线文档。

> **注意**
>
> io 模块是新添加到Python中的，它首先出现在Python 3中，然后向后移植到了Python 2.6中。在撰写本书时，该模块还不成熟，运行时性能非常糟糕，特别是对涉及大量文本I/O的应用程序。如果使用Python 2，使用内置函数open()比使用io 模块中定义的I/O类的效果更好。如果使用Python 3，似乎没有别的更好的选择。尽管以后的版本中可能会对性能进行改善，但这种对与Unicode编码耦合在一起的I/O进行分层的方法，在性能上不可能达到C标准库（是Python 2中I/O的基础）中的原始I/O性能。

19.7 logging

logging 模块为应用程序提供了灵活的手段来记录事件、错误、警告和调试信息。对这些信息可以进行收集、筛选、写入文件、发送给系统日志等操作，甚至还可以通过网络发送给远程计算机。本节讲述了使用此模块的大部分常见案例的详细情况。

19.7.1 日志记录级别

logging 模块的重点在于生成和处理日志消息。每条消息由一些文本和指示其严重性的相关级别组成。级别包含符号名称和数字值，如下所示。

级 别	值	描 述
CRITICAL	50	关键错误/消息
ERROR	40	错误
WARNING	30	警告消息

INFO	20	通知消息
DEBUG	10	调试
NOTSET	0	无级别

这些不同的级别是整个logging 模块中的各个函数和方法的基础。例如，每个级别上都有用于发出日志消息的方法，以及用于阻塞不满足某个阈值的消息的筛选器。

19.7.2 基本配置

在使用logging 模块中的任意其他函数之前，应该首先执行特殊对象的一些基本配置，这个特殊对象称为根记录器（root logger）。根记录器负责管理日志消息的默认行为，包括日志记录级别、输出目标位置、消息格式以及其他基本细节。下面的函数用于配置：

```
basicConfig([**kwargs
])
```

此函数用于执行根记录器的基本配置。此函数应该在进行其他日志记录调用之前进行调用。此函数接受以下关键字参数。

关键字参数	描　　述
filename	将日志消息附加到指定文件名的文件
filemode	指定用于打开文件的模式。默认使用模式'a'（附加）
format	用于生成日志消息的格式字符串
datefmt	用于输出日期和时间的格式字符串
level	设定根记录器的级别。级别等于或大于此级别的所有日志消息都将被处理。低于此级别的消息将被静默忽略

stream	提供打开的文件，用于把日志消息发送到其中。默认的流是sys.stderr 。此参数不能与filename 参数同时使用

这些参数中大多数参数的含义是不言而喻的。 *format* 参数用于指定日志消息，以及可选上下文消息（如文件名、级别、行号等）的格式。 *datefmt* 是与time.strftime() 函数兼容的日期格式字符串。如果省略此参数，日期格式将设置为ISO8601格式。

以下扩展在 *format* 中是可识别的。

格　　式	描　　述
%(name)s	记录器的名称
%(levelno)s	数字形式的日志记录级别
%(levelname)s	日志记录级别的文本名称
%(pathname)s	执行日志记录调用的源文件的路径名称
%(filename)s	执行日志记录调用的源文件的文件名
%(funcName)s	执行日志记录调用的函数名称
%(module)s	执行日志记录调用的模块名称
%(lineno)d	执行日志记录调用的行号
%(created)f	执行日志记录时的时间。它的值是time.time() 返回的数字
%(asctime)s	执行日志记录调用时的ASCII格式的日期和时间
%(msecs)s	执行日志记录调用时的时间的毫秒部分
%(thread)d	线程ID
%(threadName)s	线程名称
%(process)d	进程ID

%(message)s	记录的消息（由用户提供）

下面这个例子将级别为INFO 或更高的日志消息附加给一个文件：

```
import logging
logging.basicConfig(
    filename = "app.log",
    format   = "%(levelname)-10s %(asctime)s %(message)s"
    level    = logging.INFO
)
```

通过这种配置，内容为'Hello World' 的CRITICAL 级别的日志消息将会出现在日志文件'app.log' 中。

```
CRITICAL    2005-10-25 20:46:57,126 Hello World
```

19.7.3 Logger 对象

为了发出日志消息，必须获得Logger 对象。本节描述了创建、配置和使用这些对象的过程。

1. 创建**Logger** 对象

使用下面的函数可以创建新的Logger 对象：

```
getLogger([logname
])
```

返回与名称 *logname* 相关的Logger 实例。如果不存在这样的对象，就会创建并返回一个新的Logger 实例。 *logname* 是一个字符串，用于指定一个名称或一系列由句点隔开的名称，如'app' 或'app.net' 。如果省略 *logname* ，获得的将是与根记录器相关的Logger 对象。

创建Logger 实例与在大多数其他库模块中有所不同。创建Logger 时，必须要给它取一个名字，方法是把它放在 *logname* 参数中传递给getLogger() 方法。在内部，getLogger() 方法使用一块缓存来存放Logger 实例及其相关名称。如果程序的其他部分请求名称相同的记录器，就会返回以前创建的实例。这种方式极大地简化了大型应用程序中对日志消息的处理，因为用户不必去思考如何在不同的程序模块之间传递Logger

实例。相反，在每个需要记录日志的模块中，只要使用getLogger()方法获得对正确Logger对象的引用即可。

2．挑选名称

使用getLogger()方法时应该挑选有意义的名称，稍后会说明这样做的理由。例如，如果应用程序的名称为'app'，在应用程序的每个程序模块顶部至少也应该使用getLogger('app')，例如：

```
import logging
log = logging.getLogger('app')
```

还可以考虑给记录器添加模块名称，如getLogger('app.net')或getLogger('app.user')，从而更加清楚地指示日志消息的源头。这可以使用下面这样的语句来完成：

```
import logging
log = logging.getLogger('app.'+__name__)
```

添加模块名称使得有选择性地关闭或重新配置特定程序模块的日志记录变得更加容易，稍后我们将会讲到这一点。

3．发出日志消息

如果log是Logger对象（使用前面内容中讲到的getLogger()函数创建）的实例，那么可以使用以下方法在不同的日志记录级别上发出日志消息。

日志记录级别	方　　法
CRITICAL	*log* .critical(*fmt* [, **args* [, *exc_info* [, *extra*]]])
ERROR	*log* .error(*fmt* [, **args* [,*exc_info* [, *extra*]]])
WARNING	*log* .warning(*fmt* [, **args* [,*exc_info* [, *extra*]]])
INFO	*log* .info(*fmt* [, **args* [,*exc_info* [, *extra*]]])
DEBUG	*log* .debug(*fmt* [, **args* [,*exc_info* [, *extra*]]])

fmt 参数是一个格式字符串，用于指定日志消息的格式。 *args* 中的其他参数用作格式字符串中的格式说明符。字符串格式化运算符%基于这些参数来组织结果消息。如果

提供了多个参数，它们将被放入元组中用于格式化。如果只提供一个参数，格式化时将直接把它放在% 之后。

因此，如果传递一个字典作为参数，那么格式字符串可以包含字典关键字名称。下面这个例子说明了这一点：

```
log = logging.getLogger("app")
# 使用位置格式化的日志消息
log.critical("Can't connect to %s at port %d", host, port)

# 使用字典格式化的日志消息
parms = {
     'host' : 'www.python.org',
     'port' : 80
}
log.critical("Can't connect to %(host)s at port %(port)d", parms)
```

如果将关键字参数 *exc_info* 置为True ，将把来自sys.exc_info() 的异常信息添加给日志消息。如果将 *exc_info* 置为一个异常元组，如sys.exc_info() 返回的异常元组，就会使用这些信息。 *extra* 关键字参数是一个字典，它提供另外的值，可在日志消息格式字符串（前面提到过）中使用。 *exc_info* 和 *extra* 参数都必须指定为关键字参数。

发出日志消息时，应该避免在发出消息时带有字符串格式化的代码（即格式化一条消息，然后把结果传递到日志记录模块中）。例如，

```
log.critical("Can't connect to %s at port %d" % (host, port))
```

在这个例子中，字符串格式化操作发生在调用log.critical() 之前，因为传递给函数或方法的参数必须被完全求值。然而在上面的例子中，用于字符串格式化操作的参数仅仅被传递给了logging 模块，而且只在实际要处理日志消息时使用。这种差异十分微妙，但是因为很多应用程序都选择筛选日志消息或者仅在调试期间生成日志，当禁用日志记录时，第一种方法执行的工作量更少，而且运行速度更快。

除了上面给出的方法之外，还有另外一些方法可用于在一个Logger 实例 *log* 上发出日志消息。

```
log
.exception(fmt
 [, *args
 ])
```

发出一条ERROR 级别的消息，但会添加来自当前正在处理的异常的异常信息。此方法只能用在except 代码块中。

```
log
.log(level
, fmt
 [, *args
[, exc_info
 [, extra
]]])
```

发出一条日志记录消息，其级别由 *level* 指定。如果日志记录级别由一个变量确定，或者要使用的日志级别不在5种基本级别之列，可以使用这个方法。

```
log
.findCaller()
```

返回一个元组(*filename* , *lineno* , *funcname*)，分别对应于调用者的源文件名称、行号和函数名称。发出日志消息时，有时会用到这些信息，例如，要添加对一条消息的日志调用位置的信息时。

4. 筛选日志消息

每个Logger 对象 *log* 都有一个内部级别和筛选机制，用于确定要处理哪些日志消息。下面这两种方法根据日志消息的级别执行简单的筛选。

```
log
.setLevel(level
```

```
)
```

设置 *log* 的级别。只有级别大于或等于 *level* 的日志记录消息才会得到处理。所有其他的消息都将被忽略。默认的级别是logging.NOTSET，表示处理所有的日志消息。

```
log
.isEnabledFor(level
)
```

如果将处理级别为 *level* 的日志记录消息，返回True。

还可以基于与消息本身相关的信息来筛选日志记录消息，如文件名、行号和其他细节，这时可以使用以下方法。

```
log
.addFilter(filt
)
```

给记录器添加筛选器对象 *filt* 。

```
log
.removeFilter(filt
)
```

从记录器删除筛选器对象 *filt* 。

在这两个方法中， *filt* 是Filter 对象的实例。

```
Filter(logname
)
```

创建筛选器，只允许来自 *logname* 或其子日志的日志消息通过。例如，如果 *logname* 是'app'，那么来自记录器，如'app'、'app.net' 或'app.user' 的消息就可以通过，但来自记录器（如'spam'）的消息就无法通过。

可以创建自定义的筛选器，方法是声明Filter 类的子类并实现方法filter(*record*)，该方法接受的输入是包含日志记录消息相关信息的记录。根据消息是否应该得到处理，返回True 或False 。传递给此方法的 *record* 对象通常具有以下属性。

属　　性	描　　述	属　　性	描　　述
record .name	记录器名称	*record* .lineno	发出日志消息的行号
record .levelname	级别名称	*record* .funcName	发出日志消息的函数名称
record .levelno	级别编号	*record* .created	发出日志消息的时间
record .pathname	模块的路径名称	*record* .thread	线程标识符
record .filename	基础文件名称	*record* .threadName	线程名称
record .module	模块名称	*record* .process	当前执行进程的PID
record .exc_info	异常信息		

下面的例子说明了如何创建自定义的筛选器：

```
class FilterFunc(logging.Filter):
    def __init__(self,name):
        self.funcName = name
    def filter(self, record):
        if record.funcName == self.funcName: return False
        else: return True
log.addFilter(FilterFunc('foo'))   # 忽略来自foo()函数的所有消息
log.addFilter(FilterFunc('bar'))   # 忽略来自bar()函数的所有消息
```

5. 消息传播与分层记录器

在高级日志记录应用程序中，可将Logger 对象组织为一种层次结构。这项工作可以通过给记录器对象取一个类似'app.net.client' 这样的名称来完成。在这里，实际上有三个不同的Logger 对象，分别叫做'app' 、'app.net' 和'app.net.client' 。当其中任意一个记录器发出消息并且成功通过记录器的筛选器时，这条消息将向上传播，并且所有父记录器都会对它进行处理。例如，'app.net. client' 上发出的消息也会传播到'app.net' 、'app' 和根记录器。

Logger 对象 *log* 的以下属性和方法可控制这种传播过程。

```
log
.propagate
```

这是一个布尔标志，用于指示消息是否传播给父记录器。默认值为True 。

```
log
.getEffectiveLevel()
```

返回记录器的有效级别。如果已经使用setLevel() 函数设定了级别，就会返回此级别。如果尚未显式地设定任何级别（这种情况下的级别为logging.NOTSET ），此函数将返回父记录器的有效级别。如果所有父记录器都没有设定级别，就会返回根记录器的有效级别。

分层日志记录的首要目的是能够更加容易地筛选来自大型应用程序不同部分的日志消息。例如，如果要关闭来自应用程序'app.net.client' 部分的日志消息，需要添加如下配置代码：

```
import logging
logging.getLogger('app.net.client').propagate = False
```

或者像在下面的代码中，忽略来自程序模块的除最严重级别消息之外的所有消息：

```
import logging
logging.getLogger('app.net.client').setLevel(logging.CRITICAL)
```

分层记录器有一个不易注意的方面，即是否处理日志消息完全由发出消息的Logger对象上的级别和筛选器来决定，而不是任意父记录器上的筛选器。因此，如果一条消息通过了第一组筛选器，它就能传播到所有父记录器并由它们进行处理，无论它们自己的筛选器和级别设置如何——即便这些筛选器本来会拒绝这条消息。乍一看，这种行为是违反常理的，甚至似乎是个bug。但是，将子记录器的级别设置为低于其父记录器是改写父记录器上的设置的一种途径，目的是实现一种级别提升。下面给出了一个例子：

```
import logging

# 顶级记录器'app'.
log = logging.getLogger('app')
log.setLevel(logging.CRITICAL)          # 只接收来自CRITICAL级别的消息

# 子记录器'app.net'
net_log = logging.getLogger('app.net')
net_log.setLevel(logging.ERROR)         # 接受'app.net'上的ERROR消息
                                          # 这些消息现在将由'app'记录器处理，
                                          # 即使它的级别是CRITICAL
```

使用分层记录器时，只需要配置想要修改其筛选或传播行为的日志记录对象。因为消息会自然传播到根记录器，所以它将最终负责生成输出，而且使用basicConfig()进行的所有配置都将生效。

6．消息处理

通常，消息由根记录器处理。但任何Logger对象都可以自己增加特殊的处理器，用于接收和处理日志消息。可通过Logger实例 *log* 的以下方法来完成。

```
log
.addHandler(handler
)
```

添加 *Handler* 对象给记录器。

```
log
```

```
.removeHandler(handler

)
```

从记录器删除Handler 对象 *handler* 。

logging 模块拥有各种预构建的处理器，用于将日志消息写入文件、流、系统日志等。下一节中将详细讲述这些处理器。下面的例子说明了如何使用这些方法将记录器和处理器结合在一起。

```
import logging
import sys

# 创建名为'app'的顶级记录器
app_log = logging.getLogger("app")
app_log.setLevel(logging.INFO)
app_log.propagate = False

# 给'app'日志添加一些消息处理器
app_log.addHandler(logging.FileHandler('app.log'))
app_log.addHandler(logging.StreamHandler(sys.stderr))

# 发出一些消息。这些消息将写入app.log 和 sys.stderr
app_log.critical("Creeping death detected!")
app_log.info("FYI")
```

添加自己的消息处理器来处理消息时，通常目的是改写根记录器的行为。这正是前一个例子中禁用消息传播的原因（即由'app' 记录器处理所有消息）。

19.7.4 处理器对象

logging 模块提供一些预构建的处理器，可以通过各种方式处理日志消息。使用addHandler() 方法可以将这些处理器添加给Logger 对象。另外，还可以为每个处理器配置它自己的筛选和级别。

1. 内置处理器

以下处理器对象是内置的，其中一些处理器定义在子模块logging.handlers 中，如果需要，必须专门导入这个子模块。

```
handlers.DatagramHandler(host,port

)
```

发送日志消息给位于指定 *host* 和 *port* 上的UDP服务器。日志消息的编码方法是使用对应LogRecord 对象的字典，用pickle 模块对它进行编码。传输的网络消息包含一个4字节的大尾，后面跟着序列化过的记录数据。要重新构造消息，接收者必须去掉消息头，读取整条消息，反序列化内容，并调用logging.makeLogRecord() 方法。因为UDP不可靠，所以网络错误可能导致日志消息丢失。

```
FileHandler(filename
[, mode
 [, encoding
[, delay
]]])
```

将日志消息写入文件 *filename* 。 *mode* 是打开文件时使用的文件模式，默认为'a' 。 *encoding* 是文件编码。 *delay* 是一个布尔标志，如果置为True ，就会延迟打开日志文件，直到发出首条日志消息。此标志默认为False 。

```
handlers.HTTPHandler(host
, url
 [, method
])
```

使用HTTP的GET 或POST 方法将日志消息上传到一台HTTP服务器。 *host* 指定主机计算机， *url* 是要使用的URL，而 *method* 是"GET" （默认值）或"POST" 之一。日志消息的编码方法是使用对应LogRecord 对象的字典，并使用urllib.urlencode() 函数将它编码为一组URL查询字符串变量。

```
handlers.MemoryHandler(capacity
 [, flushLevel
 [, target
]])
```

这个处理器用于收集内存中的日志消息，并定期把它们转移到另一个处理器 *target* 。 *capacity* 是内存缓冲区的大小，单位为字节。 *flushLevel* 用一个数字表示日志记录级别，指示该级别或该级别以上的日志消息应该出现在内存清空中。默认值为ERROR 。 *target* 是接收消息的另一个Handler 对象。如果省略 *target* 参数，则需要使用结果处理器对象的setTarget() 方法设置一个目标，才能让这个处理器工作。

```
handlers.NTEventLogHandler(appname
 [, dllname
 [, logtype
]])
```

发送消息给Windows NT、Windows 2000或Windows XP上的事件日志。 *appname* 是事件日志中使用的应用程序名称。 *dllname* 是.DLL 或.EXE 文件的完整路径名称，此文件提供保存在日志中的消息定义。如果省略， *dllname* 将被置为"win32service.pyd" 。 *logtype* 可以是"Application" 、"System" 或"Security" 。默认值是"Application" 。只有安装了用于Python的Win32扩展，此处理器才可用。

```
handlers.RotatingFileHandler(filename
 [, mode
 [, maxBytes
 [, backupCount
 [,

encoding [, delay
]]]]])
```

将日志消息写入文件 *filename* 。但是，如果文件的大小超出 *maxBytes* 指定的值，那么它将被备份为filename.1 ，然后打开一个新的日志文件 *filename* 。 *backupCount* 指定要创建的备份文件的最大数量。 *backupCount* 的默认值为0。但在指定时，备份文件的名称依次为 *filename* .1 、 *filename* .2…… 、 *filename* .N ，

这里的 *filename* .1 是最近的备份，而 *filename* .N 是最旧的备份。 *mode* 指定打开日志文件时使用的文件模式。默认的模式为'a' 。如果 *maxBytes* 是0（默认值），日志文件永远不会进行备份，而是可以无限增长。 *encoding* 和 *delay* 的意义与FileHanlder 中相同。

```
handlers.SMTPHandler(mailhost
, fromaddr
, toaddrs
, subject
 [, credentials
])
```

使用电子邮件将日志消息发送给一台远程主机。 *mailhost* 是一台可以接收消息的SMTP服务器的地址，此地址可以是字符串形式的简单主机名称，也可以是元组(*host* , *port*)。 *fromaddr* 是源头地址， *toaddrs* 是目的地址，而 *subject* 是消息中使用的主题。 *credentials* 是元组(*username* , *password*)，分别表示用户名和密码。

```
handlers.SocketHandler(host
, port
)
```

使用TCP套接字连接发送日志消息给一台远程主机。 *host* 和 *port* 用于指定目的地。消息发送的格式与DatagramHandler 相同。和DatagramHandler 不同的是，这个处理器能够可靠地交付日志消息。

```
StreamHandler([fileobj
])
```

将日志消息写入已经打开的类文件对象 *fileobj* 。如果不提供参数，消息将被写入sys.stderr 。

```
handlers.SysLogHandler([address
 [, facility
]])
```

发送日志消息给UNIX系统日志记录定时程序。 *address* 以元组(*host* , *port*)的形式指定目的地。如果省略，就会使用('localhost', 514) 作为目的地。 *facility* 是一个整数工具代码，默认值为SysLogHandler.LOG_USER 。在SysLogHandler 的定义中可以找到工具代码的完整列表。

```
handlers.TimedRotatingFileHandler(filename
 [, when [, interval
 [, backupCount
 [,
encoding [, delay
 [, utc
]]]]]])
```

与RotatingFileHandler 相同，但文件的备份是由时间而非大小来控制的。 *interval* 是一个数字，而 *when* 是指定单元的字符串。 *when* 的可取值是'S' （秒）、'M' （分）、'H' （小时）、'D' （天）、'W' （周）和'midnight' （在午夜备份）。例如，将 *interval* 置为3 同时将 *when* 置为'D' ，效果是每三天对日志进行一次备份。 *backupCount* 指定要保存的备份文件的最大数量。 *utc* 是一个布尔标志，用于确定是使用本地时间（默认值）还是UTC时间。

```
handlers.WatchedFileHandler(filename [, mode [, encoding [, delay]]])
```

与FileHandler 相同，但已打开日志文件的模式和设备受到监控。如果在发出最后一条日志消息之后它发生了变化，文件将被关闭，然后使用相同文件名再次打开。如果由于在运行的程序外部执行了日志备份操作，从而导致日志文件已被删除或移动，这些变化

可能就会发生。此处理器只适用于UNIX系统。

2. 处理器配置

以下方法用于为每个Handler 对象 *h* 配置它自己的级别和筛选。

```
h
.setLevel(level
)
```

设置要处理消息的阈值。 *level* 是一个数字代码，如ERROR 或CRITICAL 。

```
h
.addFilter(filt
)
```

给处理器添加Filter 对象 *filt* 。参见Logger 对象的addFilter() 方法可获得更多信息。

```
h
.removeFilter(filt
)
```

从处理器删除Filter 对象 *filt* 。

要强调一点：可以用在附加处理器的Logger 对象上使用的任何设置来单独设置该处理器上的级别和筛选器。下面这个例子说明了这一点：

```
import logging
import sys
```

```
# 创建处理器，将CRITICAL级别的消息打印到stderr
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)

# 创建名为'app'的顶级记录器
app_log = logging.getLogger("app")
app_log.setLevel(logging.INFO)
app_log.addHandler(logging.FileHandler('app.log'))
app_log.addHandler(crit_handler)
```

在这个例子中，有一个名为'app' 的INFO 级别记录器。程序给它附加了两个处理器，但其中一个处理器（crit_handler ）自己的级别设置为CRITICAL 。尽管此处理器将收到级别为INFO 或以上的日志消息，但它会有选择性地丢弃那些级别不为CRITICAL 的消息。

3．处理器清理

以下方法用于在处理器上执行清理。

```
h
.flush()
```

清除所有日志记录的输出。

```
h
.close()
```

关闭处理器。

19.7.5 消息格式化

默认情况下，当在日志记录调用中对Handler 对象进行格式化时，这些对象就会发出日志消息。但是，有时需要给消息添加另外的上下文信息，如时间戳、文件名、行号等。本节描述了如何把这种额外信息自动添加给日志消息。

1．**Formatter** 对象

要修改日志消息的格式，必须首先创建Formatter 对象。

```
Formatter([fmt [, datefmt
]])
```

创建新的Formatter 对象。 *fmt* 为消息提供格式字符串。在 *fmt* 中，可以按照前面讲过的那样为basicConfig() 函数放置各种扩展。 *datefmt* 是一个与time.strftime() 函数兼容的日期格式字符串。如果省略，日期格式将置为ISO8601格式。

必须将Formatter 对象附加到处理器对象后才能生效，此时需要使用Handler 实例 *h* 的 *h* .setFor-matter() 方法。

```
h
.setFormatter(format
)
```

设置用于在Handler 实例 *h* 上创建日志消息的消息格式化对象。 *format* 必须是Formatter 的实例。

下面这个例子说明了如何自定义处理器上的日志消息格式：

```
import logging
import sys

# 设置消息格式
format = logging.Formatter("%(levelname)-10s %(asctime)s %(message)s")

# 创建处理器将CRITICAL级别消息打印到stderr
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)
crit_hand.setFormatter(format)
```

在这个例子中，在crit_hand 处理器上设置了自定义Formatter 。如果此处理器处理一条像"Creeping death detected." 这样的日志记录消息，就会生成如下日志消息：

```
CRITICAL   2005-10-25 20:46:57,126 Creeping death detected.
```

2. 给消息添加额外的上下文

在某些应用程序中，给日志消息添加额外的上下文信息很有用。提供这些额外信息有两种方式。首先，所有基本的日志记录操作（如 *log* .critical()、 *log* .warning()等）都有一个在消息格式字符串中使用的关键字参数 *extra* ，用于提供额外字段的一个字典。这些字段与前面讲到的Formatter对象的上下文数据合并在一起。下面给出了一个例子：

```
import logging, socket
logging.basicConfig(
     format = "%(hostname)s %(levelname)-10s %(asctime)s %(message)s"
)
# 一些额外的上下文
netinfo = {
    'hostname' : socket.gethostname(),
    'ip'       : socket.gethostbyname(socket.gethostname())
}
log = logging.getLogger('app')

# 使用额外的上下文数据发出一条日志消息
log.critical("Could not connect to server", extra=netinfo)
```

这种方法的缺陷在于：必须确保每次日志记录操作包括额外的信息，否则程序会崩溃。另一种方法是使用LogAdapter类作为现有记录器的包装器。

```
LogAdapter(log

[, extra

])
```

为Logger对象 *log* 创建包装器。 *extra* 是要提供给消息格式化器的额外上下文信息的字典。LogAdapter的实例与Logger对象具有相同接口。但发出日志消息的操作将自动添加 *extra* 中提供的额外信息。

下面这个例子说明了如何使用LogAdapter对象：

```
import logging, socket
logging.basicConfig(
     format = "%(hostname)s %(levelname)-10s %(asctime)s %(message)s"
)
# 一些额外的上下文
```

```
netinfo = {
    'hostname' : socket.gethostname(),
    'ip'       : socket.gethostbyname(socket.gethostname())
}

# 创建记录器
log = logging.LogAdapter(logging.getLogger("app"), netinfo)

# 发出一条日志消息。额外的上下文数据由LogAdapter提供
log.critical("Could not connect to server")
```

19.7.6 各种实用工具函数

logging 模块中的以下函数用于控制日志记录的其他方面。

```
disable(level

)
```

全局禁用级别低于 *level* 中指定值的所有日志记录消息。此函数可用于在应用程序范围内关闭日志记录功能，例如，需要暂时禁用或减少日志记录输出的数量时。

```
addLevelName(level

, levelName

)
```

创建全新的日志记录级别和名称。 *level* 是数字，而 *levelName* 是字符串。此函数可用于修改内置级别的名称或者添加默认支持级别之外的更多级别。

```
getLevelName(level

)
```

返回对应于数字值 *level* 的级别名称。

```
shutdown()
```

关闭所有日志记录对象，必要时清除输出。

19.7.7 日志记录配置

将应用程序设置为使用logging 模块，一般包括下面几个基本步骤。

（1）使用getLogger() 函数创建各种Logger 对象。正确设置像级别这样的参数。

（2）通过实例化各类处理器（FileHandler 、StreamHandler 、SocketHandler 等）创建Handler 对象，并设置正确的级别。

（3）创建消息Formatter 对象，并使用setFormatter() 方法把它们附加给Handler 对象。

（4）使用addHandler() 方法将Handler 对象附加给Logger 对象。

因为每个步骤的配置都有所涉及，所以最好的做法是将所有日志记录配置放入一个良好的归档位置。例如，创建一个文件applogconfig.py ，然后在应用程序的主程序中导入它。

```
# applogconfig.py
import logging
import sys

# 设置消息格式
format = logging.Formatter("%(levelname)-10s %(asctime)s %(message)s")

# 创建处理器，将CRITICAL级别的消息打印到stderr
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)
crit_hand.setFormatter(format)

# 创建处理器，将消息打印到文件
applog_hand = logging.FileHandler('app.log')
applog_hand.setFormatter(format)

# 创建名为'app'的顶级记录器
app_log = logging.getLogger("app")
app_log.setLevel(logging.INFO)
app_log.addHandler(applog_hand)
app_log.addHandler(crit_hand)

# 修改'app.net'记录器上的级别
logging.getLogger("app.net").setLevel(logging.ERROR)
```

如果需要修改日志记录配置的某个部分，在同一个位置上保存所有内容更加便于维护。要牢牢记住：这个特殊文件只需要在程序中的唯一位置导入一次。在需要发出日志消息的其他代码部分中，只要包含下面这样的代码即可：

```
import logging
app_log = logging.getLogger("app")
...
app_log.critical("An error occurred")
```

logging.config 子模块

除了在Python代码中对日志记录配置进行硬编码之外，还可以通过使用INI格式的配置文件来配置日志记录模块。为此可以使用logging.config 中的以下函数：

```
fileConfig(filename

[, defaults

 [, disable_existing_loggers

]])
```

从配置文件 *filename* 读取日志记录配置。 *defaults* 是一个字典，其中包含配置文件中使用的默认配置参数。指定的文件名是使用ConfigParser 模块读取的。*disable_existing_loggers* 是一个布尔标志，用于指定当读取新的配置数据时，是否禁用任何现有记录器，它的默认值为True 。

logging 模块的在线文档非常详细地介绍了配置文件需要的格式。有经验的程序员可以模仿下面的例子，它是前面内容中提过的applogconfig.py 的配置文件版本。

```
; applogconfig.ini
;
; Configuration file for setting up logging

; The following sections provide names for Logger, Handler, and Formatter
; objects that will be configured later in the file.

[loggers]
keys=root,app,app_net

[handlers]
keys=crit,applog

[formatters]
keys=format

[logger_root]
```

```
level=NOTSET
handlers=

[logger_app]
level=INFO
propagate=0
qualname=app
handlers=crit,applog

[logger_app_net]
level=ERROR
propagate=1
qualname=app.net
handlers=

[handler_crit]
class=StreamHandler
level=CRITICAL
formatter=format
args=(sys.stderr,)

[handler_applog]
class=FileHandler
level=NOTSET
formatter=format
args=('app.log',)

[formatter_format]
format=%(levelname)-10s %(asctime)s %(message)s
datefmt=
```

要读取这个配置文件并设置日志记录，需要使用这段代码：

```
import logging.config
logging.config.fileConfig('applogconfig.ini')
```

和以前一样，要发出日志消息的模块不需要关心加载和日志记录配置的细节。它们只要导入logging 模块，然后获得对正确Logger 对象的引用即可。例如：

```
import logging
app_log = logging.getLogger("app")
...
app_log.critical("An error occurred")
```

19.7.8 性能考虑

如果不小心，给应用程序添加日志记录可能会严重降低它的性能。不过可以使用多种技巧来降低负载。

首先，Python的优化模式（-O）可以移除使用像if __debug__: statements这样的语句在特定条件下执行的所有代码。如果日志记录的唯一目的是调试，那么可以在特定条件下地执行所有日志记录调用，并在优化模式中移除所有调用。

第二种技术是当完全禁用日志记录时，使用Null对象代替Logger对象。这与使用None有所区别。相反，需要使用让对象的实例悄悄覆盖在它上面执行的所有操作。例如：

```
class Null(object):
    def __init__(self, *args, **kwargs): pass
    def __call__(self, *args, **kwargs): return self
    def __getattribute__(self, name): return self
    def __setattr__(self, name, value): pass
    def __delattr__(self,name): pass

log = Null()
log.critical("An error occurred.")    # 什么也不做
```

如果您足够聪明，还可以通过使用装饰器和元类来管理日志记录。因为Python的这些功能是在定义函数、方法和类时进行的，所以它们可以用于有选择地添加或删除程序某些部分的日志记录功能，而且在禁用日志记录时也不会影响性能。请参阅第6章和第7章了解详细信息。

19.7.9 注意

- logging模块还提供了大量自定义选项，我们这里没有讨论到。读者可以参考在线文档了解详细信息。
- 在使用线程的程序中使用logging模块是安全的。具体来说，就是不必给发出日志消息的代码添加锁定操作。

19.8 mmap

mmap模块为内存映射的文件对象提供支持。此对象的行为很像文件和字节字符串，可以用在大多数需要使用普通文件或字节字符串的地方。此外，内存映射文件的内容是可变的。也就是说，可以使用索引赋值和切片赋值操作进行修改。除非已经对文件进行了私有映射，否则这种修改会直接改动底层文件的内容。

使用mmap()函数可以创建一个内存映射文件，此函数在UNIX和Windows上略有不同。

```
mmap(fileno

, length

[, flags

, [prot
```

```
 [,access

 [, offset

]]]])
```

（UNIX）。返回mmap 对象，用于映射来自某个文件的 *length* 个字节，此文件具有一个整数文件描述符 *fileno* 。如果 *fileno* 为-1，就会映射匿名的内存。 *flags* 指定映射的性质，它可取的值如下。

标　志	意　义
MAP_PRIVATE	创建私有写时复制映射。对象的修改对于此进程是私有的
MAP_SHARED	与映射同一文件区域的所有其他进程共享映射。对象的修改将影响到所有映射

flags 的默认设置是MAP_SHARED 。 *prot* 指定对象的内存保护，是以下内容的按位OR。

设　置	意　义
PROT_READ	可以从对象读取数据
PROT_WRITE	可以修改对象
PROT_EXEC	对象可以包含可执行指示

prot 的默认值是PROT_READ | PROT_WRITE 。 *prot* 中指定的模式必须匹配用于打开底层文件描述符 *fileno* 的访问权限。大多数情况下，这意味着应该以读/写模式打开文件（如os.open(name, os.O_RDWR) ）。

可选 *access* 参数可用作 *flags* 和 *prot* 的替代参数。如果指定，它的可取值如下。

访　问	意　义
ACCESS_READ	只读访问

ACCESS_WRITE	带完全写入（write-through）的读/写访问。修改影响底层文件
ACCESS_COPY	带写时复制的读/写访问。修改影响内存，但不会修改底层文件

提供 *access* 时，它通常以关键字参数的形式出现，如mmap(*fileno* , *length* , access=ACCESS_READ)。同时为 *access* 和 *flags* 参数提供值将会引发错误。 *offset* 参数指定从文件开始的字节数，默认值为0。它必须是mmap.ALLOCATIONGRANULARITY 的倍数。

```
mmap(fileno
, length
[, tagname
 [,access
 [, offset
]]])
```

（Windows）返回mmap 对象，用于映射来自某个文件的 *length* 个字节，此文件由整数文件描述符 *fileno* 指定。如果 *fileno* 为-1，就映射匿名内存。如果 *length* 大于文件的当前值，文件将被扩展到 *length* 个字节。如果 *length* 为0，只要文件非空（否则将引发异常），就使用文件的当前长度作为 *length* 。 *tagname* 是一个可选字符串，用于给映射命名。如果 *tagname* 引用的是现有映射，就会打开该映射，否则将创建一个新的映射。如果 *tagname* 是None，就会创建一个未命名的映射。 *access* 是一个可选参数，用于指定访问模式，它使用的值与前面mmap() 的UNIX版本相同。 *access* 的默认值为ACCESS_WRITE 。 *offset* 参数指定从文件开始的字节数，默认值为0 。它必须是mmap.ALLOCATIONGR-ANULARITY 的倍数。

内存映射文件对象 *m* 支持以下方法。

```
m
.close()
```

关闭文件。后续操作将导致异常。

```
m
.find(string
[, start
])
```

返回 *string* 首次出现时的索引。 *start* 指定可选的起始位置。如果找不到匹配，则返回-1 。

```
m
.flush([offset
, size
])
```

清除内存中的副本并将修改回写到文件系统。 *offset* 和 *size* 指定了要清除字节的可选范围。如未指定则清除整个映射。

```
m
.move(dst,src,count
)
```

将从索引 *src* 开始的 *count* 个字节复制到目的索引 *dst* 处。复制使用C memmove() 函数执行，当源头和目的区域恰好重叠时，它也能保证正确工作。

```
m
.read(n
```

```
)
```

从当前文件位置读取最多 *n* 个字节，然后以字符串形式返回。

```
m
.read_byte()
```

从当前文件位置读取一个字节，并返回长度为1的字符串。

```
m
.readline()
```

返回从当前文件位置开始的一行输入。

```
m
.resize(newsize
)
```

重新调整内存映射对象的大小，以便容纳 *newsize* 个字节。

```
m
.seek(pos
[, whence
])
```

将文件位置设置为一个新值。 *pos* 和 *whence* 的意义与文件对象上的seek()方法相同。

```
m
.size()
```

返回文件的长度。这个值可能大于内存映射区域的大小。

```
m
.tell()
```

返回文件指针的值。

```
m
.write(string
)
```

将一个字节字符串写入当前文件指针所在的文件。

```
m
.write_byte(byte
)
```

将一个字节写入当前文件指针所在的内存。

注意

- 尽管UNIX和Windows提供的mmap() 函数略有不同，还是可以通过依赖这两个函数共有的可选参数 *access* ，以可移植的方式使用这个模块。例如，mmap(*fileno* ,*length* ,access=ACCESS_WRITE) 能够同时在UNIX和Windows上工作。
- 只有在长度是系统页面大小（包含在常量mmap.PAGESIZE 中）的倍数时，某些内存映射才能工作。
- 在UNIX SVR4系统上，可以通过在使用正确权限打开的文件/dev/zero上调用mmap() 函数来获得匿名的映射内存。
- 在UNIX BSD系统上，可以通过调用mmap() 与负数文件描述符和使用标志 mmap.MAP_ANON 来获得匿名的映射内存。

19.9 msvcrt

msvcrt 模块提供对Microsoft Visual C运行时中大量有用函数的访问。此模块只能在Windows上使用。

getch()

读取一次键击事件并返回结果字符。如果不存在任何键击事件，调用此函数将会阻塞。如果按下的键是特殊的功能键，调用将返回'\000' 或'\xe0' ，而下一次调用将返回键码。此函数不会将字符回显到控制台，也不能用于读取Ctrl+C。

getwch()

同getch() 函数，但返回的是Unicode字符。

getche()

很像getch() 函数，但要回显字符（如果可以打印）。

getwche()

同getche() 函数，但返回的是Unicode字符。

get_osfhandle(*fd*

)

返回文件描述符 *fd* 的文件句柄。如果 *fd* 不能识别则引发IOError 异常。

heapmin()

强制让内部的Python内存管理器把未使用的块归还给操作系统。此函数只能在Windows NT上使用，如果失败将引发IOError 异常。

kbhit()

如果正在等待读取一个键击，则返回True 。

locking(*fd, mode, nbytes*

)

锁定部分文件，从C运行时指定文件描述符。 *nbytes* 是相对于当前文件指针的要锁定的字节数。 *mode* 可取以下整数。

设　置	描　述
0	解锁文件区域 (LK_UNLCK)
1	锁定文件区域(LK_LOCK)
2	锁定文件区域：非阻塞(LK_NBLCK)
3	锁定以便写入 (LK_RLCK)
4	锁定以便写入：非阻塞(LK_NBRLCK)

尝试获得锁定的过程如果超过大约10秒钟，将导致IOError 异常。

```
open_osfhandle(handle
, flags
)
```

从文件句柄 *handle* 创建一个C 运行时文件描述符。 *flags* 是os.O_APPEND 、os.O_RDONLY 和os.O_TEXT 的按位OR 。返回的整数文件描述符可用作os.fdopen() 函数创建文件对象时使用的　参数。

```
putch(char
)
```

在没有缓存的情况下，将字符 *char* 打印到控制台。

```
putwch(char
)
```

同putch() 函数，但 *char* 是Unicode字符。

```
setmode(fd
, flags
)
```

为文件描述符 *fd* 设置行结束转换模式。 *flags* 的值是os.O_TEXT 时代表文本模式，是os.O_BINARY 时代表二进制模式。

```
ungetch(char)
```

使字符 *char* “推回”到控制台缓冲区中。它将是getch() 或getche() 函数读取的下一个字符。

```
ungetwch(char)
```

同ungetch() 函数，但 *char* 是Unicode字符。

> 注意
>
> 存在大量Win32扩展可用于访问Microsoft Foundation Classes、COM组件、图形用户界面等。这些主题已经远远超出了本书的范围，关于这众多主题的详细信息可以在Mark Hammond和Andy Robinson所著的Python Programming on Win32（O’Reilly & Associates，2000）一书中找到。此外，http://www.python.org上也列出了在Windows下使用的模块的完备清单。

另请参见： 有关winreg的内容。

19.10 optparse

optparse 模块为处理sys.argv 中提供的UNIX风格命令行选项提供高级支持。第9章中给出了使用此模块的一个简单例子。使用optparse 模块的关键是了解OptionParser 类。

```
OptionParser([**args
])
```

创建一个新的命令选项解析器，并以OptionParser 实例的形式返回。可以提供各种可选的关键字参数来控制配置。下面的列表中描述了这些关键字参数。

关键字参数	描　　述
add_help_option	指定是否支持特殊的帮助选项（--help 和-h ）。默认值为Ture
conflict_handler	指定对出现冲突的命令行选项的处理。可以将它设置为'error' （默认值）或'resolve' 。在'error' 模式中，如果给解析器添加出现冲突的选项字符串，将会引发optparse.OptionConflictError 异常。在'resolve' 模式中，冲突将被解决，解决方法是后添加的选项优先。然而，如果前面添加的选项具有多个名称，而且其中至少一个名称不存在冲突，这些选项仍然可用
description	为程序提供描述的字符串，可在帮助中显示出来。此字符串将自动被格式化，从而在显示时适应屏幕尺寸
formatter	optparse 的一个实例。打印帮助时用于格式化文本的HelpFormatter 类。可以是optparse.IndentedHelpFormatter （默认值）或optparse.TitledHelpFormatter
option_class	用于每个命令行选项的相关信息的Python类。默认的类是optparse.Option
option_list	用于填充解析器的选项列表。默认情况下，这个列表是空的，要使用add_option() 方法添加选项。如果提供，此列表将包含类型Option 的对象
prog	用于替换帮助文本中'%prog' 的程序名称
usage	使用-help 选项或者在传递的选项不正确时打印的用法字符串。默认值是字符串'%prog [options]' ，其中'%prog' 关键字用os.path.basename(sys.argv[0]) 或prog 关键字参数（如果提供）的值代替。可以将它赋值为optparse.SUPPRESS_USAGE ，表示完全禁用用法消息
version	提供-version 选项时打印的版本字符串。默认情况下，version 的值为None ，表示不添加--version 选项。提供此字符串时，将自动添加-version 选项。特殊关键字'%prog' 将被程序名称代替

除非真正需要通过某些方式自定义选项处理，否则创建OptionParser 对象时通常不带参数。例如：

```
p = optparse.OptionParser()
```

OptionParser 的实例 *p* 支持以下方法：

```
p
.add_option(name1
, ..., nameN
 [, **parms
])
```

给 *p* 添加一个新的选项。参数 *name1* 、 *name2* 等分别是所有选项的名称。例如，包含的选项名称无论长短都是可以的，如'-f' 和'--file' 。紧随选项名称之后的是一组可选的关键字参数，用于指定解析时如何处理选项。下面的列表描述了这些关键字参数。

关键字参数	描　　述
action	解析选项时执行的动作。可接受的值如下： 'store' ——选项有一个参数需要读取和保存。如果没有显式指定任何动作，这就是默认动作。 'store_const' ——选项不带任何参数，但当遇到选项时，就会保存 *const* 关键字参数指定的常量值。 'store_true' ——很像'store_const' ，但解析选项时保存的是布尔类型的True 值。 'store_false' ——很像'store_true' ，但解析选项时保存的是布尔类型的False 值。 'append' ——选项有一个参数，解析时被附加到一个列表中。如果使用相同的命令行参数指定多个值，就要使用这个值。 'count' ——选项不带任何参数，但保存一个计数器值。每次遇到参数时，计数器的值就会增加。 'callback' ——遇到选项时，调用使用callback 关键字参数指定的一个回调函数。 'help' ——解析选项时打印一条帮助消息。只有需要通过标准的-h 或--help 选择之外的选项显示帮助时，才需要使用这个值。 'version' ——打印提供给OptionParser() 的参数，如果有的话。只有需要通过标准的-v 或--version 选择之外的选项显示帮助时，才需要使用这个值。
callback	指定遇到选项时调用的回调函数。这个回调函数是Python可调用对象，调用语法是callback(*option , opt _str , value ,parser , *args , **kwargs*) 。选项参数是一个optparse.Option 实例， *opt_str* 是触发回调的命令行上提供的选项字符串， *value* 是选项（如果有的话）的值， *parser* 是正在运行的 OptionParser 的实例， *args* 是使用callback_args 关键字参数提供的位置参数，而 *kwargs* 是使用callback_kwarg 关键字参数

	提供的关键字参数。
callback_args	提供给callback 参数指定的回调函数的可选位置参数
callback_kwargs	提供给callback 参数指定的回调函数的可选关键字参数
choices	指定所有可能的选项值的字符串列表。当一个选项只有一组有限的值（['small', 'medium', 'large']）时使用。
const	通过'store_const' 动作保存的常量值
default	如果未提供，设置选项的默认值。默认情况下，默认值是None
dest	设置用于保存解析期间的选项值的属性名称。名称通常来自于选项名称本身
help	这个特定选项的帮助文本。如果未提供，帮助中列出此选项时将不带描述。可以使用值optparse.SUPPRESS_HELP 隐藏选项。特殊关键字'%default' 将被帮助字符串中的选项默认值替换
metavar	指定打印帮助文本时使用的选项参数的名称
nargs	为需要参数的动作指定选项参数的数量。默认值为1。如果使用的数字大于1，选项参数将被收集到一个元组中，处理参数时将用到这个元组
type	指定选项的类型。有效类型包括'string'（默认值）、'int'、'long'、'choice'、'float'、和'complex'

```
p
.disable_interspersed_args()
```

不接受简单选项和位置参数的混合使用。例如，如果'-x' 和'-y' 是不带参数的选项，选项必须出现在任意参数之前，如'prog -x -y arg1 arg2 arg3' 。

```
p
.enable_interspersed_args()
```

选项与位置参数可以混合使用。例如，如果'-x' 和'-y' 是不带参数的选项，那么它们可以与参数（如'prog -x arg1 arg2 -y arg3' ）混合在一起。这是默认行为。

```
p
.parse_args([arglist
])
```

解析命令行选项并返回一个元组（ *options , args* ），其中 *options* 是包含所有选项的值的对象，而 *args* 是所有余下位置参数的列表。 *options* 对象使用与选项名称匹配的属性名称来保存所有选项数据。例如，选项'--output' 的值将保存在options.output 中。如果选项没有出现，值将为None 。如前所述，可以使用add_option() 的dest 关键字参数设置属性名称。默认情况下，参数取自sys.argv [1:] 。然而，还可以提供可选参数 *arglist* ，作为参数的另一个来源。

```
p
.set_defaults(dest=value
, ... dest=value
)
```

设置特定选项目的地的默认值。只要提供需要设置的目的地的关键字参数即可。关键字参数的名称应该与在add_option() 函数中使用 *dest* 参数指定的名称相匹配。

```
p
.set_usage(usage
)
```

修改由--help 选项生成的文本中显示的用法字符串。

19.10.1 例子

```
# foo.py
import optparse
p = optparse.OptionParser()

# 简单的选项，不带参数
p.add_option("-t", action="store_true", dest="tracing")

# 接受字符串参数的选项
p.add_option("-o", "--outfile", action="store", type="string", dest="outfile")

# 需要整数参数的选项
p.add_option("-d", "--debuglevel", action="store", type="int", dest="debug")

# 带有一些选择的选项
p.add_option("--speed", action="store", type="choice", dest="speed",
             choices=["slow","fast","ludicrous"])

# 带有多个参数的选项
p.add_option("--coord", action="store", type="int", dest="coord", nargs=2)

# 一组控制常用目的地的选项
p.add_option("--novice", action="store_const", const="novice", dest="mode")
p.add_option("--guru", action="store_const", const="guru", dest="mode")

# 为各个选项目的地设置默认值
p.set_defaults(tracing=False,
               debug=0,
               speed="fast",
               coord=(0,0),
               mode="novice")

# 解析参数
opt, args = p.parse_args()

# 打印选项值
print "tracing :", opt.tracing
print "outfile :", opt.outfile
print "debug   :", opt.debug
print "speed   :", opt.speed
print "coord   :", opt.coord
print "mode    :", opt.mode

# 打印余下的参数
print "args    :", args
```

下面这个简短的交互式UNIX会话说明了上面这段代码的工作方式。

```
% python foo.py -h
```

```
usage: foo.py [options]

options:
  -h, --help         show this help message and exit
  -t
  -o OUTFILE, --outfile=OUTFILE
  -d DEBUG, --debuglevel=DEBUG
  --speed=SPEED
  --coord=COORD
  --novice
  --guru

% python foo.py -t -o outfile.dat -d 3 --coord 3 4 --speed=ludicrous blah


tracing   : True
outfile   : outfile.dat
debug     : 3
speed     : ludicrous
coord     : (3, 4)
mode      : novice
args      : ['blah']

% python foo.py --speed=insane


usage: foo.py [options]

foo.py:error:option --speed:invalid choice:'insane'
(choose from 'slow', 'fast', 'ludicrous')
```

19.10.2 注意

- 指定选项名称时，使用破折号来指定较短的名称，如'-x'，而使用双破折号来指定较长的名称，如'--exclude'。如果定义混合这两种风格的选项，如'-exclude'，就会引发OptionError 异常。
- Python还包含一个getopt 模块，它也支持命令行解析，但是风格类似于与其同名的C库。从实用的角度考虑，使用该模块代替optparse 模块（级别高得多，同时需要编写代码少得多）没有任何好处。
- optparse 模块包含数量可观的高级功能，可用于自定义和专门处理某些类别的命令行选项。然而，最常用的命令行选项解析都用不到这些功能。读者应该参考在线库文档，了解更多细节和使用示例。

19.11 os

os 模块提供到常用操作系统服务的可移植接口。接口的实现方式是搜索依赖于OS 的内置模块，如nt 或posix，然后导出模块中的函数和数据。除非特别指出，否则函数在Windows和UNIX上都是可用的。UNXI系统包括Linux和Mac OS X。

此模块定义了以下通用的变量。

environ

表示当前环境变量的映射对象。对映射的修改将反映在当前环境中。如果putenv()函数也可用，那么修改也会反映在子进程中。

linesep

用于在当前平台上分隔行的字符串。在POSIX上可能是一个字符，如'\n'，在Windows上可能是多个字符如'\r\n'。

name

导入的依赖于OS 的模块名称：'posix'、'nt'、'dos'、'mac'、'ce'、'java'、'os2'或'riscos'。

path

用于路径名称操作的依赖于OS的标准模块。还可以使用import os.path 语句加载此模块。

19.11.1 进程环境

以下函数用于访问和修改与运行进程的环境相关的各个参数。除非特别指出，否则进程、组、进程组和会话ID都是整数。

chdir(*path*)

将当前的工作目录修改为 *path* 。

```
chroot(path
)
```

修改当前进程的根目录（UNIX）。

```
ctermid()
```

返回带有进程控制终端的文件名称的字符串（UNIX）。

```
fchdir(fd
)
```

修改当前的工作目录。 *fd* 是已打开目录的文件描述符（UNIX）。

```
getcwd()
```

返回带有当前工作目录的字符串。

```
getcwdu()
```

返回带有当前工作目录的Unicode字符串。

```
getegid()
```

返回有效的组ID（UNIX）。

```
geteuid()
```

返回有效的用户ID（UNIX）。

```
getgid()
```

返回进程的真实组ID（UNIX）。

```
getgroups()
```

返回进程所有者所属的整数组ID的列表（UNIX）。

```
getlogin()
```

返回与有效用户ID相关的用户名（UNIX）。

```
getpgid(pid
)
```

返回ID为 *pid* 的进程的进程组ID。如果 *pid* 为0，就会返回调用进程的进程组（UNIX）。

getpgrp()

返回当前进程组的ID。进程组通常与任务控制联合使用。进程组并不一定与进程的组ID相同（UNIX）。

getpid()

返回当前进程的真实进程ID（UNIX和Windows）。

getppid()

返回父进程的进程ID（UNIX）。

getsid(*pid*

)

返回进程 *pid* 的进程会话标识符。如果 *pid* 为0，将返回当前进程的标识符（UNIX）。

getuid()

返回当前进程的真实用户ID（UNIX）。

```
putenv(varname, value
)
```

将环境变量 *varname* 设为 *value* 。修改将影响到以os.system()、popen()、fork() 和execv() 开始的子进程。给os.environ 中的项赋值将自动调用putenv() 函数。但调用putenv() 函数不会更新os.environ （UNIX和Windows）。

```
setegid(egid
)
```

设置有效的组（UNIX）。

```
seteuid(euid
)
```

设置有效的用户ID（UNIX）。

```
setgid(gid
)
```

设置当前进程的组ID（UNIX）。

```
setgroups(groups
)
```

设置当前进程的组访问列表。 *groups* 是指定组标识符的一系列整数。只能由root调用（UNIX）。

```
setpgrp()
```

通过调用系统调用setpgrp() 或setpgrp (0, 0)创建新进程，这取决于所实现的版本（如果有的话）。返回新进程组的ID（UNIX）。

```
setpgid(pid
, pgrp
)
```

将进程 *pid* 赋给进程组 *pgrp* 。如果 *pid* 等于 *pgrp* ，进程将成为一个新的进程组主进程。如果 *pid* 不等于 *pgrp* ，进程将加入一个现有的组。如果 *pid* 为0，将使用调用进程的进程ID。如果 *pgrp* 为0， *pid* 指定的进程将成为一个进程组主进程（UNIX）。

```
setreuid(ruid,euid
)
```

设置调用进程的真实而有效的用户ID（UNIX）。

```
setregid(rgid,egid
)
```

设置调用进程的真实而有效的组ID（UNIX）。

```
setsid()
```

创建一个新的会话并返回新创建的会话ID。会话通常与在它们中启动的终端设备和进程任务控制相关（UNIX）。

```
setuid(uid
)
```

设置当前进程的真实用户ID。此函数优先级较高，通常只有以root 运行的进程才能执行它（UNIX）。

```
strerror(code
)
```

返回对应于整数错误代码的错误消息（UNIX和Windows）。errno 模块为这些错误代码定义了符号名称。

```
umask(mask
)
```

设置当前的数字umask，并返回以前的。umask 用于清除由进程创建的文件上的权限位（UNIX和Windows）。

```
uname()
```

返回字符串元组(*sysname* , *nodename* , *release* , *version* , *machine*)，用于标识系统类型（UNIX）。

```
unsetenv(name

)
```

删除环境变量 *name* 。

19.11.2 文件创建与文件描述符

以下函数提供了用于操作文件和管道的底层接口。在这些函数中，被操作的文件是以整数文件描述符 *fd* 的形式出现的。调用fileno() 方法可以查看文件对象的文件描述符。

```
close(fd

)
```

关闭以前由open() 或pipe() 函数返回的文件描述符 *fd* 。

```
closerange(low

, high

)
```

关闭范围在 *low <= fd < high* 内的所有文件描述符 *fd* 。忽略错误。

```
dup(fd

)
```

复制文件描述符 *fd* 。返回一个新的文件描述符，它是进程未使用的文件描述符中编号最小的描述符。新的和旧的文件描述符可以交换使用。此外，它们还共享状态，如当前的文件指针和锁（UNIX和Windows）。

```
dup2(oldfd
, newfd
)
```

将文件描述符 *oldfd* 复制给 *newfd* 。如果 *newfd* 已经对应一个有效的文件描述符，那么它将被首先关闭（UNIX和Windows）。

```
fchmod(fd
, mode
)
```

将与 *fd* 相关的文件模式修改为 *mode* 。关于 *mode* 的描述可参照对`os.open()`函数的描述（UNIX）。

```
fchown(fd
, uid
, gid
)
```

将与 *fd* 相关的文件的所有者和组ID修改为 *uid* 和 *gid* 。设置 *uid* 或 *gid* 为-1可以保持值不变（UNIX）。

```
fdatasync(fd
)
```

强制将所有缓存数据写入 *fd* ，以便将这些数据保存到磁盘上（UNIX）。

```
fdopen(fd
 [, mode
 [, bufsize
]])
```

创建连接到文件描述符 *fd* 的已打开文件对象。 *mode* 和 *bufsize* 参数的意义与内置函数open() 中相同。 *mode* 是一个字符串，如'r'、'w'或'a' 。在Python 3中，此函数接受能够用于内置函数open() 的所有参数，如编码和行结束的规范。但是，如果需要考虑与Python 2的可移植性，应该只使用这里描述的 *mode* 和 *bufsize* 参数。

```
fpathconf(fd
, name
)
```

返回与描述符为 *fd* 的已打开文件相关的可配置路径名称变量。 *name* 是一个字符串，指定要获取值的名称。值通常取自系统头文件中包含的参数，如<limits.h> 和 <unistd.h> 。POSIX为 *name* 定义了以下常量。

常　　量	描　　述
"PC_ASYNC_IO"	指示是否在 *fd* 上执行匿名的I/O
"PC_CHOWN_RESTRICTED"	指示是否能够使用chown() 函数。如果 *fd* 引用一个目录，这适用于该目录中的所有文件
"PC_FILESIZEBITS"	文件的最大大小
"PC_LINK_MAX"	文件的链接计数的最大值

"PC_MAX_CANON"	格式化输入行的最大长度。 *fd* 引用的是一台终端
"PC_MAX_INPUT"	输入行的最大长度。 *fd* 引用的是一台终端
"PC_NAME_MAX"	目录中一个文件的最大长度
"PC_NO_TRUNC"	指示在目录中尝试使用大于PC_NAME_MAX 的名称长度创建文件时，是否会失败并报出ENAMETOOLONG 错误
"PC_PATH_MAX"	当目录 *fd* 是当前工作目录时，相对路径名称的最大长度
"PC_PIPE_BUF"	当 *fd* 引用管道或FIFO时，管道缓冲区的大小
"PC_PRIO_IO"	指示是否能在 *fd* 上执行优先I/O
"PC_SYNC_IO"	指示是否能在 *fd* 上执行同步I/O
"PC_VDISABLE"	指示 *fd* 是否允许禁用特殊字符的处理。 *fd* 必须引用一台终端

并非所有的名称都在所有平台上可用，有些系统可能定义了另外的配置参数。但是在字典os.pathconf_names 中可以找到操作系统已知的名称列表。如果某个已知的配置名称没有包含在os.pathconf_names 中，也可以把它的整数值作为 *name* 传递。即使Python能够识别出名称，但如果主机操作系统不能识别参数或者将它与文件 *fd* 关联，那么此函数仍会引发OSError 异常。此函数只在某些版本的UNIX上可用。

```
fstat(fd)
```

返回文件描述符 *fd* 的状态。返回值与os.stat() 函数相同（UNIX和Windows）。

```
fstatvfs(fd
)
```

返回包含文件描述符 *fd* 相关文件的文件系统的相关信息。返回值与os.statvfs() 函数相同（UNIX）。

```
fsync(fd
)
```

强制将 *fd* 中已经写入的数据写入磁盘。注意，如果使用不带缓存I/O的对象（如Python的`file`对象），那么在调用`fsync()`函数之前应该首先清除数据。在UNIX和Windows上均可用。

```
ftruncate(fd
, length
)
```

截取对应于文件描述符 *fd* 的文件，从而让该文件的大小最大不超过 *length* 个字节（UNIX）。

```
isatty(fd
)
```

如果 *fd* 关联到一台TTY类型的设备，如终端，则返回`True`（UNIX）。

```
lseek(fd
, pos
, how
)
```

将文件描述符 *fd* 的当前位置设为位置 *pos* 。 *how* 的可取值如下：`SEEK_SET`用于设置相对于文件开始处的位置，`SEEK_CUR`用于设置相对于当前文件描述符的位置，而`SEEK_END`用于设置相对于文件结尾处的位置。在旧式Python代码中，经常能够看到这

些常量分别由于它们对应的数字值0、1或2所代替。

```
open(file
[, flags
 [, mode
]])
```

打开 *file* 文件。 *flags* 是以下常量值的按位OR。

值	描　　述
O_RDONLY	打开文件以供读取
O_WRONLY	打开文件以供写入
O_RDWR	打开文件以供读取和写入（更新）
O_APPEND	将字节附加到文件结尾
O_CREAT	如果文件不存在则创建它
O_NONBLOCK	不要阻塞打开、读取或写入（UNIX）
O_NDELAY	同O_NONBLOCK （UNIX）
O_DSYNC	同步写入（UNIX）
O_NOCTTY	打开一台设备时，不要设置控制终端（UNIX）
O_TRUNC	如果文件存在，则将它截取为0长度
O_RSYNC	同步读取（UNIX）
O_SYNC	同步写入（UNIX）

O_EXCL	如果O_CREAT 和文件已经存在，则报错
O_EXLOCK	在文件上设置一个排它锁
O_SHLOCK	在文件上设置一个共享锁
O_ASYNC	启用异步输入模式，在此模式中能够使用输入生成一个SIGIO 信号
O_DIRECT	使用直接I/O模式，使读取和写入都直接作用到磁盘，而不是操作系统读/写缓存
O_DIRECTORY	如果文件不是一个目录，将引发一个错误
O_NOFOLLOW	不获取符号链接
O_NOATIME	不更新文件的最后一次访问时间
O_TEXT	文本模式（Windows）
O_BINARY	二进制模式（Windows）
O_NOINHERIT	子进程不继承文件（Windows）
O_SHORT_LIVED	短期保存文件的系统提示（Windows）
O_TEMPORARY	关闭时删除文件（Windows）
O_RANDOM	随机保存文件的系统提示（Windows）
O_SEQUENTIAL	按序访问文件的系统提示（Windows）

同步I/O模式（O_SYNC 、O_DSYNC 、O_RSYNC ）强制阻塞I/O操作，直到它们在硬件层次上完成为止（例如，一次写入将阻塞，直到字节被真正写到磁盘上为止）。 *mode* 参数包含文件权限，表现为以下八进制值（它们是定义在stat 模块中的常量）的按位或：

模　　式	意　　义
0100	用户具有执行权限（stat.S_IXUSR ）

0200	用户具有写入权限（stat.S_IWUSR ）
0400	用户具有读取权限（stat.S_IRUSR ）
0700	用户具有读取/写入/执行权限（stat.S_IRWXU ）
0010	组具有执行权限（stat.S_IXGRP ）
0020	组具有写入权限（stat.S_IWGRP ）
0040	组具有读取权限（stat.S_IRGRP ）
0070	组具有读取/写入/执行权限（stat.S_IRWXG ）
0001	其余具有执行权限（stat.S_IXOTH ）
0002	其余具有写入权限（stat.S_IWOTH ）
0004	其余具有读取权限（stat.S_IROTH ）
0007	其余具有读取/写入/执行权限（stat.S_IRWXO ）
4000	设置UID模式（stat.S_ISUID ）
2000	设置GID模式（stat.S_ISGID ）
1000	设置sticky位（stat.S_ISVTX ）

文件的默认模式是（0777 &~umask ），这里的umask 设置用于删除所选的权限。例如，umask 为0022 时，将删除组和其余的写入权限。使用os.umask() 函数可以修改umask 。umask 设置对Windows毫无影响。

openpty()

打开一台伪终端并返回PTY和TTY的一对文件描述符(*master ,slave*)，可在一些

版本的UNIX上使用。

```
pipe()
```

创建一个管道，可用于与另一进程建立单向通信。返回一对文件描述符(*r, w*)，分别用于读取和写入。此函数通常在执行fork() 函数之前调用。在fork() 函数之后，发送进程将关闭管道的读取端，而接收进程将关闭管道的写入端。至此管道被激活，可以使用read() 和write() 函数将数据从一个进程发送到另一个进程（UNIX）。

```
read(fd
, n
)
```

从文件描述符 *fd* 读取最多 *n* 个字节。返回一个字节字符串，其中包含读取出的字节。

```
tcgetpgrp(fd
)
```

返回与 *fd* 指定的控制终端相关联的进程组（UNIX）。

```
tcsetpgrp(fd
, pg
)
```

设置与 *fd* 指定的控制终端相关联的进程组（UNIX）。

```
```

```
ttyname(fd
)
```

返回一个字符串，用于指定与文件描述符 *fd* 相关联的终端设备。如果 *fd* 没有与终端设备相关联，就会引发OSError 异常（UNIX）。

```
write(fd
, str
)
```

将字节字符串 *str* 写入文件描述符 *fd* ，并返回实际写入的字节数。

19.11.3 文件与目录

以下函数和变量用于操作文件系统上的文件和目录。为了处理文件命名约定方面的差异，以下变量包含与路径名称构造相关的信息。

变量	描　　述
altsep	OS用于分隔路径名称组件的另一个字符，或者如果只存在一个分隔字符，则为None 。这在DOS和Windows系统上被设为'/' ，其中 *sep* 是一个反斜杠
curdir	用于引用当前工作目录的字符串：在UNIX和Windows上为'.' ，在Macintosh上为 ':'
devnull	空设备的路径（如/dev/null ）
extsep	用于将基本文件名称与其类型分离的字符（如'foo.txt' 中的'.' ）
pardir	用于引用父目录的字符串：在UNIX和Windows上为'..' ，在Macintosh上为'::'
pathsep	用于分隔搜索路径组件的字符（如包含在环境变量$PATH 中的字符）：在UNIX上为':' ，在DOS和Windows上为';'
sep	用于分隔路径名称组件的字符：在UNIX和Windows上为'/' ，在Macintosh上为':'

以下函数用于操作文件。

```
access(path, accessmode
)
```

检查此进程访问文件path 的读取/写入/执行权限。 *accessmode* 的值是R_OK 、W_OK 、X_OK 或F_OK ， 分别对应于读取、写入、执行或存在。如果访问得到授权返回1，否则返回0。

```
chflags(path, flags
)
```

修改path 上的文件标志。 *flags* 是下面所列常量的按位OR。以UF_ 开头的标志可以由任何用户设置，而以SF_ 开头的标志只有超级用户才能修改（UNIX）。

标　志	意　义	标　志	意　义
stat.UF_NODUMP	不转储文件	stat.SF_ARCHIVED	文件可以存档
stat.UF_IMMUTABLE	文件是只读的	stat.SF_IMMUTABLE	文件是只读的
stat.UF_APPEND	文件只支持附加操作	stat.SF_APPEND	文件只支持附加操作
stat.UF_OPAQUE	目录是不透明的	stat.SF_NOUNLINK	文件不能被删除或重命名
stat.UF_NOUNLINK	文件不能被删除或重命名	stat.SF_SNAPSHOT	文件是一个快照文件

```
chmod(path, mode
)
```

修改 *path* 的模式。 *mode* 的值与open() 函数中相同（UNIX和Windows）。

```
chown(path, uid, gid
)
```

将路径的所有者和组ID修改为 *uid* 和 *gid* ，二者均为数字。将 *uid* 或 *gid* 设为-1，该参数将保持不变（UNIX）。

```
lchflags(path, flags
)
```

同chflags() 函数，但不获取符号链接（UNIX）。

```
lchmod(path, mode
)
```

同chmod() 函数，但当路径是符号链接时，它将修改链接本身，而不是链接引用的文件。

```
lchown(path, uid, gid
)
```

同chown() 函数，但不获取符号链接（UNIX）。

```
link(src, dst
```

```
)
```

创建指向 *src* 的名为 *dst* 的硬链接（UNIX）。

```
listdir(path
)
```

返回包含目录路径中各项名称的列表。该列表以任意次序返回，而且不包括特殊项'.'和'..'。如果路径是Unicode编码的，结果列表将只包含Unicode字符串。要清楚，如果目录中的任意文件不能被正确地编码为Unicode，那么它们将被悄悄地跳过。如果路径是字节字符串，那么所有文件名都以字节字符串列表的形式返回。

```
lstat(path
)
```

很像stat()函数，但不获取符号链接（UNIX）。

```
makedev(major, minor
)
```

给定主要和次要的设备编号，创建原始的设备编号（UNIX）。

```
major(devicenum
)
```

返回由makedev() 创建的原始设备编号 *devicenum* 中的主要设备编号。

```
minor(devicenum
)
```

返回由makedev() 创建的一个原始设备编号 *devicenum* 的次要设备编号。

```
makedirs(path
 [, mode
])
```

递归的目录创建函数。很像mkdir() 函数，但会创建包含叶子目录所需要的中间级目录。如果叶子目录已经存在或者无法创建，将引发OSError 异常。

```
mkdir(path
 [, mode
])
```

创建模式为数字 *mode* 的目录。默认模式是0777 。在非UNIX系统上，模式设置可能没有效果或被忽略。

```
mkfifo(path
 [, mode
])
```

创建模式为数字 *mode* 的FIFO（一个命名管道）。默认模式是0666 （UNIX）。

```
mknod(path
 [, mode
, device
])
```

创建设备特殊文件。 *path* 是文件的名称， *mode* 用于指定文件的权限和类型，而 *device* 是使用os.makedev() 函数创建的原始设备编号。设置文件的访问权限时，*mode* 参数接受的参数与open() 函数相同。另外，还给 *mode* 添加了标志 stat.S_IFREG 、stat.S_IFCHR 、stat.S_IFBLK 和stat.S_IFIFO ，用于指示文件类型（UNIX）。

```
pathconf(path, name
)
```

返回与路径名称 *path* 相关的可配置系统参数。 *name* 是一个指定参数的字符串，与fpathconf() 函数中相同（UNIX）。

```
readlink(path
)
```

返回一个字符串，代表符号链接 *path* 指向的路径（UNIX）。

```
remove(path
)
```

删除文件路径。这与unlink() 函数完全相同。

```
removedirs(path
)
```

递归的目录删除函数。工作方式很像rmdir() 函数，但如果成功删除叶子目录，对应于最右边路径部分的目录将被删除，直到整条路径消失或错误出现（这将被忽略，因为它通常表示父路径不为空）。如果无法成功删除叶子目录，就会引发OSError 异常。

```
rename(src, dst
)
```

将文件或目录 *src* 重命名为 *dst* 。

```
renames(old, new
)
```

递归的目录重命名或文件重命名函数。工作方式很像rename() 函数，但它首先会尝试创建命名新路径所需的中间目录。重命名之后，将使用removedirs() 函数删除对应于旧名称最右边路径部分的目录。

```
rmdir(path
)
```

删除目录路径。

```
stat(path
)
```

在指定路径上执行一次stat() 系统调用，以提取关于某个文件的信息。返回值是一个对象，其属性包含文件信息。常用属性如下所示。

属　　性	描　　述	属　　性	描　　述
st_mode	Inode保护模式	st_gid	所有者的组ID
st_ino	Inode编号	st_size	文件大小，单位为字节
st_dev	Inode所在的设备	st_atime	最后一次访问的时间
st_nlink	指向Inode链接的编号	st_mtime	最后一次修改的时间
st_uid	所有者的用户ID	st_ctime	最后一次状态变化的时间

但根据系统的具体情况，还可能存在另外的可用属性。stat() 函数返回的对象是包含10个参数的元组(st_mode, st_ino, st_dev, st_nlink, st_uid, st_gid, st_size, st_atime, st_mtime, st_ctime)。提供后一种形式是为了向后兼容性。stat 模块定义了可用于从这个元组中提取字段的　　常量。

```
stat_float_times([newvalue

])
```

如果stat() 函数返回的次数是浮点数而不是整数，返回True 。为 *newvalue* 提供一个布尔值可以改变这种方式。

```
statvfs(path

)
```

在指定路径上执行一次statvfs() 系统调用，获取文件系统的相关信息。返回值是一个对象，其属性可以描述文件系统。常用的属性包括。

属　　性	描　　述	属　　性	描　　述
f_bsize	首选的系统块大小	f_files	文件Inode的总数
f_frsize	基础的文件系统块大小	f_ffree	空闲文件Inode的总数
f_blocks	文件系统中块的总数	f_favail	非超级用户可用的空闲节点
f_bfree	空闲块的总数	f_flag	标志（依赖于系统）
f_bavail	非超级用户可用的空闲块	f_namemax	文件名的最大长度

返回对象的行为也类似于一个元组，其中包含上面列出的这些属性。标准模块statvfs 定义了可用于从返回的statvfs 数据中提取信息的常量（UNIX）。

```
symlink(src, dst
)
```

创建指向 *src* 的名为 *dst* 的符号链接。

```
unlink(path
)
```

删除文件 *path* 。同remove() 函数。

```
utime(path
, (atime, mtime
))
```

将文件的访问和修改时间设为指定值（第二个参数是一个包含两个元素的元组）。时间参数使用time.time() 函数返回的数字来指定。

```
walk(top [, topdown
 [, onerror
 [,followlinks
]]])
```

创建一个生成器对象来遍历整棵目录树。 *top* 指定目录的顶级，而 *topdown* 是一个布尔值，用于指示是由上而下（默认值）还是由下而上来遍历目录。返回的生成器将生成元组（ *dirpath* , *dirnames* , *filenames* ），其中dirpath 是一个字符串，包含通向目录的路径， *dirnames* 是 *dirpath* 中所有子目录的一个列表，而 *filenames* 是 *dirpath* 中文件的一个列表，不包括目录。 *onerror* 参数是一个接受单个参数的函数。如果处理期间出现任何错误，将使用os.error 的实例来调用此函数。默认的行为是忽略错误。如果由上而下地遍历目录，修改 *dirnames* 将影响到遍历过程。例如，如果从 *dirnames* 中删除目录，这些目录将被跳过。默认不会获取符号链接，除非将 *followlinks* 参数设为True 。

19.11.4 进程管理

以下函数和变量用于创建、销毁和管理进程。

```
abort()
```

生成发送给调用进程的SIGABRT 信号。除非使用处理器捕捉信号，否则进程默认将以错误终止。

```
defpath
```

如果环境没有'PATH' 变量，那么此变量包含exec*p*() 函数使用的默认搜索路径。

```
execl(path, arg0, arg1
```

```
, ...)
```

等价于execv(*path, (arg0, arg1 , ...)*) 函数。

```
execle(path, arg0, arg1, ..., env
)
```

等价于execve(*path , (arg0 , arg1 , ...), env*) 函数。

```
execlp(path, arg0, arg1
, ...)
```

等价于execvp(*path , (arg0 , arg1 , ...)*) 函数。

```
execv(path, args
)
```

使用参数列表 *args* 执行程序 *path* ， 替换当前进程（即Python解释器）。参数列表可以是元组或字符串列表。.

```
execve(path, args, env
)
```

执行一个很像execv() 的新程序，但另外还接受一个字典 *env* ，这个字典定义了程序的运行环境。 *env* 必须是将字符串映射为字符串的字典。

```
execvp(path, args
)
```

很像execv(*path* , *args*)函数，但在目录列表中搜索可执行文件的过程中将复制shell的动作。字典列表从environ['PATH']获得。

```
execvpe(path, args, env
)
```

很像execvp()，但像在execve()函数中一样，带有一个另外的环境变量。

```
_exit(n
)
```

使用状态 *n* 立即退回到系统，同时不执行任何清理动作。这通常只在由fork()函数创建的子进程中完成。这与调用sys.exit()函数也不同，sys.exit()函数会正常关闭解释器。退出代码 *n* 依赖于应用程序，但一般0值表示成功，而非0值表示出现某种错误。根据系统的不同，可以定义大量标准的退出代码值。

值	描　　述	值	描　　述
EX_OK	没有错误	EX_OSERR	操作系统错误
EX_USAGE	错误的命令用法	EX_OSFILE	文件系统错误
EX_DATAERR	错误的输入数据	EX_CANTCREAT	无法创建输出
EX_NOINPUT	缺少输入	EX_IOERR	I/O错误
EX_NOUSER	用户不存在	EX_TEMPFAIL	临时故障

EX_NOHOST	主机不存在	EX_PROTOCOL	协议错误
EX_NOTFOUND	未找到	EX_NOPERM	权限不足
EX_UNAVAILABLE	服务不可用	EX_CONFIG	配置错误
EX_SOFTWARE	内部软件错误		

fork()

创建一个子进程。在新创建的子进程中返回0，在原始进程中返回子进程的进程ID。子进程是原始进程的克隆，它们共享众多资源，如打开的文件（UNIX）。

forkpty()

创建一个子进程，并使用一台新的伪终端作为子进程的控制终端。返回值为一对 (*pid* , *fd*)，其中 *pid* 在子进程中为0，而 *fd* 是伪终端主端的文件描述符。此函数只能在某些版本的UNIX上使用。

kill(*pid, sig*

)

发送信号 *sig* 到进程 *pid* 。在 *signal* 模块中可以找到信号名称的列表（UNIX）。

killpg(*pgid, sig*

)

发送信号 *sig* 到进程组 *pgid* 。在 *signal* 模块中可以找到信号名称的列表（UNIX）。

```
nice(increment
)
```

提高进程的调度优先级（"niceness"）。返回新的niceness 。用户一般只能降低进程的优先级，因为提高优先级需要root 访问权限。修改优先级的效果与系统有关，但是降低优先级通常会让进程在后台运行，使其不会明显影响到其他进程的性能（UNIX）。

```
plock(op
)
```

将程序段锁定在内存中，从而防止它们被交换。 *op* 的值是一个整数，用于确定要锁定哪些段。 *op* 的值是特定于平台的，但一般是UNLOCK 、PROCLOCK 、TXTLOCK 或DATLOCK 之一。这些常量并非由Python定义，但可以在<sys/lock.h> 头文件中找到。此函数不能用在所有平台上，而且通常只能由有效用户ID为0（具有root 权限）的进程执行（UNIX）。

```
popen(command
 [, mode
 [, bufsize
]])
```

给命令打开一条管道。返回值是连接到管道的打开文件对象，根据模式是'r' （默认值）还是'w' ，读取或写入这个文件对象。 *bufsize* 的意义与内置函数open() 相同。命令的退出状态由返回的文件对象的close() 方法返回，但如果退出状态为0，则返回None 。

```
```

```
spawnv(mode
, path
, args
)
```

在新进程中执行程序，以命令行参数的形式传递 *args* 中指定的参数。 *args* 可以是列表或元组。 *args* 的第一个元素应该是程序的名称。 *mode* 可取以下常量值。

常　　量	描　　述
P_WAIT	执行程序并等待它终止。返回程序的退出代码
P_NOWAIT	执行程序并返回进程句柄
P_NOWAITO	同P_NOWAIT
P_OVERLAY	执行程序并销毁调用进程（同exec 函数）
P_DETACH	执行程序并从此程序分离。调用程序继续运行，但不会等待进程中创建的进程

spawnv() 函数在Windows和一些版本的UNIX上可用。

```
spawnve(mode, path, args, env
)
```

在新进程中执行程序，以命令行参数的形式传递 *args* 中指定的参数并以环境的形式传递映射env 中的内容。 *args* 可以是列表或元组。 *mode* 的意义与spawnv() 函数中相同。

```
spawnl(mode, path, arg1, ..., argn
)
```

同spawnv() 函数，但所有参数均以额外参数的形式提供。

```
spawnle(mode, path, arg1, ... , argn, env
)
```

同spawnve() 函数，但参数以形参形式提供。最后一个参数是包含环境变量的映射。

```
spawnlp(mode, file, arg1, ... , argn
)
```

同spawnl() 函数，但会使用PATH 环境变量的设置查找 *file* （UNIX）。

```
spawnlpe(mode, file, arg1, ... , argn, env
)
```

同spawnle() 函数，但会使用PATH 环境变量的设置查找 *file* （UNIX）。

```
spawnvp(mode, file, args
)
```

同spawnv() 函数，但会使用PATH 环境变量的设置查找 *file* （UNIX）。

```
spawnvpe(mode, file, args, env
)
```

同spawnve() 函数，但会使用PATH 环境变量的设置查找 *file* （UNIX）。

```
startfile(path
 [, operation
])
```

运行与文件 *path* 相关的应用程序。这样执行的动作与在Windows Explorer中双击文件相同。在应用程序运行之后，函数就会返回。此外不能等待完成或者从应用程序获得退出代码。 *path* 的值是相对于当前目录而言。 *operation* 是一个可选的字符串，用于指定打开 *path* 时执行的动作。它的默认值是'open'，但也可以将它置为'print'、'edit'、'explore' 或'find' ［确切的列表跟 *path* 的类型有关（Windows）］。

```
system(command
)
```

在子shell中执行 *command* （字符串）。在UNIX上，返回值和wait() 函数一样是进程的退出状态。在Windows上，退出代码始终为0。subprocess 模块提供的功能更加强大，是运行子进程的首选方式。

```
times()
```

返回一个包含5个浮点数元素的元组，指示累计的时间，单位为秒。在UNIX上，该元组包含用户时间、系统时间、子进程的用户时间、子进程的系统时间，以及按照上述顺序排列的实际用时。在Windows上，该元组包含用户时间、系统时间，其他三个值均为0。

```
wait([pid
])
```

等待一个子进程完成并返回包含其进程ID和退出状态的元组。退出状态是一个16位数字，其低位字节是终止进程的信号编号，而高位字节则是退出状态（如果信号编号为0）。如果生成了核心文件，就会设置低位字节的高位。 *pid* 参数指定了要等待的进程。如果省略这个参数，当任意子进程退出时，`wait()` 函数就会返回（UNIX）。

```
waitpid(pid, options
)
```

等待进程ID为 *pid* 的子进程的状态出现变化，并返回包含其进程ID和退出状态指示的一个元组，编码与`wait()` 函数中相同。 *options* 对于常规操作应该为0或`WNOHANG`，以避免当没有子进程状态可用时出现挂起。此函数还可以用于收集只为某些原因而停止执行的子进程的相关信息。将 *options* 设置为`WCONTINUED` 时，可以从一个停止之后通过任务控制恢复操作的子进程收集信息。将 *options* 设置为`WUNTRACED` 时，可以从一个已经停止，但尚未报告状态信息的子进程收集信息。

```
wait3([options
])
```

同`waitpid()` 函数，但此函数将等待所有子进程中的变化。返回包含3个元素的元组 (*pid* , *status* , *rusage*)，其中 *pid* 是子进程ID， *status* 是退出状态代码，而 *rusage* 包含使用`resource.getrusage()` 函数返回的资源使用信息。 *options* 参数的意义与`waitpid()` 函数中的相同。

```
wait4(pid, options
)
```

同`waitpid()` 函数，但返回的元组与`wait3()` 返回的相同。

以下函数使用`waitpid()`、`wait3()` 或`wait4()` 函数返回的进程状态代码作为参数，用于检查进程的状态（UNIX）。

```
WCOREDUMP(status
)
```

如果进程转储核心，返回True 。

```
WIFEXITED(status
)
```

如果进程使用exit() 系统调用退出，返回True 。

```
WEXITSTATUS(status
)
```

如果WIFEXITED(*status*) 为True ，就返回exit() 系统调用的整数参数，否则返回值毫无意义。

```
WIFCONTINUED(status
)
```

如果进程已经从任务控制停止中恢复，返回True 。

```
WIFSIGNALED(status
)
```

如果进程由于信号而退出，返回True 。

```
WIFSTOPPED(status
)
```

如果进程已经停止，返回True 。

```
WSTOPSIG(status
)
```

返回导致进程停止的信号。

```
WTERMSIG(status
)
```

返回导致进程退出的信号。

19.11.5 系统配置

以下函数用于获得系统配置信息。

```
confstr(name
)
```

返回字符串类型的系统配置变量。 *name* 是指定变量名称的字符串。可接受的名称是特定于平台的，但在os.confstr_names 中可以找到主机系统中已知名称的字典。如果某个指定名称的配置值未定义，则返回空的字符串。如果 *name* 未知，就会引发ValueError 异常。如果主机系统不支持配置名称，还会引发OSError 异常。此函数返回的参数属于主机计算机上的编译环境，包括系统实用工具的路径，各种程序配置的编译器

选项（如32位、64位和大型文件支持），以及连接器选项（UNIX）。

```
getloadavg()
```

返回一个3个元素的元组，其中包含过去1分钟、5分钟和15分钟内系统运行队列中的平均项数（UNIX）。

```
sysconf(name
)
```

返回一个整数值的系统配置变量。 *name* 是指定变量名称的字符串。主机系统上定义的名称可在字典os.sysconf_names 中找到。如果配置名称是已知的，但其值未定义，那么将返回-1，否则将引发ValueError 或OSError 异常。有些系统定义的系统参数可能超过100个。下面的列表详细列出了POSIX.1定义的参数，这些参数应该在大多数UNIX系统上可用。

参　　数	描　　述
"SC_ARG_MAX"	exec() 函数使用参数的最大长度
"SC_CHILD_MAX"	每个用户的最大进程数
"SC_CLK_TCK"	每秒时钟走的次数
"SC_NGROUPS_MAX"	同步补充组ID的最大数量
"SC_STREAM_MAX"	进程一次能打开的最大流数量
"SC_TZNAME_MAX"	时区名称中的最大字节数
"SC_OPEN_MAX"	进程一次能打开的最大文件数量
"SC_JOB_CONTROL"	系统支持任务控制

"SC_SAVED_IDS"	指示每个进程是否有已经保存的set-user-ID和已经保存的set-group-ID

```
urandom(n)
```

返回一个包含由系统生成的n 个随机字节的字符串（例如，在UNIX上是/dev/urandom）。返回的字节适合用于加密。

19.11.6 异常

os 模块定义了一个异常来表示错误。

```
error
```

当函数返回系统相关错误时引发的异常。这与内置的异常OSError 相同。异常带有两个值：errno 和strerr 。第一个值包含的整数错误值与errno 模块中的相同。后一个值包含一条字符串错误消息。对于涉及文件系统的异常，它们还包含第三个属性filename ，它是传递给函数的文件名。

19.12 os.path

os.path 模块用于以一种可移植的方式操作路径名称。它由os 模块导入。

```
abspath(path
)
```

返回路径名称 *path* 的绝对路径，同时将当前的工作目录考虑在内。例如，abspath('../Python /foo') 的返回值是'/home/beazley/Python/foo' 。

```
basename(path
)
```

返回路径名称 *path* 的基本名称。例如，basename('/usr/local/python') 的返回值是'python' 。

commonprefix(*list*

)

返回 *list* 中所有字符串的前缀中最长的字符串。如果 *list* 为空，将返回空字符串。

dirname(*path*

)

返回路径名称 *path* 的目录名称。例如，dirname('/usr/local/python') 的返回值是'/usr/ local' 。

exists(*path*

)

如果 *path* 引用的是现有路径，返回True 。如果 *path* 引用的是已经损坏的符号链接，返回False 。

expanduser(*path*

)

使用用户的主目录替换'~user' 格式的路径名称。如果扩展失败或者 *path* 不以'~' 开头，将原封不动地返回路径。

expandvars(*path*

)

扩展 *path* 中'$name' 或'${name}' 格式的环境变量。不符合规范或不存在的变量名称将保持不变。

getatime(*path*

)

返回最后一次访问 *path* 的时间，返回值是从纪元开始的秒数（参见time 模块）。如果os.stat_float_times() 函数返回True ，结果是一个浮点数。

getctime(*path*

)

在UNIX上返回的是最后一次修改 *path* 的时间，而在Windows上返回的是创建 *path* 的时间。返回值是从纪元开始的秒数（参见time 模块）。在某些情况下，返回值是浮点数（参见getatime() 函数）。

getmtime(*path*

)

返回最后一次修改 *path* 的时间，返回值是从纪元开始的秒数（参见time 模块）。在某些情况下，返回值是浮点数（参见getatime() 函数）。

getsize(*path*

)

返回 *path* 的大小，以字节为单位。

isabs(*path*

)

如果 *path* 是绝对路径名称（以一个斜杠开头），返回True 。

isfile(*path*

)

如果path 是普通文件，返回True 。此函数效仿符号链接，因此islink() 和 isfile() 对于相同路径均返回True 。

isdir(*path*

)

如果 *path* 是目录，返回True 。此函数仿效符号链接。

islink(*path*

)

如果 *path* 引用的是符号链接，返回True 。如果不支持符号链接则返回False 。

ismount(*path*

```
)
```

如果 *path* 是挂载点，返回True 。

```
join(path1
 [, path2
 [, ...]])
```

将一个或多个路径组件智能地连接为一个路径名称。例如，join('home', 'beazley', 'Python') 的返回值是'home/beazley/Python' 。

```
lexists(path
)
```

如果 *path* 存在，返回True 。对所有符号链接均返回True ，即便链接已经损坏。

```
normcase(path
)
```

标准化路径名称的大小写。在不区分大小写的文件系统上，它把路径转换为小写字母。在Windows上，它把正斜杠转换为反斜杠。

```
normpath(path
)
```

标准化路径名称。它将折叠多余（或冗长的）分隔符和上层引用，它将'A//B'、'A/./B'和'A/foo/../B'都变为'A/B'。在Windows上，它把正斜杠转换为反斜杠。

```
realpath(path
)
```

返回 *path* 的真实路径，并除去路径中的所有符号链接（如果有的话）（UNIX）。

```
relpath(path
 [, start
])
```

返回从当前工作目录到 *path* 的一条相对路径。可以提供 *start* 参数来指定另一个起始目录。

```
samefile(path1
, path2
)
```

如果 *path1* 和 *path2* 引用同一个文件或目录，返回True （UNIX）。

```
sameopenfile(fp1
, fp2
)
```

如果打开的文件对象 *fp1* 和 *fp2* 引用同一个文件，返回True （UNIX）。

```
samestat(stat1
, stat2
)
```

如果fstat()、lstat()或stat()返回的stat元组stat1和stat2引用同一个文件，则返回True（UNIX）。

```
split(path
)
```

将 *path* 拆分为(*head* , *tail*)对，其中 *tail* 是最后一个路径名称组件，而head是 *tail* 之前的内容。例如，'/home/user/foo'将被分割为('/home/user', 'foo')。这个元组与(dirname(), basename())返回的元组相同。

```
splitdrive(path
)
```

将path拆分为(*drive* , *filename*)对，其中 *drive* 是驱动器说明或空字符串。在没有驱动器说明的系统上， *drive* 始终是空字符串。

```
splitext(path
)
```

将路径名称拆分为基本文件名称和后缀。例如，splitext('foo.txt')的返回值为('foo', '.txt')。

```
splitunc(path
```

```
)
```

将路径名称拆分为(*unc* ,*rest*)对，其中 *unc* 是UNC（Universal Naming Convention，通用命名规则）挂载点，而 *rest* 是路径的余下部分（Windows）。

```
supports_unicode_filenames
```

如果文件系统支持Unicode文件名称，那么此变量将被设置为`True` 。

> 注意
>
> 在Windows上，处理包含驱动器字母的文件名称时（如`'C:spam.txt'`），需要特别小心。大多数情况下，会根据当前工作目录解释文件名称。例如，如果当前目录是`'C:\Foo\'`，那么文件将被解释为`'C:\Foo\C:spam.txt'`，而非`'C:\spam.txt'` 。

另请参见： 18.3节、18.4节和19.11节的内容。

19.13 `signal`

`signal` 模块在Python中用于编写信号处理器。信号通常对应异步事件，包括定时器到期、有输入数据到达或者用户执行的一些操作。信号接口仿效UNIX，但其他平台也支持此模块的某些部分。

```
alarm(time

)
```

如果 *time* 非0，就会安排在 *time* 秒内将`SIGALRM` 信号发送给程序。以前安排的任何警报都会被取消。如果 *time* 为0，不会安排任何警报，而且会取消以前设置的所有警报。返回值为以前安排的所有警报之前所余的秒数，如果没有安排警报则返回0（UNIX）。

```
getsignal(signalnum

)
```

返回信号 *signalnum* 的信号处理器。返回对象是可调用的Python对象。此函数还会为被忽略的信号返回SIG_IGN，为默认处理器返回SIG_DFL，如果在Python解释器中没有安装信号处理器则返回None。

```
getitimer(which

)
```

返回由 *which* 表示的内部定时器的当前值。

```
pause()
```

进入睡眠状态，直到收到下一个信号为止（UNIX）。

```
set_wakeup_fd(fd

)
```

设置文件描述符 *fd*，当接收到信号时会在它上面写入一个'\0'字节。接着使用该信号，通过类似select模块中的函数来处理轮询文件描述符的程序中的信号。必须以非阻塞模式打开 *fd* 描述的文件，此函数才会有效。

```
setitimer(which

, seconds

 [, interval

])
```

将内部定时器设为在 *seconds* 秒之后生成一个信号，并在此之后每 *interval* 秒重复发出信号。这两个参数均被指定为浮点数。 *which* 参数是ITIMER_REAL、ITIMER_VIRTUAL 或ITIMER_PROF 。 *which* 的选择决定了在定时器到期后生成什么信号，如果是ITIMER_REAL 则生成SIGALRM，是ITIMER_VIRTUAL 则生成SIGVTALRM，是ITIMER_PROF 生成SIGPROF 。将 *seconds* 置为0将清除定时器。此函数使用定时器的以前设置返回元组(*seconds, interval*)。

```
siginterrupt(signalnum
, flag
)
```

为给定的信号编号设置系统调用重启行为。如果 *flag* 为False，被信号 *signalnum* 中断的系统调用将自动重新启动。如果 *flag* 为True，将中断系统调用。被中断的系统调用通常会导致OSError 或IOError 异常，这里的相关错误编号设为errno.EINTR 或errno.EAGAIN 。

```
signal(signalnum
, handler
)
```

将信号 *signalnum* 的信号处理器设为函数 *handler* 。 *handler* 必须是接受两个参数（信号编号和帧对象）的可调用Python对象。将 *handler* 指定为SIG_IGN 或SIG_DFL 分别代表忽略信号和使用默认的信号处理器。返回值是以前的信号处理器SIG_IGN 或SIG_DFL 。启用线程时，只能从主线程调用此函数，否则将引发ValueError 异常。

使用SIG* 格式的符号内容可以识别各种信号。这些名称对应于特定于计算机的整数值。典型的值包括以下这些。

信号名称	描　　述	信号名称	描　　述
SIGABRT	异常终止	SIGPWR	电源故障
SIGALRM	警报	SIGQUIT	终端退出字符

SIGBUS	总线错误	SIGSEGV	段错误
SIGCHLD	子状态中的变化	SIGSTOP	停止
SIGCLD	子状态中的变化	SIGTERM	终止
SIGCONT	继续	SIGTRAP	硬件故障
SIGFPE	浮点错误	SIGTSTP	终端停止字符
SIGHUP	挂起	SIGTTIN	控制TTY
SIGILL	非法指令	SIGTTOU	控制TTY
SIGINT	终端中断字符	SIGURG	紧急情况
SIGIO	异步I/O	SIGUSR1	用户定义
SIGIOT	硬件错误	SIGUSR2	用户定义
SIGKILL	终止	SIGVTALRM	虚拟时间警报
SIGPIPE	写入管道而非读取器	SIGWINCH	Window大小变化
SIGPOLL	可轮询的事件	SIGXCPU	超出CPU限制
SIGPROF	配置警报	SIGXFSZ	超出文件大小限制

另外，该模块还定义了以下变量。

变　　量	描　　述
SIG_DFL	调用默认信号处理器的信号处理器
SIG_IGN	忽略信号的信号处理器
NSIG	比最大的信号编号大一号

19.13.1 例子

下面的例子演示了建立网络连接时出现超时的情况（socket 模块已经提供了超时选项，因此这个例子只是说明使用signal 模块的基本概念）。

```
import signal, socket
def handler(signum, frame):
    print 'Timeout!'
    raise IOError, 'Host not responding.'
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)                     # 5秒后发生警报
sock.connect('www.python.org', 80) # 连接
signal.alarm(0)                     # 清除警报
```

19.13.2 注意

- 信号处理器保持已安装状态，直到被显式地重置，除了SIGCHLD 信号（它的行为是特定于实现的）。
- 不能暂时禁用信号。
- 只能在Python解释器的原子指令之间处理信号。使用C语言编写的长时间计算将延迟信号传输（因为扩展模块可能会执行这样的操作）。
- 如果信号出现在一次I/O操作中，这次I/O操作可能会出现异常而最终失败。在这种情况下，errno 值设为errno.EINTR ，用于指示一次被中断的系统调用。
- 某些信号无法在Python中处理，如SIGSEGV 。
- Python默认安装了少数信号处理器。SIGPIPE 被忽略，ISGINT 被转换为KeyboardInterrupt 异常，而ISGTERM 则被捕捉，以便进行清理和调用sys.exitfunc 。
- 如果在一个程序中同时使用信号和线程，需要特别当心。目前，只有执行的主线程能够设置新的信号处理器或者接收信号。
- Windows上的信号处理功能有限。在这个平台上，受支持的信号数量十分有限。

19.14 subprocess

subprocess 模块包含的函数和对象用于泛化创建新进程的任务、控制输入与输出流，以及处理返回代码。此模块集纳了各个其他模块中的许多功能，如os 、popen2 和commands 。

```
Popen(args

, **parms

)
```

以子进程形式执行一个新命令，然后返回代表新进程的Popen 对象。命令在 *args* 中指定，是字符串（如'ls -l'）或字符串列表（如['ls', '-l']）。 *parms* 表示关键字参数的集合，设置这些参数可以控制子进程的各种属性。需要了解的关键字参数如下。

关 键 字	描 述
bufsize	指定缓存行为，0代表无缓存，1代表行缓存，负值代表使用系统默认行为，而其他正数值用于指定近似的缓存大小。默认值为0
close_fds	如果为True，在子进程执行之前将关闭除0、1和2之外的所有文件描述符。默认值为False
creation_flags	指定Windows上的进程创建标志。目前唯一可用的标志是CREATE_NEW_CONSOLE。默认值为0
cwd	将在其中执行命令的目录。执行前将子进程的当前目录修改为cwd。默认值为None，表示使用父进程的当前目录
env	新进程的环境变量的字典。默认值为None，表示使用父进程的环境变量
executable	指定要使用可执行程序的名称。很少需要用到这个关键字，因为程序名称已经包含在了 *args* 中。如果指定了shell，此参数则用于指定要使用shell 的名称。默认值为None
preexec_fn	指定刚好在执行命令之前，子进程中要调用的函数。此函数应该不带任何参数
shell	如果为True，将使用像os.system() 函数这样的UNIX shell来执行命令。默认的shell是/bin/sh，但也可以通过设置executable 来修改它。shell的默认值为None
startupinfo	提供在Windows上创建进程时使用的启动标志。默认值为None。可能的值包括STARTF_USESHOWWINDOW 和STARTF_USESTDHANDLERS
stderr	这个文件对象代表stderr 在子进程中使用的文件。可能是通过open() 函数创建的文件对象，整数的文件描述符，或者特殊值PIPE，它表示应该创建一条新的管道。默认值为None
stdin	这个文件对象代表stdin 在子进程中使用的文件。可以设置为与stderr 相同的值。默认值为None
stdout	这个文件对象代表stdout 在子进程中使用的文件。可以设置为与stderr 相同的值。默认值为None
universal_newlines	如果为True，将在启用通用换行模式的情况下使用文本模式打开表示stdin 、stdout 和

	stderr 的文件。完整的描述参见open() 函数

```
call(args
, **parms
)
```

此函数与Popen() 函数完全相同，但它只会简单地执行命令然后返回它的状态代码（也就是说，它不会返回Popen 对象）。如果要执行一个命令，但又不需要捕捉它的输出或者以其他方式控制它，可以使用这个函数。参数的意义与Popen() 函数中相同。

```
check_call(args
, **parms
)
```

同call() 函数，但如果退出代码为非0值，将引发CalledProcessError 异常。此异常将退出代码保存在它的returncode 属性中。

Popen() 函数返回的Popen 对象 *p* 具有各种方法和属性，可用于与子进程进行交互。

```
p.communicate([input
])
```

通过将input 中提供的数据发送给进程的标准输入，与子进程进行通信。数据一旦发出，方法就会等待进程终止，同时收集标准输出上接收到的输出和标准错误。返回值是一个元组(*stdout* , *stderr*)，其中 *stdout* 和 *stderr* 是字符串。如果没有给子进程发送数据， *input* 将被设置为None （默认值）。

```
p.kill()
```

终止子进程，具体方法在UNIX上是发送给子进程一个SIGKILL 信号，而在Windows上是调用 *p* .terminate() 方法。

```
p
.poll()
```

检查 *p* 是否已经终止。如果已经终止，则返回子进程的返回代码，否则返回None。

```
p
.send_signal(signal
)
```

发送一个信号给子进程。 *signal* 是 *signal* 模块中定义的信号编号。在Windows上，唯一支持的信号是SIGTERM 。

```
p
.terminate()
```

终止子进程，具体方法在UNIX上是发送给子进程一个SIGTERM 信号，而在Windows上是调用Win32 API TerminateProcess 函数。

```
p
.wait()
```

等待 *p* 终止，然后返回返回代码。

```
p
.pid
```

子进程的进程ID。

```
p
.returncode
```

进程的数字返回代码。如果为None ， 表示进程尚未终止。如果为负值，表示进程已经被一个信号所终止（UNIX）。

```
p
.stdin, p
.stdout, p
.stderr
```

当以管道形式打开相应的I/O流时，这三个属性将被设为打开文件对象。例如，将Popen() 函数中的 *stdout* 参数设置为PIPE 。提供这些文件对象的目的是将管道与其他子进程连接。如果不使用管道，这些属性将设置为None 。

19.14.1 例子

```
# 执行基本的系统命令，如os.system()
ret = subprocess.call("ls -l", shell=True)
```

```
# 静默执行基本的系统命令
ret = subprocess.call("rm –f *.java",shell=True,
                      stdout=open("/dev/null"))
# 执行系统命令，但是捕捉输出
p = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
out = p.stdout.read()

# 执行命令，但是发送输入和接收输出
p = subprocess.Popen("wc", shell=True, stdin=subprocess.PIPE,
                     stdout=subprocess.PIPE, stderr=subprocess.PIPE)
out, err = p.communicate(s)    # 发送字符串给进程

# 创建两个子进程，然后通过一条管道将它们链接在一起
p1 = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
p2 = subprocess.Popen("wc",shell=True, stdin=p1.stdout,
                      stdout=subprocess.PIPE)
out = p2.stdout.read()
```

19.14.2 注意

- 一般而言，最好以字符串列表的形式提供命令行，而不要使用带有shell命令的单个字符串（例如，使用`['wc','filename']`而不要使用`'wc filename'`）。在很多系统上，文件名称经常包含有趣字母和空格（如Windows上的“Documents and Settings”文件夹）。如果以列表形式提供命令参数，将一切正常。如果尝试使用shell命令，必须要采取另外的步骤来保证对特殊字符和空格进行正确的转义。
- Windows上将以二进制模式打开管道。因此，如果要从子进程读取文本输出，行结束将包含额外的回车字符（`'\r\n'`而非`'\n'`）。如果这样做出现问题，可以在使用`Popen()`函数时提供`universal_newlines`选项。
- 模块`subprocess`不能用于控制要运行在终端或TTY中的进程。最常见的例子是用户需要输入密码的程序，如`ssh`、`ftp`、`svn`等。要控制这些程序，需要使用基于常用“Expect”UNIX实用工具的第三方模块。

19.15 time

`time`模块提供与时间相关的各种函数。在Python中，测量时间的方法是计算从纪元开始的秒数。 *epoch* 是时间的开始（time=0秒的时候）。在UNIX上，纪元是1970年1月1日，而在其他系统上则由`time.gmtime(0)`决定。

此模块定义了以下变量。

accept2dyear

指示是否接受两位年份的布尔值。它通常为`True`，但如果环境变量`$PYTHONY2K`设为非空字符串，它将设置为`False`。也可以手动修改它的值。

altzone

夏令时期间使用的时区，如果可以使用的话。

daylight

如果定义了夏令时时区，它将被设为一个非0值。

timezone

本地的（非夏令时）时区。

tzname

包含本地时区和本地夏令时时区（如果定义了的话）的名称的元组。

可以使用以下函数。

**asctime([*tuple*
])**

将表示gmtime() 或localtime() 函数返回时间的元组转换为'Mon Jul 12 14:45:23 1999' 格式的字符串。如果不提供参数，就会使用当前时间。

clock()

返回当前的CPU时间，单位为秒。返回值是一个浮点数。

```
ctime([secs
])
```

将从纪元开始计算而且以秒数形式表示的时间转换为代表本地时间的字符串。ctime(*secs*) 与asctime(localtime(*secs*)) 相同。如果省略 *secs* 参数或者将它赋值为None ， 就会使用当前的时间。

```
gmtime([secs
])
```

将从纪元开始计算而且以秒数形式表示的时间转换为UTC协调世界时（Coordinated Universal Time），又名格林尼治标准时间（Greenwich Mean Time）。此函数返回的是一个struct_time 对象，它具有以下属性。

属　　性	值	属　　性	值
tm_year	一个4位数的值，如1998	tm_sec	0～61
tm_mon	1～12	tm_wday	0～6 （0=Monday）
tm_mday	1～31	tm_yday	1～366
tm_hour	0～23	tm_isdst	-1 、 0 、 1
tm_min	0～59		

如果夏令时有效，tm_isdst 属性的值为1 ， 如果无效则为0 ， 如果无可用信息则为-1 。为了实现向后兼容，返回的struct_time 对象的行为与包含按上面顺序列出的这

些属性的9元素元组相似。

```
localtime([secs
])
```

和gmtime() 函数一样，返回一个struct_time 对象，但对应于本地时区。如果省略 *secs* 参数或将其赋值为 *None* ，就会使用当前的时间。

```
mktime(tuple
)
```

此函数使用一个struct_time 对象或代表本地时区中某个时间的元组作为参数（格式同localtime() 函数），并返回一个浮点数，表示从纪元开始计算的秒数。如果输入值不是有效的时间，将引发OverflowError 异常。

```
sleep(secs
)
```

让当前进程进入睡眠状态并持续 *secs* 秒钟。 *secs* 是一个浮点数。

```
strftime(format
 [, tm
])
```

将代表gmtime() 或localtime() 返回时间的struct_time 对象转换为字符串（为了实现向后兼容性， *tm* 也可能是表示时间值的元组）。 *format* 是格式字符串，可在其中嵌入以下格式代码。

指　令	意　义
%a	地区缩写形式的星期几名称
%A	地区完整形式的星期几名称
%b	地区缩写形式的月份名称
%B	地区完整形式的月份名称
%c	地区正确日期与时间表示
%d	一月中的一天，范围在[01-31] 的十进制数
%H	小时（24小时的时钟），范围在[00-23] 的十进制数
%I	小时(12小时的时钟)，范围在[01-12] 的十进制数
%j	一年中的一天，范围在[001-366] 的十进制数
%m	月份，范围在[01-12] 的十进制数
%M	分钟，范围在[00-59] 的十进制数
%p	地区的上午或下午
%S	秒，范围在[00-61] 的十进制数
%U	一年的周数，范围在[00-53] （星期日为首日）
%w	一周中的一天，范围在[0-6] 的十进制数（0 = 星期日）
%W	一年的周数（星期一为首日）
%x	地区的正确日期表示
%X	地区的正确时间表示
%y	没有世纪部分的年，范围在[00-99] 的十进制数

%y	没有世纪部分的年，范围在[00-99]的十进制数
%Y	带有世纪部分的年，为十进制数
%z	时区名称（如果不存在时区则无字符）
%%	%字符

格式代码可以包含宽度和精度，这类似于字符串上的% 运算符。如果元组中的任意字段超出范围，将引发ValueError 异常。如果省略 *tuple* 参数，就会使用对应于当前时间的时间元组。

```
strptime(string
 [, format
])
```

解析一个代表时间的字符串，然后返回一个struct_time 对象，这与localtime() 或gmtime() 函数的返回值相同。 *format* 参数使用与strftime() 参数相同的说明符，默认为'%a %b %d %H:%M:%S %Y' 。这与ctime()函数生成的格式相同。如果不能解析字符串，就会引发ValueError 异常。

```
time()
```

返回当前时间，返回值为UTC协调世界时中自从纪元开始计算的秒数。

```
tzset()
```

基于UNIX上TZ环境变量的值重置时区设置。例如：

```
os.environ[‘TZ'] = ‘US/Mountain'
```

```
time.tzset()


os.environ['TZ'] = "CST+06CDT,M4.1.0,M10.5.0"

time.tzset()
```

注意

- 接受两位数的年份时，它们将根据POSIX X/Open标准被转换为四位数的年份，值69～99 被映射为1969～1999，而值0～68 映射为2000～2068。
- 时间函数的精度通常比建议表示时间的单位小得多。例如，操作系统在一秒钟内可能只更新时间50~100 次。

另请参见： 19.3节的内容。

19.16 winreg

winreg 模块（在Python 2中为_winreg）提供了Windows注册表的底层接口。注册表是一棵很大的层次结构树，其中的每个节点叫做键（key）。特定键的孩子叫做子键（subkey），可能包含另外的子键或值。例如，Python sys.path 变量的设置一般包含在注册表的以下位置：

```
\HKEY_LOCAL_MACHINE\Software\Python\PythonCore\2.6\PythonPath
```

在这个例子中，Software 是HKEY_LOCAL_MACHINE 的一个子键，Python 是Software 的一个子键，依此类推。PythonPath 键的值包含实际的路径设置。

访问键需要通过打开和关闭操作。打开键由特殊的句柄表示（Windows通常使用封装整数句柄标识符的封装器）。

```
CloseKey(key)
```

关闭一个以前使用句柄 *key* 打开的注册表键。

```
ConnectRegistry(computer_name, key
)
```

返回指向另一台计算机上预定义注册表键的句柄。computer_name 是远程计算机的名称，如字符串*computername* 。如果 *computer_name* 为None ，将使用本地的注册表。 *key* 是预定义的句柄，如HKEY_CURRENT_USER 或HKEY_USERS 。出错时将引发 EnvironmentError 异常。下面的列表显示了_winreg 模块中定义的所有HKEY_* 值：

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_CURRENT_USER
- HKEY_DYN_DATA
- HKEY_LOCAL_MACHINE
- HKEY_PERFORMANCE_DATA
- HKEY_USERS

```
CreateKey(key
, sub_key
)
```

创建或打开一个键，并返回一个句柄。 *key* 是以前打开的键或者是HKEY_* 常量定义的预定义键。 *sub_key* 是要打开或创建的键的名称。如果 *key* 是预定义键，那么 *sub_key* 可能为None ，这种情况下将返回 *key* 。

```
DeleteKey(key, sub_key
)
```

删除 *sub_key* 。 *key* 是打开的键或者是预定义的HKEY_* 常量之一。 *sub_key* 是用于识别要删除键的字符串。 *sub_key* 绝不能有任何子键，否则将引发 EnvironmentError 异常。

```
DeleteValue(key, value
```

```
)
```

从一个注册表键删除命名值。 *key* 是打开的键或者是预定义的HKEY_* 常量之一。 *value* 是包含要删除值的名称的字符串。

```
EnumKey(key, index
)
```

返回索引为 *index* 的子键的名称。 *key* 是一个打开的键或者是预定义的HKEY_* 常量之一。 *index* 用于指定要获取键的整数。如果 *index* 超出范围，将引发 EnvironmentError 异常。

```
EnumValue(key, index
)
```

返回打开键的值。 *key* 是打开的键或者是预定义的HKEY_* 常量之一。 *index* 用于指定要获取键的整数。此函数返回一个元组(*name* , *data* , *type*)，其中 *name* 是值的名称， *data* 是保存值数据的对象，而type 是指定值数据类型的整数。目前定义了以下类型代码。

代　　码	描　　述
REG_BINARY	二进制数据
REG_DWORD	32位数字
REG_DWORD_LITTLE_ENDIAN	小尾格式的32位数字
REG_DWORD_BIG_ENDIAN	大尾格式的32位数字
REG_EXPAND_SZ	对环境变量进行非扩展引用的以Null 结尾的字符串

REG_LINK	Unicode符号链接
REG_MULTI_SZ	以Null 结尾的字符串序列
REG_NONE	不存在已定义的值类型
REG_RESOURCE_LIST	设备驱动程序资源列表
REG_SZ	以Null 结尾的字符串

```
ExpandEnvironmentStrings(s
)
```

扩展Unicode字符串s 中格式为%name% 的环境字符串。

```
FlushKey(key
)
```

将 *key* 的属性写入注册表，并强制将修改保存到磁盘。只有当应用程序必须将注册表数据保存在磁盘上时，才应该调用此函数。它直到数据被写入后才会返回。一般情况下没有必要使用这个函数。

```
RegLoadKey(key
, sub_key
, filename
)
```

创建一个子键，并将来自文件的注册表信息保存到此子键中。 *key* 是打开的键或者

是预定义的HKEY_* 常量之一。 *sub_key* 是字符串，用于识别要加载的子键。 *filename* 是要从中加载数据的文件的名称。此文件的内容必须是使用SaveKey() 函数创建的，而且调用进程必须具有SE_RESTORE_PRIVILEGE 权限才能使用此函数。如果ConnectRegistry() 函数返回 *key* ， *filename* 应该是相对于远程计算机的一条路径。

```
OpenKey(key, sub_key[, res [, sam]])
```

打开一个键。 *key* 是打开的键或者是预定义的HKEY_* 常量之一。 *sub_key* 是字符串，用于识别要打开的子键。 *res* 是保留的整数，必须为0（默认值）。 *sam* 是整数，用于定义键的安全访问掩码，默认值为KEY_READ 。下面列出了 *sam* 的可取值。

- KEY_ALL_ACCESS
- KEY_CREATE_LINK
- KEY_CREATE_SUB_KEY
- KEY_ENUMERATE_SUB_KEYS
- KEY_EXECUTE
- KEY_NOTIFY
- KEY_QUERY_VALUE
- KEY_READ
- KEY_SET_VALUE
- KEY_WRITE

```
OpenKeyEx()
```

同OpenKey() 函数。

```
QueryInfoKey(key
)
```

返回关于键的信息，返回值是元组(*num_subkeys* , *num_values* , *last_modified*)，其中 *num_subkeys* 是子键的数量， *num_values* 是值的数量，而 *last_modified* 是一个长整数，其中包含最后一次修改的时间。时间测量从1601年1月1日开始计算，单位是100纳秒。

```
QueryValue(key
, sub_key
)
```

返回键的未命名值，返回值是字符串。 *key* 是打开的键或者是预定义的HKEY_* 常量之一。 *sub_key* 是要使用子键的名称，如果有的话。如果省略此参数，函数将返回与 *key* 相关的值。此函数返回的是第1个名称为空的值的数据，但不返回类型（如需要返回类型，使用QueryValueEx 函数）。

```
QueryValueEx(key
, value_name
)
```

返回元组(*value* , *type*)，其中包含一个键的数据值和类型。 *key* 是打开的键或者是预定义的HKEY_* 常量之一。 *value_name* 是要返回的值的名称。返回的类型是EnumValue() 函数中描述的整数代码之一。

```
SaveKey(key
, filename
)
```

将 *key* 及其子键保存到一个文件。 *key* 是打开的键或者预定义的HKEY_* 常量之一。 *filename* 必需为不存在的文件，而且不应该包含文件名扩展。此外，调用者必须拥有备份权限，操作才能成功完成。

```
SetValue(key
, sub_key
, type
, value
```

```
)
```

设置一个键的值。key是打开的键或者是预定义的HKEY_* 常量之一。 *sub_key* 是要设置的子键。 *type* 是整数类型代码，目前限制为REG_SZ 。 *value* 是包含值数据的字符串。如果 *sub_key* 不存在，就创建它。必须已经使用KEY_SET_VALUE 访问权限打开 *key* ，此函数才能成功执行。

```
SetValueEx(key
, value_name
, reserved
, type
, value
)
```

设置一个键的值字段。 *key* 是打开的键或者是预定义的HKEY_* 常量之一。 *value_name* 是值的名称。 *type* 是整数的类型代码，这与EnumValue() 函数相同。 *value* 是包含新值的字符串。当设置数字类型的值时（如REG_DWORD ）， *value* 仍然是包含原始数据的字符串。此字符串可以使用struct 模块进行创建。 *reserved* 参数目前被忽略，可以设为任意内容，它的值不会被使用。

注意

- 如果函数返回Windows HKEY对象，它返回的就是特殊的注册表句柄对象。PyHKEY 类中描述了这个特殊对象。可以使用int() 函数将此对象转换为Windows句柄值。还可以使用这个对象和上下文管理协议一起自动关闭底层的句柄。例如：

```
with winreg.OpenKey(winreg.HKEY_LOCAL_MACHINE, "spam") as key:

statements
```

---

① 中文版名为《C程序设计语言（第2版）》由机械工业出版社出版。——译者注

② 中文版名为《UNIX环境高级编程（第2版）》由人民邮电出版社出版。——译者注

第20章　线程与并发

本章描述在Python中编写并发程序时使用的库模块和编程策略。主题包括线程、消息传递、多进程和协程。在讲述特定的库模块之前，首先要说明一些基本的概念。

20.1　基本概念

一个运行的程序称作进程 。每个进程都有自己的系统状态，包括内存、已打开文件列表、用于跟踪正在执行的指令的程序计数器以及用于保存函数的局部变量的调用栈。通常在一个控制流序列中，进程逐条执行语句，这有时被称为进程的主线程 。在任何一个给定的时刻，程序都只做一件事情。

程序可以使用库函数创建新的进程，如os 或subprocess 模块中的函数（如os.fork() 、subprocess.Popen 等）。然而，这些叫子进程 的进程是作为完全独立的实体运行的——每个子进程都有自己的私有系统状态和执行主线程。因为子进程是独立的，所以它可以与原始进程并发执行。也就是说，创建子进程的进程可以继续处理别的事情，同时子进程在后台执行它自己的任务。

尽管进程是孤立的，但它们可以彼此通信，这称为进程间通信 （Interprocess Communication，IPC）。进程间通信的最常见形式是基于消息传递 。一条消息就是一块原始字节的缓存。然后，就可以使用像send() 和recv() 这样的简单操作，通过一个I/O通道（如管道或网络套接字）来传输或接收消息。另一种不太常见的IPC机制依赖于内存映射区域（参见mmap 模块）。借助内存映射，进程可以创建共享的内存区域。如果对这些内存区域进行修改，那么查看这些区域的所有进程都能看到这些修改。

如果需要同时处理多个任务，可以在一个应用程序内使用多个进程，每个进程负责处理一部分工作。另一种将工作细分为多个任务的方法是使用线程。线程 类似于进程，也具有自己的控制流和执行栈。但线程运行在创建它的进程内部，它们共享所有的数据和系统资源。当应用程序需要并发执行多个任务时，可以使用线程，但可能存在大量需要在各个任务之间共享的系统状态。

使用多个进程或线程时，主机操作系统负责安排它们的工作。安排的具体做法是：给每个进程（线程）分配一个小的时间片，并在所有活动任务之间快速循环——给每个任务分配一部分可用的CPU周期。例如，如果系统同时运行10个活动的进程，操作系统将给每个进程分配大约1/10的CPU时间，同时在进程之间快速循环。在具有多个CPU核心的系统上，操作系统在安排进程时可以尽可能使用每个CPU，从而并行执行进程。

编写使用并发执行的程序原本就很复杂，其复杂性的一个主要原因就是同步和访问共享数据。也就是说，多个任务同时更新一个数据结构可能导致数据损坏和程序状态不一致（这个问题的正式说法是竞争条件 ）。要解决这些问题，并发程序必须找出关键的代码段，并使用互斥锁和其他类似的同步手段保护它们。例如，如果不同的线程尝试同时向一个文件写入数据，可以使用互斥锁来同步它们的操作，从而当其中一个线程开始写入时，其他线程必须等到该线程结束才可以开始写入。此场景的代码通常如下所示：

```
write_lock = Lock()
...
# 发生写入操作的关键部分
write_lock.acquire()
f.write("Here's some data.\n")
f.write("Here's more data.\n")
...
write_lock.release()
```

Jason Whittington曾经讲过一个冷笑话：“问：为什么多线程的小鸡要过马路？答：想马路，去对面。”[①] 这个笑话道出了任务同步和并发编程所带来的问题。如果你抓着头皮说：“我没搞懂。”，那么在深入阅读本章余下内容之前，最好再仔细读一读。

20.2 并发编程与Python

在大多数系统上，Python同时支持消息传递和基于线程的并发编程。尽管大多数程序员熟悉的往往是线程接口，但实际上Python线程受到的限制有很多。尽管最低限度是线程安全的，但Python解释器还是使用了内部的GIL（Global Interpreter Lock，全局解释器锁），在任意指定的时刻只允许单个Python线程执行。无论系统上存在多少个可用的CPU核心，这限制了Python程序只能在一个处理器上运行。尽管GIL经常是Python社区中争论的热点，但在可以预见的将来它不太可能消失。

GIL的存在直接影响着许多Python程序员解决并发编程问题的方式。如果一个应用程序的大部分是I/O密集型的，那么使用线程一般没有问题，因为额外的处理器对于花费大多数时间等待事件的程序帮助不大。对于涉及大量CPU处理的应用程序而言，使用线程来细分工作没有任何好处，反而还会降低程序的运行速度（一般比你想象的还要慢得多）。为此，用户需要使用子进程和消息传递。

即使在使用线程时，很多程序员也会发现它们的伸缩特性十分怪异。例如，一台使用线程的网络服务器对于100个线程工作情况良好，但如果增加到10 000个线程，性能就会变得很糟糕。一般而言，程序员不会真地编写使用10 000个线程的程序，因为每个线程都需要有自己的系统资源，而且还会产生与线程上下文切换、锁和其他事务相关的开销，算下来不是个小数目（更不要提把所有线程都限制在一个CPU上运行）。为了解决这个问题，较为常见的做法是把这类应用程序重新构造为异步事件处理系统。例如，中心的事件循环可以使用`select` 模块监控所有I/O，并将异步事件分离给大量I/O处理器。这是诸如`asyncore` 这样的库模块和诸如Twisted（http://twistedmatrix/com ）等流行的第三方模块的基础。

展望未来，如果要在Python中进行各种类型的并发编程，消息传递很可能是必须熟练掌握的概念。即便使用线程，常推荐的方法也是将应用程序的结构设计为大量独立的线程集合，这些线程通过消息队列交换数据。这种方法往往很少出错，因为它极大地减少了对使用锁和其他同步手段的需求。消息传递还会自然扩展到网络和分布式系统中。例如，如果部分程序以线程形式启动，并给这个线程发送消息，那么该组件稍后将被迁移到一个单独的进程，或者通过基于网络连接发送消息迁移到另一台计算机上。消息传递的抽象也与高级Python功能（如协程）有关。例如，协程 是可以接收并处理发送给它的消息的函数。因此，掌握了消息传递之后，写出的程序会较之前更加灵活。

本章的余下部分讲述支持并发编程的各个库模块。最后提供关于常用编程习惯方法的详细信息。

20.3 multiprocessing

multiprocessing 模块为在子进程中运行任务、通信和共享数据，以及执行各种形式的同步提供支持。这个编程接口有意模仿threading 模块中线程的编程接口。但和线程不同，进程没有任何共享状态，这一点需要重点强调。因此，如果某个进程修改数据，改动只限于该进程内。

multiprocessing 模块的功能众多，是最大和最高级的内置库之一。在这里讲述该模块的所有细节是不可能的，但会给出一些例子来说明它的一些关键部分。有经验的程序员应该吃透这些例子，并扩展它们以解决更大的问题。

20.3.1 进程

multiprocessing 模块中所有功能的重点都放在进程上。下面的类描述了它们。

```
Process([group
 [, target
 [, name
 [, args
 [, kwargs
]]]]])
```

这个类表示运行在一个子进程中的任务。应该使用关键字参数来指定构造函数中的参数。 *target* 是当进程启动时执行的可调用对象， *args* 是传递给 *target* 的位置参数的元组，而 *kwargs* 是传递给target 的关键字参数的字典。如果省略 *args* 和 *kwargs* 参数，将不带参数调用 *target* 。 *name* 是为进程指定描述性名称的字符串。 *group* 参数未使用，值始终为None 。这个构造函数简单地构造了一个Process 进程，这与threading 模块中创建线程的过程相似。

Process 的实例 *p* 具有以下方法。

```
p
.is_alive()
```

如果 *p* 仍然运行，返回True 。

```
p

.join([
timeout

])
```

等待进程 *p* 终止。 *timeout* 是可选的超时期限。进程可以被连接无数次，但如果连接自身则会出错。

```
p
.run()
```

进程启动时运行的方法。默认情况下，会调用传递给Process 构造函数的 *target* 。定义进程的另一种方法是继承Process 类并重新实现run() 函数。

```
p
.start()
```

启动进程。这将运行代表进程的子进程，并调用该子进程中的 *p* .run() 函数。

```
p
.terminate()
```

强制终止进程。如果调用此函数，进程 *p* 将被立即终止，同时不会进行任何清理动作。如果进程p 创建了它自己的子进程，这些进程将变为僵尸进程。使用此方法时需特别小心。如果 *p* 保存了一个锁或参与了进程间通信，那么终止它可能会导致死锁或I/O损坏。

Process实例 *p* 也具有以下数据属性。

```
p
.authkey
```

进程的身份验证键。除非显式设定，这是由os.urandom() 函数生成的32字符的字符串。这个键的用途是为涉及网络连接的底层进程间通信提供安全性。这类连接只有在两端具有相同的身份验证键时才能成功。

```
p
.daemon
```

一个布尔标志，指示进程是否是后台进程。当创建它的Python进程终止时，后台（Daemonic）进程将自动终止。另外，禁止后台进程创建自己的新进程。 *p* .daemon 的值必须在使用 *p* .start() 函数启动进程之前进行设置。

```
p
.exitcode
```

进程的整数退出代码。如果进程仍然在运行，它的值为None 。如果值为负数，-*N* 表示进程由信号 *N* 所终止。

```
p
.name
```

进程的名称。

```
p
.pid
```

进程的整数进程ID。

下面这个例子说明了如何以单独进程的形式创建和启动函数（或其他可调用对象）：

```
import multiprocessing
import time

def clock(interval):
    while True:
        print("The time is %s" % time.ctime())
        time.sleep(interval)

if __name__ == '__main__':
   p = multiprocessing.Process(target=clock, args=(15,))
   p.start()
```

下面这个例子说明了如何将这个进程定义为继承自Process 的类：

```
import multiprocessing
import time

class ClockProcess(multiprocessing.Process):
    def __init__(self,interval):
        multiprocessing.Process.__init__(self)
        self.interval = interval
    def run(self):
        while True:
            print("The time is %s" % time.ctime())
            time.sleep(self.interval)

if __name__ == '__main__':
   p = ClockProcess(15)
```

```
    p.start()
```

在这两个例子中，子进程应该每15秒钟打印一次时间。需要重点强调的是：为了实现跨平台的可移植性，只能像上面这样由主程序创建新的进程。这在UNIX上是可选的，但在Windows上是必需的。还应该注意，在Windows上，很可能需要在命令shell（`command.exe`）中运行上述例子，而不能在IDLE这样的Python IDE中运行。

20.3.2 进程间通信

`multiprocessing` 模块支持进程间通信的两种主要形式：管道和队列。这两种方法都是使用消息传递实现的，但队列接口有意模仿线程程序中常见的队列用法。

```
Queue([maxsize
])
```

创建共享的进程队列。`maxsize` 是队列中允许的最大项数。如果省略此参数，则无大小限制。底层队列使用管道和锁实现。另外，还需要运行支持线程以便将队列中的数据传输到底层管道中。

`Queue` 的实例 *q* 具有以下方法。

```
q
.cancel_join_thread()
```

不会在进程退出时自动连接后台线程。这可以防止`join_thread()`方法阻塞。

```
q
.close()
```

关闭队列，防止队列中加入更多数据。调用此方法时，后台线程将继续写入那些已入队列但尚未写入的数据，但将在此方法完成时马上关闭。如果 *q* 被垃圾回收，将自动调

用此方法。关闭队列不会在队列消费者中生成任何类型的数据结束信号或异常。例如，如果某个消费者正被阻塞在get() 操作上，关闭生产者中的队列不会导致get() 方法返回错误。

```
q
.empty()
```

如果调用此方法时 *q* 为空，返回True 。如果其他进程或线程正在往队列中添加项，结果是不可靠的。也就是说，在返回和使用结果之间，队列中可能已经加入了新项。

```
q
.full()
```

如果 *q* 已满，返回True 。由于线程的存在，结果也可能不可靠（参见 *q* .empty() 方法）。

```
q
.get([block
 [, timeout
]])
```

返回 *q* 中的一个项。如果 *q* 为空，此方法将阻塞，直到队列中有项可用为止。*block* 用于控制阻塞行为，默认为True 。如果设置为False ，将引发Queue.Empty 异常（定义在Queue 库模块中）。timeout 是可选超时时间，用在阻塞模式中。如果在指定的时间间隔内没有项变为可用，将引发Queue.Empty 异常。

```
q
.get_nowait()
```

同 *q* .get(False) 方法。

```
q
.join_thread()
```

连接队列的后台线程。此方法用于在调用 *q* .close() 方法之后，等待所有队列项被消耗。默认情况下，此方法由不是 *q* 的原始创建者的所有进程调用。调用 *q* .cancel_join_thread() 方法可以禁止这种行为。

```
q
.put(item
 [, block
 [, timeout
]])
```

将 *item* 放入队列。如果队列已满，此方法将阻塞至有空间可用为止。 *block* 控制阻塞行为，默认为True 。如果设置为False ，将引发Queue.Empty 异常（定义在Queue 库模块中）。 *timeout* 指定在阻塞模式中等待可用空间的时间长短。超时后将引发Queue.Full 异常。

```
q
.put_nowait(item
)
```

同 *q* .put(*item,False*) 方法。

```
q
.qsize()
```

返回目前队列中项的正确数量。此函数的结果并不可靠，因为在返回结果和在稍后程序中使用结果之间，队列中可能添加或删除了项。在某些系统上，此方法可能引发NotImplementedError 异常。

```
JoinableQueue([maxsize
])
```

创建可连接的共享进程队列。这就像是一个Queue 对象，但队列允许项的消费者通知生产者项已经被成功处理。通知进程是使用共享的信号和条件变量来实现的。

JoinableQueue 的实例 *p* 除了与Queue 对象相同的方法之外，还具有以下方法。

```
q
.task_done()
```

消费者使用此方法发出信号，表示 *q* .get() 返回的项已经被处理。如果调用此方法的次数大于从队列中删除的项的数量，将引发ValueError 异常。

```
q
.join()
```

生产者使用此方法进行阻塞，直到队列中的所有项均被处理。阻塞将持续到为队列中的每个项均调用 *q* .task_done() 方法为止。

下面的例子说明如何建立永远运行的进程，使用和处理队列上的项。生产者将项放入队列，并等待它们被处理。

```
import multiprocessing

def consumer(input_q):
    while True:
        item = input_q.get()
        # 处理项
        print(item)        # 此处替换为有用的工作
        # 发出信号通知任务完成
        input_q.task_done()

def producer(sequence, output_q):
    for item in sequence:
        # 将项放入队列
        output_q.put(item)

# 建立进程
if __name__ == '__main__':
   q = multiprocessing.JoinableQueue()
   # 运行消费者进程
   cons_p = multiprocessing.Process(target=consumer,args=(q,))
   cons_p.daemon=True
   cons_p.start()

   # 生产多个项。sequence代表要发送给消费者的项序列。
   # 在实践中，这可能是生成器的输出或通过一些其他方式生产出来。
   sequence = [1,2,3,4]
   producer(sequence, q)

   # 等待所有项被处理
   q.join()
```

在这个例子中，消费者进程设置为后台进程，因为它永远运行，而我们希望当主程序结束时它随之终止（如果忘记这么做，程序将会挂起）。这里使用了JoinableQueue，以便让生产者实际了解队列中的所有项何时被处理完毕。join() 操作保证了这一点。如果忘记这个步骤，消费者进程将在有时间完成所有工作之前被终止。

如果需要，可以使用多个进程从同一个队列中放置或者获取项。例如，如果要构造消费者进程池，可以编写下面这样的代码：

```
if __name__ == '__main__':
   q = multiprocessing.JoinableQueue()
   # 启动一些消费者进程
   cons_p1 = multiprocessing.Process(target=consumer,args=(q,))
   cons_p1.daemon=True
   cons_p1.start()

   cons_p2 = multiprocessing.Process(target=consumer,args=(q,))
   cons_p2.daemon=True
```

```
   cons_p2.start()

   # 生产多个项。sequence代表要发送给消费者的项序列。
   # 在实践中，这可能是生成器的输出或者是通过一些其他方式生产出来的。
   sequence = [1,2,3,4]
   producer(sequence, q)

   # 等待所有项被处理
   q.join()
```

编写这类代码时，要注意放入队列中的每个项都会被序列化，然后通过管道或套接字连接发送给进程。一般来说，发送数量较少的大对象比发送大量小对象更好。

在某些应用程序中，生产者需要通知消费者，它们将不再生产项，消费者应该关闭。为此，编写的代码中应该使用哨兵 （sentinel）——表示完成的特殊值。下面这个例子使用None 作为哨兵说明这个概念：

```
import multiprocessing

def consumer(input_q):
    while True:
        item = input_q.get()
        if item is None:
            break
        # 处理项
        print(item)        # 替换为有用的工作
    # 关闭
    print("Consumer done")

def producer(sequence, output_q):
    for item in sequence:
        # 把项放入队列
        output_q.put(item)

if __name__ == '__main__':
   q = multiprocessing.Queue()
   # 启动消费者进程
   cons_p = multiprocessing.Process(target=consumer,args=(q,))
   cons_p.start()

   # 生产项
   sequence = [1,2,3,4]
   producer(sequence, q)

    # 在队列上安置哨兵，发出完成信号
    q.put(None)
    # 等待消费者进程关闭
    cons_p.join()
```

如果像上面这个例子中那样使用哨兵，一定要在队列上为每个消费者上都安置哨兵。例如，如果有三个消费者进程使用队列上的项，那么生产者需要在队列上安置三个哨兵，才能让所有消费者都关闭。

除了使用队列外，还可以使用管道在进程之间执行消息传递。

```
Pipe([duplex
])
```

在进程之间创建一条管道，并返回元组(*conn1* ，*conn2*)，其中 *conn1* 和 *conn2* 是表示管道两端的Connection 对象。默认情况下，管道是双向的。如果将 *duplex* 置为False， *conn1* 只能用于接收，而 *conn2* 只能用于发送。必须在创建和启动使用管道的Process 对象之前调用Pipe() 方法。

Pipe() 方法返回的Connection 对象的实例 *c* 具有以下方法和属性。

```
c
.close()
```

关闭连接。如果 *c* 被垃圾回收，将自动调用此方法。

```
c
.fileno()
```

返回连接使用的整数文件描述符。

```
c
.poll([timeout
])
```

如果连接上的数据可用，返回True 。 *timeout* 指定等待的最长时限。如果省略此参数，方法将立即返回结果。如果将 *timeout* 置为None ，操作将无限期地等待数据到

达。

```
c
.recv()
```

接收 *c* .send() 方法返回的对象。如果连接的另一端已经关闭，再也不存在任何数据，将引发EOFError 异常。

```
c
.recv_bytes([maxlength
])
```

接收 *c* .send_bytes() 方法发送的一条完整的字节消息。 *maxlength* 指定要接收的最大字节数。如果进入的消息超过了这个最大值，将引发IOError 异常，并且在连接上无法进行进一步读取。如果连接的另一端已经关闭，再也不存在任何数据，将引发EOFError 异常。

```
c.recv_bytes_into(buffer
 [, offset
])
```

接收一条完整的字节消息，并把它保存在 *buffer* 对象中，该对象支持可写入的缓冲区接口（即bytearray 对象或类似的对象）。 *offset* 指定缓冲区中放置消息处的字节位移。返回值是收到的字节数。如果消息长度大于可用的缓冲区空间，将引发BufferTooShort 异常。

```
c.send(obj
)
```

通过连接发送对象。 *obj* 是与序列化兼容的任意对象。

```
c.send_bytes(buffer
 [, offset
 [, size
]])
```

通过连接发送字节数据缓冲区。 *buffer* 是支持缓冲区接口的任意对象， *offset* 是缓冲区中的字节偏移量，而 *size* 是要发送字节数。结果数据以单条消息的形式发出，然后调用 *c* .recv_bytes() 函数进行接收。

可以通过与队列类似的方式使用管道。下面这个例子说明如何使用管道实现前面的生产者—消费者问题：

```
import multiprocessing
# 使用管道上的项
def consumer(pipe):
    output_p, input_p = pipe
    input_p.close()      # 关闭管道的输入端
    while True:
        try:
            item = output_p.recv()
        except EOFError:
            break
        # 处理项
        print(item)       # 替换为有用的工作
    # 关闭
    print("Consumer done")

# 生产项并将其放置到队列上。sequence是代表要处理项的可迭代对象
def producer(sequence, input_p):
    for item in sequence:
        # 将项放在队列上
        input_p.send(item)

if __name__ == '__main__':
    (output_p, input_p) = multiprocessing.Pipe()
    # 启动消费者进程
    cons_p = multiprocessing.Process(target=consumer,args=((output_p, input_p),))
    cons_p.start()

    # 关闭生产者中的输出管道
    output_p.close()

    # 生产项
```

```
    sequence = [1,2,3,4]
    producer(sequence, input_p)

    # 关闭输入管道，表示完成
    input_p.close()

    # 等待消费者进程关闭
    cons_p.join()
```

应该特别注意管道端点的正确管理问题。如果生产者或消费者中都没有使用管道的某个端点，就应将其关闭。这也说明了为何在生产者中关闭了管道的输出端，在消费者中关闭管道的输入端。如果忘记执行这些步骤，程序可能在消费者中的recv()操作上挂起。管道是由操作系统进行引用计数的，必须在所有进程中关闭管道后才能生成EOFError异常。因此，在生产者中关闭管道不会有任何效果，除非消费者也关闭了相同的管道端点。

管道可用于双向通信。利用通常在客户端/服务器计算中使用的请求/响应模型或远程过程调用，就可以使用管道编写与进程交互的程序，例如：

```
import multiprocessing
# 一个服务器进程
def adder(pipe):
    server_p, client_p = pipe
    client_p.close()
    while True:
        try:
            x,y = server_p.recv()
        except EOFError:
            break
        result = x + y
        server_p.send(result)
    # 关闭
    print("Server done")

if __name__ == '__main__':
    (server_p, client_p) = multiprocessing.Pipe()
    # 启动服务器进程
    adder_p = multiprocessing.Process(target=adder,args=((server_p, client_p),))
    adder_p.start()

    # 关闭客户端中的服务器管道
    server_p.close()

    # 在服务器上提出一些请求
    client_p.send((3,4))
    print(client_p.recv())

    client_p.send(('Hello','World'))
    print(client_p.recv())

    # 完成。关闭管道
    client_p.close()

    # 等待消费者进程关闭
    adder_p.join()
```

在这个例子中，adder() 函数以服务器的形式运行，等待消息到达管道的端点。收到之后，它会执行一些处理并将结果发回给管道。要记住，send() 和recv() 方法使用pickle 模块对对象进行序列化。在本例中，服务器接收到元组(*x , y*) 并将其作为输入，然后返回结果x+y 。但对于使用远程过程调用的高级应用程序而言，应该使用下面描述的进程池。

20.3.3 进程池

下面的类可以创建进程池，可以把各种数据处理任务都提交给进程池。进程池提供的功能有点类似于列表解析和函数式编程操作（如映射—归约）提供的功能。

```
Pool([numprocess
 [,initializer
 [, initargs
]]])
```

创建工作进程池。 *numprocess* 是要创建的进程数。如果省略此参数，将使用cpu_count() 的值。 *initializer* 是每个工作进程启动时要执行的可调用对象。*initargs* 是要传递给 *initializer* 的参数元组。 *initializer* 默认为None 。

Pool 类的实例 *p* 支持以下操作。

```
p
.apply(func
 [, args
 [, kwargs
]])
```

在一个池工作进程中执行函数(**args , **kwargs*)，然后返回结果。这里要强调一点：此操作并不会在所有池工作进程中并行执行 *func* 函数。如果要使用不同参数并发地执行 *func* 函数，必须从不同线程调用 *p* .apply() 函数或者使用 *p* .apply_async() 函数。

```
p
.apply_async(func
 [, args
 [, kwargs
 [, callback
]]])
```

在一个池工作进程中异步地执行函数(*args ，**kwargs)，然后返回结果。此方法的结果是AsyncResult 类的实例，稍后可用于获得最终结果。 *callback* 是可调用对象，接受输入参数。当 *func* 的结果变为可用时，将立即传递给 *callback* 。 *callback* 禁止执行任何阻塞操作，否则将阻塞接收其他异步操作中的结果。

```
p
.close()
```

关闭进程池，防止进行进一步操作。如果还有挂起的操作，它们将在工作进程终止之前完成。

```
p
.join()
```

等待所有工作进程退出。此方法只能在close() 或terminate() 方法之后调用。

```
p
.imap(func, iterable
 [, chunksize
```

```
])
```

map() 函数的版本之一，返回迭代器而非结果列表。

```
p
.imap_unordered(func
, iterable
 [, chunksize
]))
```

同imap() 函数一样，只是结果的顺序根据从工作进程接收到的时间任意确定。

```
p
.map(func
, iterable
 [, chunksize
])
```

将可调用对象 *func* 应用给 *iterable* 中的所有项，然后以列表的形式返回结果。通过将iterable划分为多块并将工作分派给工作进程，可以并行地执行这项操作。*Chunksize* 指定每块中的项数。如果数据量较大，可以增大 *chunksize* 的值来提升性能。

```
p
.map_async(func
, iterable
```

```
 [, chunksize

 [, callback

]])
```

同map() 函数，但结果的返回是异步的。返回值是AsyncResult 类的实例，稍后可用于获得结果。 *callback* 是接受一个参数的可调用对象。如果提供 *callable* ，当结果变为可用时，将使用结果调用 *callable* 。

```
p

.terminate()
```

立即终止所有工作进程，同时不执行任何清理或结束任何挂起工作。如果p 被垃圾回收，将自动调用此函数。

方法apply_async() 和map_async() 的返回值是AsyncResult 实例。AsyncResult 实例a 拥有以下方法。

```
a

.get([timeout

])
```

返回结果，如果有必要则等待结果到达。 *timeout* 是可选的超时。如果结果在指定时间内没有到达，将引发multiprocessing.TimeoutError 异常。如果远程操作中引发了异常，它将在调用此方法时再次被引发。

```
a

.ready()
```

如果调用完成，返回True 。

```
a
.sucessful()
```

如果调用完成且没有引发异常，返回True 。如果在结果就绪之前调用此方法，将引发AssertionError 异常。

```
a
.wait([timeout
])
```

等待结果变为可用。 *timeout* 是可选的超时。

下面的例子说明如何使用进程池构建字典，将整个目录中文件的文件名映射为SHA512摘要值：

```
import os
import multiprocessing
import hashlib

# 可以手动调整的部分参数
BUFSIZE  = 8192           # 读取缓冲区大小
POOLSIZE = 2              # 工作进程的数量

def compute_digest(filename):
    try:
        f = open(filename,"rb")
    except IOError:
        return None
    digest = hashlib.sha512()
    while True:
        chunk = f.read(BUFSIZE)
        if not chunk: break
        digest.update(chunk)
    f.close()
    return filename, digest.digest()
```

```
def build_digest_map(topdir):
    digest_pool = multiprocessing.Pool(POOLSIZE)
    allfiles = (os.path.join(path,name)
                for path, dirs, files in os.walk(topdir)
                     for name in files)

    digest_map = dict(digest_pool.imap_unordered(compute_digest,allfiles,20))
    digest_pool.close()
    return digest_map

# 尝试按照需要修改目录名称
if __name__ == '__main__':
   digest_map = build_digest_map("/Users/beazley/Software/Python-3.0")
   print(len(digest_map))
```

在这个例子中，使用生成器表达式指定一个目录树中所有文件的路径名称序列。然后使用imap_unordered() 函数将这个序列分割并传递给进程池。每个池工作进程使用compute_digest() 函数为它的文件计算SHA512摘要值。将结果发回给主进程，然后收集到Python字典中。尽管它绝不是一个科学的结果，但在我的双核Macbook上运行时，这个例子的运行速度比单进程的解决方案提升了75%。

要记住，只有池工作进程执行足够弥补额外通信开销的工作，使用进程池才有意义。一般而言，对于简单的计算（如两个数相加），使用进程池是没有意义的。

20.3.4 共享数据与同步

通常，进程之间彼此是完全孤立的，唯一的通信方式是队列或管道。但可以使用两个对象来表示共享数据。其实，这些对象使用了共享内存（通过mmap 模块）使访问多个进程成为可能。

```
Value(typecode
, arg1
, ... argN
, lock
)
```

在共享内存中创建ctypes 对象。 *typecode* 要么是包含array 模块使用的相同类型代码（如'i' 、'd' 等）的字符串，要么是来自ctypes 模块的类型对象（如ctypes.c_int 、ctypes.c_double 等）。所有额外的位置参数 *arg1* 、 *arg2* ……*argN* 将传递给指定类型的构造函数。 *lock* 是只能使用关键字调用的参数，如果把它置为True （默认值），将创建一个新锁来保护对值的访问。如果传入一个现有锁，如Lock 或RLock 实例，该锁将用于进行同步。如果 *v* 是Value 创建的共享值的实例，便可使用 *v* .value 访问底层的值。例如，读取 *v* .value 将获取值，而赋值 *v* .value

将修改值。

```
RawValue(typecode
, arg1
, ..., argN
)
```

同 *Value* 对象，但不存在锁定。

```
Array(typecode
, initializer
, lock
)
```

在共享内存中创建ctypes 数组。 *typecode* 描述了数组的内容，意义与Value() 函数中的相同。 *initializer* 要么是设置数组初始大小的整数，要么是项序列，其值和大小用于初始化数组。 *lock* 是只能使用关键字调用的参数，意义与Value*()* 函数中相同。如果 *a* 是Array 创建的共享数组的实例，便可使用标准的Python索引、切片和迭代操作访问它的内容，其中每种操作均由锁进行同步。对于字节字符串， *a* 还具有 *a* .value 属性，可以把整个数组当作一个字符串进行访问。

```
RawArray(typecode
, initializer
)
```

同 *Array* 对象，但不存在锁定。当所编写的程序必须一次性操作大量的数组项时，如果同时使用这种数据类型和用于同步的单独大的锁（如果需要的话），性能将得到极大的提升。

除了使用Value() 和Array() 创建的共享值之外，multiprocessing 模块还提供以

下同步原语的共享版本。

原　　语	描　　述
Lock	互斥锁
RLock	可重入的互斥锁（同一进程可以多次获得它，同时不会造成阻塞）
Semaphore	信号量
BoundedSemaphore	有边界的信号量
Event	事件
Condition	条件变量

这些对象的行为与threading 模块中定义的名称相同的同步原语相似。请参考threading 文档了解更多细节。

应该注意，使用多进程后，通常不必再担心与锁、信号量或类似构造的底层同步，这一点与线程不相伯仲。在某种程度上，管道上的send() 和receive() 操作，以及队列上的put() 和get() 操作已经提供了同步功能。但是，在某些特定的设置下还是需要用到共享值和锁。下面这个例子说明了如何使用共享数组代替管道，将一个由浮点数组成的Python列表发送给另一个进程：

```
import multiprocessing

class FloatChannel(object):
    def __init__(self, maxsize):
        self.buffer     = multiprocessing.RawArray('d',maxsize)
        self.buffer_len = multiprocessing.Value('i')
        self.empty      = multiprocessing.Semaphore(1)
        self.full       = multiprocessing.Semaphore(0)
    def send(self,values):
        self.empty.acquire()            # 只在缓存为空时继续
        nitems = len(values)
        self.buffer_len = nitems         # 设置缓冲区大小
        self.buffer[:nitems] = values    # 将值复制到缓冲区中
        self.full.release()             # 发信号通知缓冲区已满
    def recv(self):
        self.full.acquire()             # 只在缓冲区已满时继续
        values = self.buffer[:self.buffer_len.value] # 复制值
        self.empty.release()            # 发信号通知缓冲区为空
        return values
# 性能测试。接收多条消息
def consume_test(count, ch):
    for i in xrange(count):
```

```
        values = ch.recv()

# 性能测试。发送多条消息
def produce_test(count, values, ch):
    for i in xrange(count):
        ch.send(values)

if __name__ == '__main__':
   ch = FloatChannel(100000)
   p = multiprocessing.Process(target=consume_test,
                               args=(1000,ch))
   p.start()
   values = [float(x) for x in xrange(100000)]
   produce_test(1000, values, ch)
   print("Done")
   p.join()
```

读者可以进一步研究这个例子。但在我的计算机上执行性能测试时，通过FloatChannel 发送一个较大的浮点数列表，速度比通过Pipe 发送快大约80%，因为后者必须对所有值进行序列化和反序列化。

20.3.5 托管对象

和线程不同，进程不支持共享对象。尽管可以像前面所述那样创建共享值和数组，但这对更高级的Python对象（如字典、列表或用户定义类的实例）不起作用。但是multiprocessing 模块确实提供了一种使用共享对象的途径，但前提是它们运行在所谓的管理器 的控制之下。管理器 是独立的子进程，其中存在真实的对象，并以服务器的形式运行。其他进程通过使用代理访问共享对象，这些代理作为管理器服务器的客户端运行。

使用简单托管对象的最直观方式是使用Manager() 函数。

```
Manager()
```

在一个单独的进程中创建运行的管理器。返回值是SyncManager 类型的实例，SyncManager 类型定义在multiprocessing.managers 模块中。

Manager() 函数返回的SyncManager 的实例 *m* 具有一系列方法，可用于创建共享对象并返回用于访问这些共享对象的代理。通常，可以创建一个管理器，并在启动任何新进程之前使用这些方法创建共享对象。这些已定义的方法如下。

```
m

.Array(typecode
```

```
, sequence
)
```

在服务器上创建共享的Array 实例并返回可访问它的代理。参见20.3.4节中关于参数的描述。

```
m
.BoundedSemaphore([value
])
```

在服务器上创建共享的threading.BoundedSemaphore 实例，并返回可访问它的代理。

```
m
.Condition([lock
])
```

在服务器上创建共享的threading.Condition 实例，并返回可访问它的代理。*lock* 是 *m* .Lock() 或 *m* .Rlock() 方法创建的代理实例。

```
m
.dict([args
])
```

在服务器上创建共享的dict 实例，并返回可访问它的代理。此方法的参数与内置函

数dict() 中的参数相同。

```
m
.Event()
```

在服务器上创建共享的threading. Event 实例，并返回可访问它的代理。

```
m
.list([sequence
])
```

在服务器上创建共享的list 实例，并返回可访问它的代理。此方法的参数与内置函数list() 中相同。

```
m
.Lock()
```

在服务器上创建共享的threading.Lock 实例，并返回可访问它的代理。

```
m
.Namespace()
```

在服务器上创建共享的namespace 对象，并返回可访问它的代理。namespace 就是一个类似于Python模块的对象。例如，如果 *n* 是一个命名空间代理，那么可以使用(.)赋值和读取属性，如n.*name = value* 或 *value = n.name* 。但 *name* 的名称十分重

要。如果 *name* 以字母开头，那么这个值就是管理器所拥有的共享对象的一部分，所有其他进程均可访问它。如果 *name* 以下划线开头，那么它只是代理对象的一部分，并不是共享的。

```
m
.Queue()
```

在服务器上创建共享的Queue.Queue 对象，并返回可访问它的代理。

```
m
.RLock()
```

在服务器上创建共享的threading.Rlock 对象，并返回可访问它的代理。

```
m
.Semaphore([value
])
```

在服务器上创建共享的threading.Semaphore 对象，并返回可访问它的代理。

```
m
.Value(typecode
, value
)
```

在服务器上创建一个共享的Value 对象，并返回可访问它的代理。参见20.3.4节中关于参数的描述。

下面的例子说明了如何使用管理器创建一个在进程之间共享的字典。

```
import multiprocessing
import time

# 只要设定要传递的事件，就打印d
def watch(d, evt):
    while True:
        evt.wait()
        print(d)
        evt.clear()
if __name__ == '__main__':
   m   = multiprocessing.Manager()
   d   = m.dict()        # 创建共享的dict
   evt = m.Event()       # 创建共享的Event

    # 启动监视字典的进程
    p = multiprocessing.Process(target=watch,args=(d,evt))
    p.daemon=True
    p.start()

    # 更新字典并通知监视者
    d['foo'] = 42
    evt.set()
    time.sleep(5)

    # 更新字典并通知监视者
    d['bar'] = 37
    evt.set()
    time.sleep(5)

    # 终止进程和管理器
    p.terminate()
    m.shutdown()
```

如果运行这个例子，每次设定要传递的事件时，watch() 函数都会打印出 *d* 的值。在主程序中，主进程中创建并操作了一个共享的字典和事件。运行这个例子时，将看到子进程打印数据。

如果需要其他类型的共享对象，如用户定义类的实例，则必须创建自定义管理器对象。为此，需要创建一个继承自BaseManager 类的类，BaseManager 类定义在multiprocessing.managers 子模块中。

```
managers.BaseManager([address

 [, authkey

]])
```

为用户定义对象创建管理器服务器的基类。address 是一个可选元组 (*hostname,port*)，用于指定服务器的网络地址。如果省略此参数，操作系统将简单地分配一个对应于某些空闲端口号的地址。authkey 是一个字符串，用于对连接到服务器的客户端进行身份验证。如果省略此参数，将使用current_process().authkey 的值。

如果 *mgrclass* 是一个继承自BaseManager 的类，可以使用以下类方法来创建用于返回代理给共享对象的方法。

```
mgrclass.register(typeid
 [, callable
 [, proxytype
 [, exposed
 [, method
_to
_typeid

[,create
_method
]]]]])
```

使用管理器类注册一种新的数据类型。 *typeid* 是一个字符串，用于命名特定类型的共享对象。这个字符串应该是有效的Python标识符。 *callable* 是创建或返回要共享的实例的可调用对象。 *proxytype* 是一个类，负责提供客户端中要使用的代理对象的实现。通常，这些类是默认生成的，因此一般置为None 。 *exposed* 是共享对象上方法名称的序列，它将会公开给代理对象。如果省略此参数，将使用 *proxytype.* exposed 的值，如果 *proxytype.* exposed 未定义，序列将包含所有的公共方法（所有不以下划线开头的可调用方法）。 *method_to_typeid* 是从方法名称到类型IDS的映射，这里的IDS用于指定哪些方法应该使用代理对象返回它们的结果。如果某个方法在这个映射中找不到，将复制和返回它的返回值。如果 *method_to_typeid* 为None ，将使用 *proxytype._method_to_typeid* 的值。 *create_method* 是一个布尔标志，用于指定是否在 *mgrclass* 中创建名为 *typeid* 的方法。它的默认值为True 。

从BaseManager 类派生的管理器的实例 *m* 必须手动启动才能运行。相关的属性和方法如下。

```
m
.address
```

元组(*hostname, port*)，代表管理器服务器正在使用的地址。

```
m
.connect()
```

连接到远程管理器对象，该对象的地址在BaseManager 构造函数中指定。

```
m.serve_forever()
```

在当前进程中运行管理器服务器。

```
m
.shutdown()
```

关闭由m.start() 方法启动的管理器服务器。

```
m
.start()
```

启动一个单独的子进程，并在该子进程中启动管理器服务器。

下面的例子说明了如何为用户定义类创建管理器：

```
import multiprocessing
from multiprocessing.managers import BaseManager

class A(object):
    def __init__(self,value):
        self.x = value
    def __repr__(self):
        return "A(%s)" % self.x
    def getX(self):
        return self.x
    def setX(self,value):
        self.x = value
    def __iadd__(self,value):
        self.x += value
        return self

class MyManager(BaseManager): pass
MyManager.register("A",A)

if __name__ == '__main__':
   m = MyManager()
   m.start()
   # 创建托管对象
   a = m.A(37)
   ...
```

在这个例子中，最后一条语句为位于管理器服务器上的A 创建了一个实例。上面代码中的变量 *a* 只是此实例的代理。此代理的行为类似于（但不完全等同于）服务器上的对象 *referent* 。首先，你会发现无法访问数据属性，而必须使用访问函数：

```
>>> a.x


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AutoProxy[A]' object has no attribute 'x'
>>> a.getX()


37
>>> a.setX(42)


>>>
```

借助代理，repr() 函数返回代表代理的字符串，而str() 函数返回的是引用对象上的__repr__() 的输出。例如：

```
>>> a


<AutoProxy[A] object, typeid 'A' at 0xcef230>
>>> print(a)


A(37)
>>>
```

在代理上无法访问特殊方法和以下划线（_）开头的所有方法。例如，如果尝试调用a.__iadd__()方法是不会奏效的：

```
>>> a += 37


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +=: 'AutoProxy[A]' and 'int'
>>> a.__iadd__(37)


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AutoProxy[A]' object has no attribute '__iadd__'
>>>
```

在更高级的应用程序中，可以自定义代理，从而进行更仔细地控制访问。这是通过定义继承自BaseProxy的类来实现的，BaseProxy类定义在multiprocessing.managers模块中。以下代码说明如何为前面代码中的A类自定义代理，从而正确地公开__iadd__()方法，并使用特性（property）公开x属性（attribute）：

```
from multiprocessing.managers import BaseProxy

class AProxy(BaseProxy):
    # referent上公开的所有方法列表
    _exposed_ = ['__iadd__','getX','setX']
    # 实现代理的公共接口
    def __iadd__(self,value):
        self._callmethod('__iadd__',(value,))
        return self
    @property
    def x(self):
        return self._callmethod('getX',())
    @x.setter
    def x(self,value):
        self._callmethod('setX',(value,))

class MyManager(BaseManager): pass
MyManager.register("A", A, proxytype=AProxy)
```

继承自BaseProxy 的类的实例 *proxy* 具有以下方法。

```
proxy
._callmethod(name [, args
 [, kwargs
]])
```

调用代理引用对象上的方法 *name* 。 *name* 是表示方法名称的字符串，args 是包含位置参数的元组，而 *kwargs* 是关键字参数的字典。方法 *name* 必须被显式公开，这通常是通过在代理类的_exposed_class 属性中包含名称来实现的。

```
proxy
._getvalue()
```

返回调用者中引用对象的副本。如果这次调用是在另一个进程中进行的，那么引用对象将被序列化，发送给调用者，然后再进行反序列化。如果无法序列化引用对象，将引发异常。

20.3.6 连接

使用multiprocessing 模块的程序可以与运行在同一台计算机上的其他进程或者位于远程系统上的进程进行消息传递。如果要求程序不仅能在一个系统上运行，而且还能扩展到一个计算集群上，那么可以使用此模块。multiprocessing.connection 子模块包含实现该目的的函数和类：

```
connections.Client(address
 [, family
 [, authenticate
 [, authkey
]])
```

连接到另一个进程，此进程必须已经正在侦听地址 *address* 。 *address* 是代表网络地址的元组(*hostname* , *port*)，或者代表UNIX域套接字的文件名，或者代表r'*servername* \pipe*pipename* ' 形式的字符串，代表远程系统 *servername* （本地计算机的 *servername* 为'.'）上的一条Windows命名管道。 *family* 是表示地址格式的字符串，一般是'AF_INET' 、'AF_UNIX' 或'AF_PIPE' 。如果省略此参数，将从 *address* 的格式推出它的值。 *authentication* 是一个布尔标志，指定是否使用摘要身份验证。 *authkey* 是包含身份验证密钥的字符串。如果省略此参数，将使用current_process().authkey 的值。此函数的返回值是Connection 对象，前面在20.3.2节的管道部分中讲到过这个对象。

```
connections.Listener([address
 [, family
 [, backlog
 [, authenticate
 [,
authkey
]]]]])
```

这个类实现了一台服务器，用于侦听和处理Client() 函数发出的连接。 *address* 、 *family* 、 *authenticate* 和 *authkey* 参数的意义与Client() 函数中相同。 *backlog* 是一个整数，当 *address* 参数指定一个网络连接时，对应于传递给套接字的listen() 方法的值。 *backlog* 的默认值为1。如果省略 *address* 参数，将选择默认地址。如果同时省略 *address* 和 *family* 两个参数，将选择本地系统上速度最快的可用通信模式。

Listener 实例 *s* 支持以下方法和属性。

```
s
.accept()
```

接受一个新连接，并返回一个Connection 对象。如果身份验证失败，将引发Authentication-Error 异常。

s

.address

侦听器正在使用的地址。

s

.close()

关闭侦听器正在使用的管道或套接字。

s

.last_accepted

接受的最后一个客户端的地址。

下面是一个服务器程序的例子，它负责侦听客户端并实现简单的远程操作（加法）：

```
from multiprocessing.connection import Listener

serv = Listener(('',15000),authkey='12345')
while True:
    conn = serv.accept()
    while True:
        try:
            x,y = conn.recv()
        except EOFError:
            break
        result = x + y
        conn.send(result)
    conn.close()
```

下面这个简单的客户端程序连接到这台服务器，然后发送一些消息：

```
from multiprocessing.connection import Client
conn = Client(('localhost',15000), authkey="12345")

conn.send((3,4))
r = conn.recv()
print(r)          # 打印'7'

conn.send(("Hello","World"))
r = conn.recv()
print(r)          # 打印'HelloWorld'

conn.close()
```

20.3.7 各种实用工具函数

还定义了以下实用工具函数。

```
active_children()
```

返回所有活动子进程的Process 对象组成的列表。

```
cpu_count()
```

返回系统上的CPU数量，如果能够确定的话。

```
current_process()
```

返回当前进程的Process 对象。

```
freeze_support()
```

在使用各种打包工具（如py2exe ）进行冻结的应用程序中，此函数应该作为主程序

的首行。使用此函数可以防止与在冻结的应用程序中启动子进程相关的运行时错误。

```
get_logger()
```

返回与多进程处理模块相关的日志记录对象，如果它不存在则创建之。返回的记录器不会把消息传播给根记录器，级别为logging.NOTSET，而且会将所有日志消息打印到标准错误上。

```
set_executable(executable
)
```

设置用于执行子进程的Python可执行程序的名称。这个函数只定义在Windows上。

20.3.8 多进程处理的一般建议

multiprocessing 模块是Python库中最高级和功能最强大的模块之一。记住下面这些一般性技巧，可以更好地掌握这个模块。

- 在构建较大型的应用程序之前，仔细阅读在线文档。尽管本节已经讲述了主要的基础知识，但官方文档还涉及了一些可能出现的更奇怪的问题。
- 确保进程之间传递的所有数据都能够序列化。
- 避免使用共享数据，尽可能使用消息传递和队列。使用消息传递时，不必过于担心同步、锁定和其他问题。当进程的数量增长时，它往往还能提供更好的扩展。
- 在必须运行在单独进程中的函数内部，不要使用全局变量而应当显式地传递参数。
- 尽量不要在同一个程序中混合使用线程和多线程处理，除非需要大幅提高任务的安全性（或者根据检查代码的角色降低安全性）。
- 特别要注意关闭进程的方式。一般而言，需要显式地关闭进程，并使用一种定义良好的终止模式，而不要仅仅依赖于垃圾回收或者被迫使用terminate() 操作强制终止子进程。
- 管理器和代理的使用与分布式计算中的多个概念密切相关（如分布式对象）。参考与分布式计算的相关书籍会有所帮助。
- multiprocessing 模块源自于名为pyprocessing 的第三方库。搜索关于这个库的使用技巧和信息，会对你有很大帮助。
- 尽管此模块可以工作在Windows上，但还是应该仔细阅读官方文档中的各种微妙细节。例如，要在Windows上启动一个新进程。多进程处理模块实现了它自己的UNIX fork() 函数的克隆版本，其中将通过管道把进程状态复制给子进程。一般而言，此模块更加适用于UNIX系统。
- 最重要的一点是：尽量让事情变得简单。

20.4 threading

threading 模块提供Thread 类和各种同步原语，用于编写多线程的程序。

20.4.1 Thread 对象

Thread 类用于表示单独的控制线程。使用下面的函数可以创建一个新线程：

```
Thread(group
=None, target
=None, name
=None, args
=(), kwargs
={})
```

此函数创建一个新的Thread 实例。 *Group* 的值是None ，为以后的扩展而保留。 *target* 是一个可调用对象，线程启动时，run() 方法将调用此对象，它的默认值是None ，表示不调用任何内容。 *name* 是线程名称。默认将创建一个"Thread-*N* " 格式的唯一名称。 *args* 是传递给 *target* 函数的参数元组。 *kwargs* 是传递给target 的关键字参数的字典。

Thread 实例 *t* 支持以下方法和属性。

```
t
.start()
```

通过在一个单独的控制线程中调用run() 方法，启动线程。此方法只能调用一次。

```
t
.run()
```

线程启动时将调用此方法。默认情况下，它将调用传递到构造函数中的目标函数。还可以在Thread 的子类中重新定义此方法。

```
t
.join([timeout
])
```

等待直到线程终止或者出现超时为止。 *timeout* 是一个浮点数，用于指定以秒为单位的超时时间。线程不能连接自身，而且在线程启动之前就连接它将出现错误。

```
t
.is_alive()
```

如果线程是活动的，返回True ，否则返回False 。从start() 方法返回的那一刻开始，线程就是活动的，直到它的run() 方法终止为止。 *t* .isAlive() 是老式代码中此方法的别名。

```
t
.name
```

线程名称。这个字符串用于唯一识别，可以根据需要将它改为更有意义的值（这样做可以简化调试）。在老式代码中， *t* .getName() 和 *t* .setName(name) 函数用于操作线程名称。

```
t
.ident
```

整数线程标识符。如果线程尚未启动，它的值为None 。

t

.daemon

线程的布尔型后台标志。必须在调用start() 方法之前设置这个标志，它的初始值从创建线程的后台状态继承而来。当不存在任何活动的非后台线程时，整个Python程序将退出。所有程序都有一个主线程，代表初始的控制线程，它不是后台线程。在老式代码中， *t* .setDaemon(*flag*) 和 *t* .isDaemon() 函数用于操作这个值。

下面这个例子说明如何以线程的形式创建和启动一个函数（或其他可调用对象）：

```
import threading
import time

def clock(interval):
    while True:
        print("The time is %s" % time.ctime())
        time.sleep(interval)

t = threading.Thread(target=clock, args=(15,))
t.daemon = True
t.start()
```

下面这个例子说明了如何将同一个线程定义为一个类：

```
import threading
import time

class ClockThread(threading.Thread):
    def __init__(self,interval):
        threading.Thread.__init__(self)
        self.daemon = True
        self.interval = interval
    def run(self):
        while True:
            print("The time is %s" % time.ctime())
            time.sleep(self.interval)

t = ClockProcess(15)
t.start()
```

如果将线程定义为类，并且定义自己的__init__() 方法，必须像上面这样调用基类构造函数Thread.__init__() 。如果忘记这一点，将导致严重错误。除run() 方法之外，改写线程已经定义的其他方法也会出现错误。

这些例子中对于daemon 属性的设置，是永远在后台运行的线程的常见功能。通常，Python在解释器退出之前，会等待所有线程终止。但是，对于不会结束的后台任务来说，这种行为通常是不好的。设置daemon 标志会使解释器在主程序退出后立即退出。在这种情况下，daemonic 线程将被销毁。

20.4.2 Timer 对象

Timer 对象用于在稍后的某个时间执行一个函数。

```
Timer(interval
, func
 [, args
 [, kwargs
]])
```

创建定时器对象，在过去 *interval* 秒时间之后运行函数 *func* 。 *args* 和 *kwargs* 提供传递给 *func* 的参数和关键字参数。在调用start() 方法后才会启动定时器。

Timer 对象 *t* 具有以下方法。

```
t.start()
```

启动定时器。提供给Timer() 方法的函数 *func* 将在指定的时间间隔之后执行。

```
t.cancel()
```

如果函数尚未执行，取消定时器。

20.4.3 Lock 对象

原语锁 （或互斥锁 ）是一个同步原语，状态是“已锁定”或“未锁定”之一。两个方法acquire() 和release() 用于修改锁的状态。如果状态为已锁定，尝试获取锁将被阻塞，直到锁被释放为止。如果有多个线程等待获取锁，当锁被释放时，只有一个线程能获得它。等待线程获得锁的顺序没有定义。

使用下面的构造函数可以创建新的Lock 实例：

```
Lock()
```

创建新的Lock 对象，初始状态为未锁定。

Lock 实例lock 支持以下方法。

```
lock
.acquire([blocking
 ])
```

获取锁，如果有必要，需要阻塞到锁释放为止。如果提供 *blocking* 参数并将它设为False ， 当无法获取锁时将立即返回False ， 如果成功获取锁则返回True 。

```
lock
.release()
```

释放一个锁。当锁处于未锁定状态时，或者从与原本调用acquire() 方法的线程不同的线程调用此方法，将出现错误。

20.4.4 Rlock对象

可重入锁 是一个类似于Lock 对象的同步原语，但同一个线程可以多次获取它。这允许拥有锁的线程执行嵌套的acquire() 和release() 操作。在这种情况下，只有最外面

的release() 操作才能将锁重置为未锁定状态。

使用下面的构造函数可以创建一个新的RLock 对象：

```
RLock()
```

创建新的可重入锁对象。RLock 对象 *rlock* 支持以下方法。

```
rlock
.acquire([blocking
 ])
```

获取锁，如果有必要，需要阻塞到锁被释放为止。如果没有线程拥有锁，它将被锁定，而且递归级别被置为1。如果此线程已经拥有锁，锁的递归级别加1，而且函数立即返回。

```
rlock
.release()
```

通过减少锁的递归级别来释放它。如果在减值后递归级别为0，锁将被重置为未锁定状态。否则，锁将保持已锁定状态。只能由目前拥有锁的线程来调用此函数。

20.4.5 信号量与有边界的信号量

信号量 是一个基于计数器的同步原语，每次调用acquire() 方法时此计数器减1，每次调用release() 方法时此计数器加1。如果计数器为0，acquire() 方法将会阻塞，直到其他线程调用release() 方法为止。

```
Semaphore([value
])
```

创建一个新的信号量。*value* 是计数器的初始值。如果省略此参数，计数器的值将被置为1。

Semaphore 实例 *s* 支持以下方法。

```
s
.acquire([blocking
])
```

获取信号量。如果进入时内部计数器大于0，此方法将把它的值减1，然后立即返回。如果它的值为0，此方法将阻塞，直到另一个线程调用release() 方法为止。 *blocking* 参数的行为与Lock 和RLock 对象中描述的相同。

```
s
.release()
```

通过将内部计数器的值加1来释放一个信号量。如果计数器为0，而且另一个线程正在等待，该线程将被唤醒。如果有多个线程正在等待，只能从它的acquire() 调用返回其中一个。线程释放的顺序并不确定。

```
BoundedSemaphore([value
])
```

创建一个新的信号量。 *Value* 是计数器的初始值。如果省略此参数，计数器的值将被置为1。BoundedSemaphore 的工作方式与Semaphore 完全相同，但release() 操作的次数不能超过acquire() 操作的次数。

信号量与互斥锁之间的微妙差别在于：信号量可用于发信号。例如，可以从不同线程

调用acquire() 和release() 方法，以便在生产者和消费者线程之间进行通信。

```
produced = threading.Semaphore(0)
consumed = threading.Semaphore(1)

def producer():
    while True:
        consumed.acquire()
        produce_item()
        produced.release()

def consumer():
    while True:
        produced.acquire()
        item = get_item()
        consumed.release()
```

这个例子中的信号发出类型经常被条件变量所代替，稍后将讲述条件变量。

20.4.6 事件

事件 用于在线程之间通信。一个线程发出“事件”信号，一个或多个其他线程等待它。Event 实例管理着一个内部标志，可以使用set() 方法将它置为True ，或者使用clear() 方法将它重置为False 。wait() 方法将阻塞，直到标志为Ture 。

```
Event()
```

创建新的Event 实例，并将内部标志置为False 。Event 实例 *e* 支持以下方法。

```
e

.is_set()
```

只有当内部标志为True 时才返回True 。在老式代码中，此方法叫isSet() 。

```
e

.set()
```

将内部标志置为True 。等待它变为True 的所有线程都将被唤醒。

```
e
.clear()
```

将内部标志重置为False 。

```
e
.wait([timeout
])
```

阻塞直到内部标志为True 。如果进入时内部标志为True ，此方法将立即返回。否则，它将阻塞，直到另一个线程调用set() 方法将标志置为True ，或者直到出现可选的超时。 *timeout* 是一个浮点数，用于指定以秒为单位的超时期限。

尽管Event 对象可用于给其他线程发信号，但不应该使用它们来实现在生产者/消费者问题中十分典型的通知。例如，应该避免写出下面这样的代码：

```
evt = Event()

def producer():
    while True:
        # 生产项
        ...
        evt.signal()

def consumer():
    while True:
        # 等待一个项
        evt.wait()
        # 消费项
        ...
        # 清除事件并再次等待
        evt.clear()
```

这段代码不可靠，因为在evt.wait() 和evt.clear() 操作之间，生产者可能生产了一个新项。但是，通过清除事件，在生产者创建一个新项之前，消费者可能看不到这个新项。最好的情况是，程序将经过一段很短的停滞，对项的处理被莫名其妙推迟了。最坏的情况是，由于事件信号丢失，整个程序将会挂起。要解决这类问题，最好使用条件变量。

20.4.7 条件变量

条件变量 是构建在另一个锁上的同步原语，当需要线程关注特定的状态变化或事件的发生时将使用这个锁。典型的用法是生产者-消费者问题，其中一个线程生产的数据供另一个线程使用。使用下面的构造函数可以创建新的Condition 实例：

```
Condition([lock
])
```

创建新的条件变量。 *lock* 是可选的Lock 或RLock 实例。如果未提供 *lock* 参数，就会创建新的RLock 实例供条件变量使用。

条件变量 *cv* 支持以下方法。

```
cv
.acquire(*args
)
```

获取底层锁。此方法将调用底层锁上对应的acquire(*args) 方法。

```
cv
.release()
```

释放底层锁。此方法将调用底层锁上对应的release() 方法。

```
cv
.wait([timeout
])
```

等待直到获得通知或出现超时为止。此方法在调用线程已经获取锁之后调用。调用时，将释放底层锁，而且线程将进入睡眠状态，直到另一个线程在条件变量上执行`notify()`或`notifyAll()`方法将其唤醒为止。在线程被唤醒之后，线程将重新获取锁，方法也会返回。 *timeout* 是浮点数，单位为秒。如果这段时限耗尽，线程将被唤醒，重新获取锁，而控制将被返回。

```
cv
.notify([n])
```

唤醒一个或多个等待此条件变量的线程。此方法只会在调用线程已经获取锁之后调用，而且如果没有正在等待的线程，它就什么也不做。 *n* 指定要唤醒的线程数量，默认为1。被唤醒的线程在它们重新获取锁之前不会从`wait()`调用返回。

```
cv
.notify_all()
```

唤醒所有等待此条件的线程。此方法在老式代码中叫`notifyAll()`。

下面这个例子提供了使用条件变量的模板：

```
cv = threading.Condition()
def producer():
    while True:
        cv.acquire()
        produce_item()
        cv.notify()
        cv.release()
def consumer():
```

```
    while True:
        cv.acquire()
        while not item_is_available():
            cv.wait() # 等待项出现
        cv.release()
        consume_item()
```

使用条件变量时需要注意的是，如果存在多个线程等待同一个条件，notify() 操作可能唤醒它们中的一个或多个（这种行为通常取决于底层的操作系统）。因此，始终有这样的可能：某个线程被唤醒后，却发现它等待的条件不存在了。这解释了为什么在consumer() 函数中使用while 循环。如果线程醒来，但是生成的项已经消失，它就会回去等待下一个信号。

20.4.8 使用Lock

使用诸如Lock 、RLock 或Semphore 之类的锁原语时，必须多加小心。锁的错误管理经常导致死锁或竞争条件。依赖锁的代码应该保证当出现异常时正确地释放锁。典型的代码如下所示：

```
try:
    lock.acquire()
    # 关键部分
    statements


    ...
finally:
    lock.release()
```

另外，所有种类的锁还支持上下文管理协议（写起来更简洁些）：

```
with lock:
    # 关键部分
    statements


    ...
```

在最后一个例子中，with 语句自动获取锁，并且在控制流离开上下文时自动释放锁。

此外，编写代码时一般应该避免同时获取多个锁，例如：

```
with lock_A:
    # 关键部分
    statements
```

```
    ...
    with lock_B:
         # B的关键部分
         statements


    ...
```

这通常很容易导致应用程序神秘死锁。尽管有几种策略可以避免出现这种情况（如分层锁定），但最好在编写代码时就避免这种做法。

20.4.9 线程终止与挂起

线程没有任何方法可用于强制终止或挂起。这是设计上的原因，因为编写线程程序本身十分复杂。例如，如果某个线程已经获取了锁，在它能够释放锁之前强制终止或挂起它，将导致整个应用程序出现死锁。此外，终止时一般不能简单地“释放所有的锁”，因为复杂的线程同步经常涉及锁定和解除锁定操作，而这些操作在执行时的次序要十分精确。

如果要为终止或挂起提供支持，需要自己构建这些功能。一般的做法是在循环中运行线程，这个循环的作用是定期检查线程状态以决定它是否应该终止。例如：

```
class StoppableThread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__()
        self._terminate    = False
        self._suspend_lock = threading.Lock()
    def terminate(self):
        self._terminate = True
    def suspend(self):
        self._suspend_lock.acquire()
    def resume(self):
        self._suspend_lock.release()
    def run(self):
        while True:
             if self._terminate:
              break
            self._suspend_lock.acquire()
            self._suspend_lock.release()
            statements




...
```

要记住，要让这种方法可靠地工作，线程应该千万小心不要执行任何类型的阻塞I/O操作。例如，如果线程阻塞等待数据到达，那么它会直到该操作唤醒它时才会终止。因此，你很可能需要在实现中使用超时、非阻塞I/O和其他高级功能，从而确保终止检查执行的频率是足够的。

20.4.10 实用工具函数

可用的实用工具函数如下。

active_count()

返回当前活动的Thread 对象数量。

current_thread()

返回对应于调用者的控制线程的Thread 对象。

enumerate()

列出当前所有活动的Thread 对象。

local()

返回local 对象，用于保存线程本地的数据。应该保证此对象在每个线程中是唯一的。

setprofile(*func*

)

设置一个配置文件函数，用于已创建的所有线程。 *func* 在每个线程开始运行之前被传递给sys.setprofile() 函数。

```
settrace(func
)
```

设置一个跟踪函数，用于已创建的所有线程。 *func* 在每个线程开始运行之前被传递给sys. settrace() 函数。

```
stack_size([size
])
```

返回创建新线程时使用的栈大小。可选的整数参数 *size* 表示创建新线程时使用的栈大小。 *size* 的值可以是32 768（32 KB）或更大，而且是4 096（4 KB）的倍数，这样可移植性更好。如果系统上不支持此操作，将引发ThreadError 异常。

20.4.11 全局解释器锁

Python解释器被一个锁保护，只允许一次执行一个线程，即便存在多个可用的处理器。在计算密集型程序中，这严重限制了线程的作用。事实上，在计算密集型程序中使用线程，经常比仅仅按照顺序执行同样的工作慢得多。因此，实际上应该只在主要关注I/O的程序，如网络服务器中使用线程。对于计算密集程度更高的任务，最好使用C扩展模块或multiprocessing 模块来代替。C扩展具有释放解释器锁和并行运行的选项，前提是释放锁时不与解释器进行交互。multiprocessing 模块将工作分派给不受锁限制的单独子进程。

20.4.12 使用线程编程

尽管在Python中可以使用各种锁和同步原语的组合编写非常传统的多线程程序，但有一种首推的编程方式要优于其他所有编程方式——即将多线程程序组织为多个独立任务的集合，这些任务之间通过消息队列进行通信。下一节（queue 模块）中将举例说明这种方式。

20.5 queue 、Queue

queue 模块（在Python 2中叫Queue ）实现了各种多生产者-多消费者队列，可用于在执行的多个线程之间安全地交换信息。

queue 模块定义了3种不同的队列类。

```
Queue([maxsize
])
```

创建一个FIFO（first-in first-out，先进先出）队列。 *maxsize* 是队列中可以放入的项的最大数量。如果省略 *maxsize* 参数或将它置为0，队列大小将为无穷大。

```
LifoQueue([maxsize
])
```

创建一个LIFO（last-in first-out，后进先出）队列（也叫栈）。

```
PriorityQueue([maxsize
])
```

创建一个优先级队列，其中项按照优先级从低到高依次排好。使用这种队列时，项应该是(*priority, data*)形式的元组，其中 *priority* 是一个数字。

队列类的实例 *q* 具有以下方法。

```
q
.qsize()
```

返回队列的正确大小。因为其他线程可能正在更新队列，此方法返回的数字不完全可靠。

```
q.empty()
```

如果队列为空，返回True ， 否则返回False 。

```
q
.full()
```

如果队列已满，返回True ， 否则返回False 。

```
q
.put(item
 [, block
 [, timeout
]])
```

将item 放入队列。如果可选参数 *block* 为True （默认值），调用者将被阻塞直到队列中出现可用的空闲位置为止。否则（ *block* 为False ），队列满时将引发Full 异常。 *timeout* 提供可选的超时值，单位为秒。如果出现超时，将引发Full 异常。

```
q
.put_nowait(item
)
```

等价于 *q* .put(*item, False*) 方法。

```
q
```

```
.get([block
 [, timeout
]])
```

从队列中删除一项，然后返回这个项。如果可选参数 *block* 为True （默认值），调用者将阻塞，直到队列中出现可用的空闲位置。否则（ *block* 为False ），队列为空时将引发Empty 异常。 *timeout* 提供可选的超时值，单位为秒。如果出现超时，将引发Empty 异常。

```
q
.get_nowait()
```

等价于get(0) 方法。

```
q
.task_done()
```

队列中数据的消费者用来指示对于项的处理已经结束。如果使用此方法，那么从队列中删除的每一项都应该调用一次。

```
q
.join()
```

阻塞直到队列中的所有项均被删除和处理为止。一旦为队列中的每一项都调用了一次 *q* .task_done() 方法，此方法将会直接返回。

使用队列的线程示例

使用队列一般可以简化多线程的程序。例如，可以使用共享队列将线程连接在一起，而不必依赖于必须由锁保护的共享状态。在这种模型中，工作者线程一般充当数据的消费者。下面这个例子说明了这个概念：

```
import threading
from queue import Queue # Python 2中使用from Queue

class WorkerThread(threading.Thread):
    def __init__(self,*args,**kwargs):
        threading.Thread.__init__(self,*args,**kwargs)
        self.input_queue = Queue()
    def send(self,item):
        self.input_queue.put(item)
    def close(self):
        self.input_queue.put(None)
        self.input_queue.join()
    def run(self):
        while True:
            item = self.input_queue.get()
            if item is None:
                break
            # 处理项（使用有用的工作代替）
            print(item)
            self.input_queue.task_done()
        # 完成。指示收到和返回了哨兵
        self.input_queue.task_done()
        return

# 使用示例
w = WorkerThread()
w.start()
w.send("hello")    # 将项发送给工作线程（通过队列）
w.send("world")
w.close()
```

这个类的设计经过了仔细挑选。首先应该注意到，编程接口是`multiprocessing` 模块中管道创建的`Connection` 对象的一个子集。这支持在未来进行扩展。例如，可以把工作者合并到一个单独的进程中，同时不会影响到负责给它们发送数据的代码。

其次，编程接口支持线程终止。`close()` 方法在队列上安置了一个哨兵，它将导致线程在处理完毕后关闭。

最后，编程接口还几乎与协程完全相同。如果要执行的工作不涉及任何阻塞操作，可以将`run()` 方法重新实现为协程，这就省却了使用线程的麻烦。后一种方法的运行速度可能更快，因为节省了线程上下文切换带来的开销。

20.6 协程与微线程

在某些类型的应用程序中，可以使用一个任务调度器和一些生成器或协程实现协作式用户空间多线程。这有时称为微线程 ，但使用的术语还有很多种，如叫tasklet、绿色线

程、greenlet等。这种技术的一种常见用法是在需要管理大量的已打开文件或套接字的程序中。例如，一台需要同时管理1 000个客户端连接的网络服务器。此时的解决方案是联合使用异步I/O或轮询（使用`select` 模块）与处理I/O事件的任务调度器，而不是创建1 000个线程。

这种编程技术的基础概念是这样产生的：生成器或协程函数中的`yield` 语句挂起函数的执行，直到稍后使用`next()` 或`send()` 操作进行恢复为止。这样就可以使用一个调度器循环在一组生成器函数之间协作多个任务。如下例所示：

```
def foo():
    for n in xrange(5):
        print("I'm foo %d" % n)
        yield

def bar():
    for n in xrange(10):
        print("I'm bar %d" % n)
        yield

def spam():
    for n in xrange(7):
        print("I'm spam %d" % n)
        yield

# 创建并填充一个任务队列
from collections import deque
taskqueue = deque()
taskqueue.append(foo())    # 添加一些任务（生成器）
taskqueue.append(bar())
taskqueue.append(spam())

# 运行所有任务
while taskqueue:
     # 获得下一个任务
     task = taskqueue.pop()
     try:
         # 运行它直到下一条yield和enqueue语句
         next(task)
         taskqueue.appendleft(task)
     except StopIteration:
         # 任务完成
         pass
```

像上面这样让程序定义一系列CPU密集型协程并调度它们的做法并不常见。相反地，看到这个技巧与I/O密集型任务、轮询或事件处理一起使用的机会可能更大。在第21章的21.4节中可以找到说明此技术的高级例子。

---

① 如果看不明白，请把回答中的5个字组合成一个句子，再想一下和多线程的关系。——编者注

第21章　网络编程和套接字

本章介绍用来实现底层网络服务器和客户端的模块。Python在网络方面提供了广泛支持，从直接使用套接字编程到处理高级应用程序协议（如HTTP）。首先，我们将非常简要地介绍网络编程。建议读者参考W.Richard Stevens的UNIX Network Programming，Volume 1：Networking APIs：Sockets and XTI（Prentice Hall）[①] 了解更多信息。第22章将详细介绍与应用层协议有关的模块。

21.1　网络编程基础

Python的网络编程模块主要支持两种网络协议：TCP和UDP。TCP协议 是一种面向连接的可靠协议，用于建立机器之间的双向通信流。UDP是一个较底层的、以数据包为基础的协议（无连接传输模式），由机器发送和接收分离的信息包，不需要建立正式的连接。与TCP不同，UDP通信是不可靠的，在需要可靠通信的应用程序中难以管理。因此，大部分网络应用程序都采用TCP连接。

这两种网络协议都通过一种名为套接字的编程抽象进行处理。套接字 是类似于文件的对象，使程序能够接受传入连接，进行传出连接，发送和接收数据。在两台机器进行通信之前，它们都必须创建套接字对象。

接收连接的机器（服务器）必须将其套接字对象绑定到一个已知端口号。端口 是由操作系统管理的16位数字，范围从0到65 535，客户端使用端口作为服务器的唯一标识符。系统保留端口0～1 023供通用网络协议使用。下面列出了为一些通用协议分配的端口。（完整的列表参见http://www.iana.org/assignments/port-numbers 。）

服　　务	端　口　号	服　　务	端　口　号
FTP-Data	20	HTTP (WWW)	80
FTP-Control	21	POP3	110
SSH	22	IMAP	143
Telnet	23	HTTPS (Secure WWW)	443
SMTP (Mail)	25		

在建立TCP连接之前，服务器和客户端需要严格执行一系列步骤，如图21-1所示。

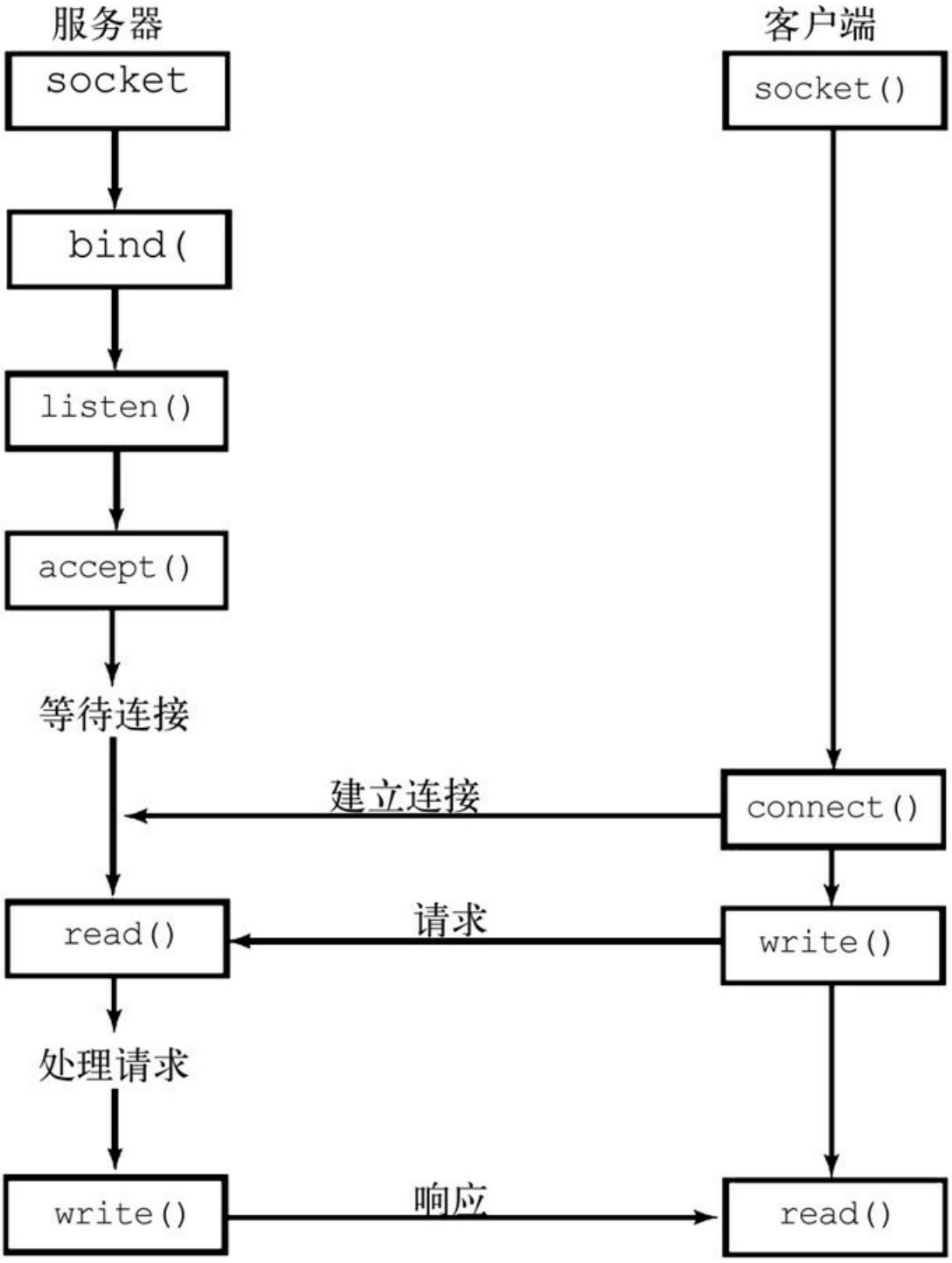


图21-1　TCP连接协议

在TCP服务器中，用来接收连接的套接字对象与用来执行客户端后续通信的套接字对象是不同的。具体来说，`accept()`系统调用返回实际用来连接的新套接字对象。这样一来，服务器就可以同时管理大量客户端连接。

执行UDP连接的方式大致相同，不同之处在于客户端和服务器之间不建立“连接”，如

图21-2所示。

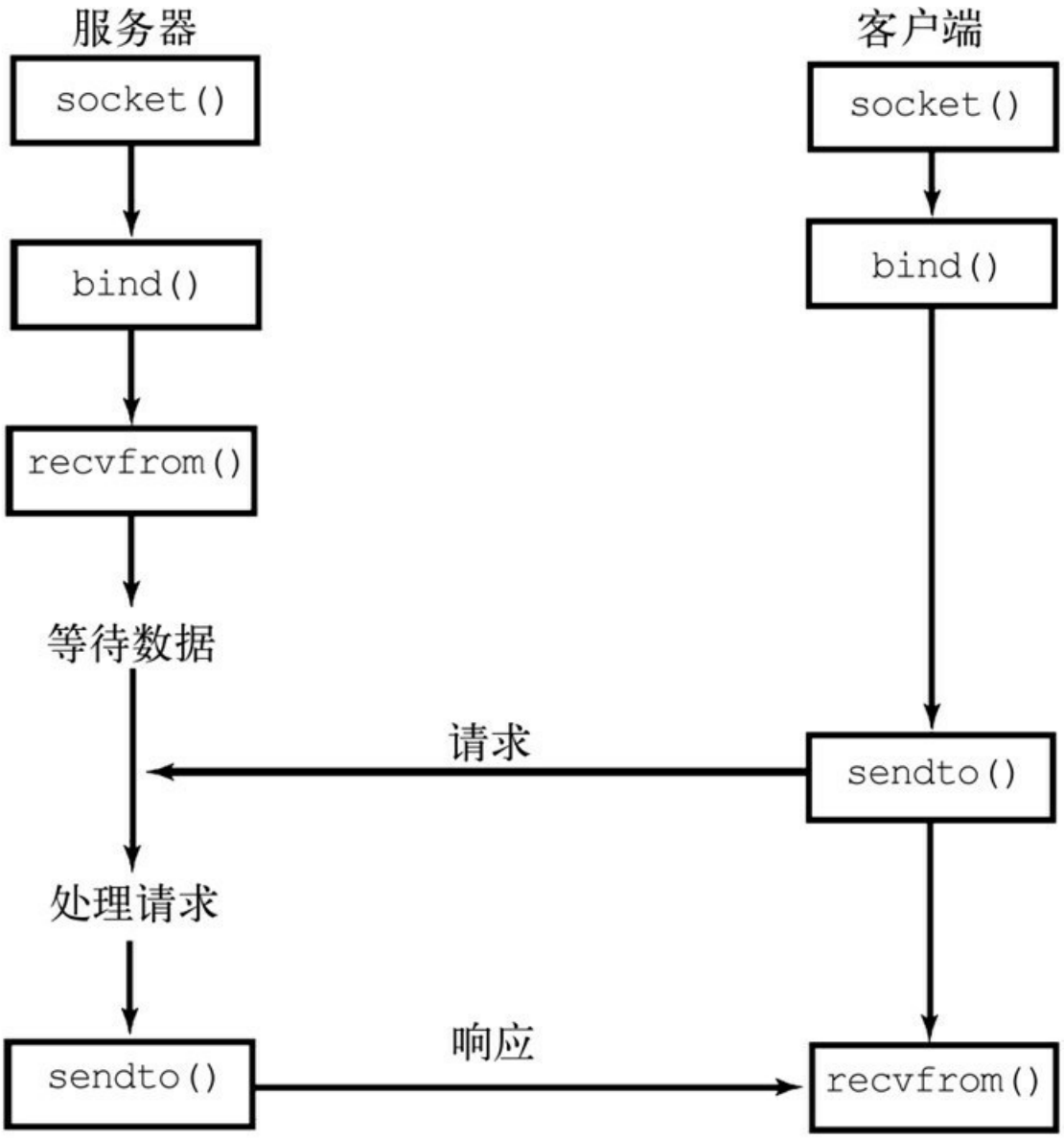


图21-2　UDP连接协议

下例通过使用socket 模块编写的客户端和服务器，对TCP协议进行了说明。在本例中，服务器以字符串形式向服务器返回当前时间。

```
# 时间服务器程序
from socket import *
import time

s = socket(AF_INET, SOCK_STREAM)  # 创建TCP套接字
s.bind(('',8888))                 # 绑定到端口8888
s.listen(5)                       # 监听，但只能挂起5个以下的连接。

while True:
```

```
    client,addr = s.accept()    # 连接
    print("Got a connection from %s" % str(addr))
    timestr = time.ctime(time.time()) + "\r\n"
    client.send(timestr.encode('ascii'))
    client.close()
```

以下是客户端程序：

```
# 时间客户端程序
from socket import *
s = socket(AF_INET,SOCK_STREAM)  # 创建TCP套接字
s.connect(('localhost', 8888))   # 连接到服务器
tm = s.recv(1024)                 # 最多接收1024个字节
s.close()
print("The time is %s" % tm.decode('ascii'))
```

本章后面的socket 模块一节将介绍建立UDP连接的示例。

网络协议经常以文本的形式交换数据。但是，对于文本编码要尤其小心。在Python 3中，所有字符串都是Unicode编码的。因此，任何类型的文本字符串如果需要通过网络发送，都必须进行编码。这也是服务器对它传输的数据使用encode('ascii') 方法的原因。同样，客户端收到网络数据时，最初接收的是原始的未编码字节形式。如果你将其打印出来或当作文本进行处理，很可能得不到想要的结果。因此，首先必须进行解码。这就是客户端代码对结果使用decode('ascii') 的原因。

下面将介绍与套接字编程有关的模块。第22章将介绍支持各种网络应用程序（如电子邮件和Web）的高级模块。

21.2 asynchat

Asynchat 模块简化了使用asyncore 模块实现异步网络的应用程序的实现过程。它将asyncore 的底层I/O功能封装进了一个高级编程接口，该接口是专门针对基于简单请求/响应机制的网络协议（如HTTP）而设计的。

要使用该模块，必须定义继承自async_chat 的类。在该类中，必须定义两个方法：collect_incoming_data() 和found_terminator() 。前一个方法在网络连接收到数据时调用。通常，该方法只是提取数据并将其存储在某处。found_terminator() 方法在检测到请求结束时调用。例如，在HTTP中，空白行表示请求结束。

在数据输出方面，async_chat 有一个生产者FIFO队列。如果需要输出数据，要输出的数据将被添加到该队列。然后，只要网络连接上可以进行写入时，就可以透明地从该队列中取出数据。

```
async_chat([sock

])
```

用来定义新处理程序的基类。async_chat 继承自asyncore.dispatcher ，提供的方法相同。 *sock* 是用于通信的套接字对象。

除了asyncore.dispatcher 基类提供的方法之外，async_chat 的实例 *a* 还具有以下方法。

```
a
.close_when_done()
```

将None 推入生成函数FIFO队列，表示传出数据流已经到达文件结尾。写入程序收到该信号时将关闭通道。

```
a
.collect_incoming_data(data
)
```

通道收到数据时调用该方法。 *data* 是收到的数据，通常保存起来以便进行后续处理。用户必须实现该方法。

```
a
.discard_buffers()
```

丢弃输入/输出缓冲区和生产者FIFO队列中保存的所有数据。

```
a
```

```
.found_terminator()
```

set_terminator() 设置终止状态时调用该方法。该方法必须由用户实现。通常，它会处理此前collect_incoming_data() 方法收集的数据。

```
a
.get_terminator()
```

返回通道的终止符。

```
a
.push(data
)
```

将数据加入到通道的传出生产者FIFO队列。 *data* 是包含要发送数据的字符串。

```
a
.push_with_producer(producer
)
```

将生产者对象 *producer* 加入到生产者FIFO队列。 *producer* 可以是任何具有简单方法more() 的对象。more() 方法应该在每次调用时生成一个字符串。返回空字符串表示到达数据末尾。在内部，async_chat 类重复调用more() 获取数据，以写入传出通道。重复调用push_with_producer() 可以将多个生产者对象推入FIFO队列。

```
```

```
s
.set_terminator(term
)
```

在通道上设置终止状态。 *term* 可以是字符串、整数或者None 。如果 *term* 是字符串，则在输入流出现该字符串时调用found_terminator() 方法。如果 *term* 是整数，则它指定字节计数。读取该数指定的字节之后，将调用found_terminator() 方法。如果 *term* 是None ，则持续收集数据。

该模块定义了一个为a.push_with_producer() 方法生成数据的类。

```
simple_producer(data
 [, buffer_size
])
```

创建简单的生产者对象，从字节字符串 *data* 生成数据块。 *buffer_size* 指定数据块大小，默认值为512。

asynchat 模块总是与asyncore 模块一起使用。例如，asyncore 用来设置接受传入连接的高级服务器，然后使用asynchat 实现每个连接的处理程序。为了演示工作原理，我们在下面的示例中实现了处理GET 请求的简版Web服务器。该例忽略了很多错误校验和细节问题，但应该足以说明问题。读者可以比较该例与下一节asyncore 模块中的示例。

```
# 使用asynchat的异步HTTP服务器
import asynchat, asyncore, socket
import os
import mimetypes
try:
    from http.client import responses         # Python 3
except ImportError:
    from httplib import responses             # Python 2
# 该类继承自asyncore模块，仅处理接受的事件
class async_http(asyncore.dispatcher):
    def __init__(self,port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET,socket.SOCK_STREAM)
        self.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.bind(('',port))
        self.listen(5)

    def handle_accept(self):
        client,addr = self.accept()
```

```
        return async_http_handler(client)

# 处理异步HTTP请求的类。
class async_http_handler(asynchat.async_chat):
    def __init__(self,conn=None):
        asynchat.async_chat.__init__(self,conn)
        self.data = []
        self.got_header = False
        self.set_terminator(b"\r\n\r\n")

    # 获取传入数据并添加到数据缓冲区
    def collect_incoming_data(self,data):
        if not self.got_header:
            self.data.append(data)

    # 到达终止符（空白行）
    def found_terminator(self):
        self.got_header = True
        header_data      = b"".join(self.data)
        # 将报头数据（二进制）解码为文本以便进一步处理
        header_text      = header_data.decode('latin-1')
        header_lines     = header_text.splitlines()
        request          = header_lines[0].split()
        op               = request[0]
        url              = request[1][1:]
        self.process_request(op,url)

    # 将文本加入到传出流，但首先要编码
    def push_text(self,text):
        self.push(text.encode('latin-1'))

    # 处理请求
    def process_request(self, op, url):
        if op == "GET":
            if not os.path.exists(url):
                self.send_error(404,"File %s not found\r\n")
            else:
                type, encoding = mimetypes.guess_type(url)
                size = os.path.getsize(url)
                self.push_text("HTTP/1.0 200 OK\r\n")
                self.push_text("Content-length: %s\r\n" % size)
                self.push_text("Content-type: %s\r\n" % type)
                self.push_text("\r\n")
                self.push_with_producer(file_producer(url))
        else:
            self.send_error(501,"%s method not implemented" % op)
        self.close_when_done()
    # 错误处理
    def send_error(self,code,message):
        self.push_text("HTTP/1.0 %s %s\r\n" % (code, responses[code]))
        self.push_text("Content-type: text/plain\r\n")
        self.push_text("\r\n")
        self.push_text(message)

class file_producer(object):
    def __init__(self,filename,buffer_size=512):
        self.f = open(filename,"rb")
        self.buffer_size = buffer_size
    def more(self):
```

```
        data = self.f.read(self.buffer_size)
        if not data:
            self.f.close()
        return data

a = async_http(8080)
asyncore.loop()
```

要测试该例，你需要提供一个URL，并且该URL必须对应于运行服务器的那个目录中的文件。

21.3 asyncore

asyncore 模块用来构建网络应用程序，在使用该方法构建的应用程序中，网络活动将作为一系列由事件循环（使用select() 系统调用构建）分派的事件进行异步处理。这种方法在希望提供并发性但又无法使用线程（或进程）的网络程序中非常有用。它还可以提高短途传输的性能。该模块的所有功能都由dispatcher 类提供，dispatcher 类内部封装了一个普通套接字对象。

```
dispatcher([sock

])
```

定义事件驱动型非阻塞套接字对象的基类。 *sock* 是现有的套接字对象。如果忽略该参数，则必须使用create_socket() 方法（稍后介绍）创建套接字。创建套接字之后，即可使用特殊的处理程序方法处理网络事件。此外，所有公开分派程序对象将存储在一个内部列表中，供众多轮询函数使用。

我们调用dispatcher 类的以下方法处理网络事件。这些方法应该在继承自dispatcher的类中定义。

```
d

.handle_accept()
```

收到新连接时对监听的套接字调用该方法。

```
```

```
d
.handle_close()
```

关闭套接字时调用该方法。

```
d
.handle_connect()
```

进行连接时调用该方法。

```
d
.handle_error()
```

发生未捕获的Python异常时调用该方法。

```
d
.handle_expt()
```

收到套接字带外数据（out-of-band data）时调用该方法。

```
d
.handle_read()
```

可以从套接字读取新数据时调用该方法。

```
d
.handle_write()
```

尝试写入数据时调用该方法。

```
d
.readable()
```

select() 循环使用该函数查看对象是否准备读取数据。如果是，则返回True ； 否则返回False 。调用该方法可以查看是否应该使用新数据调用handle_read() 方法。

```
d
.writable()
```

select() 循环使用该函数查看对象是否想写入数据。如果是，则返回True ； 否则返回False 。可以调用该方法确定是否应该调用handle_write() 方法来生成输出。

除了上文介绍的方法之外，还可以使用以下方法来执行一些底层的套接字操作。这些方法类似于套接字对象上的方法。

```
d
.accept()
```

接受连接，返回(*client, addr*)，其中 *client* 是用来通过连接发送和接收数据

的套接字对象，addr 是客户端的地址。

```
d
.bind(address
)
```

将套接字绑定到 *address* 。 *address* 通常是一个元组(*host, port*)，具体取决于要使用的地址族。

```
d
.close()
```

关闭套接字。

```
d
.connect(address
)
```

建立连接。 *address* 是一个元组(*host, port*)。

```
d
.create_socket(family
, type
)
```

新建套接字。参数与socket.socket() 的参数相同。

```
d
.listen([backlog
])
```

监听传入连接。 *backlog* 是传递给底层socket.listen() 函数的整数。

```
d
.recv(size
)
```

最多接收 *size* 个字节。空字符串表示客户端已经关闭了通道。

```
d
.send(data
)
```

发送数据。 *data* 是字节字符串。

以下函数用于启动事件循环和处理事件。

```
loop([timeout
 [, use_poll
 [, map
```

```
 [, count

]]]])
```

无限轮询事件。使用select() 函数进行轮询，但如果 *use_poll* 参数为 *True* ，则使用poll() 进行轮询。 *timeout* 表示超时期，默认情况下设置为30秒。 *map* 是一个字典，包含所有要监视的通道。 *count* 指定返回之前要执行的轮询操作次数。如果 *count* 为None （默认值），则loop() 将一直轮询，直到所有通道关闭。如果 *count* 为1，则函数只执行一次事件轮询即返回。

示例

下例使用asyncore 实现一个非常简单的Web服务器。它实现两个类：用于接受连接的asynhttp 和用于处理客户端请求的asynclient 。应该将本例与asynchat 模块中介绍的示例比较学习。两者的主要区别在于：本例更偏底层——需要我们将输入流分解成行，缓冲过量的数据并识别终止请求报头的空白行。

```
# 一台异步HTTP服务器
import asyncore, socket
import os
import mimetypes
import collections
try:
    from http.client import responses       # Python 3
except ImportError:
    from httplib import responses           # Python 2

# 该类仅处理接受事件
class async_http(asyncore.dispatcher):
    def __init__(self,port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET,socket.SOCK_STREAM)
        self.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.bind(('',port))
        self.listen(5)

    def handle_accept(self):
        client,addr = self.accept()
        return async_http_handler(client)

# 处理客户端
class async_http_handler(asyncore.dispatcher):
    def __init__(self, sock = None):
        asyncore.dispatcher.__init__(self,sock)
        self.got_request    = False              # 是否读取HTTP请求？
        self.request_data   = b""
        self.write_queue    = collections.deque()
        self.responding     = False

    # 仅在未读取请求报头时才能读取
    def readable(self):
```

```
        return not self.got_request
# 读取传入请求数据
def handle_read(self):
    chunk = self.recv(8192)
    self.request_data += chunk
    if b'\r\n\r\n' in self.request_data:
        self.handle_request()

# 处理传入请求
def handle_request(self):
    self.got_request = True
    header_data  = self.request_data[:self.request_data.find(b'\r\n\r\n')]
    header_text  = header_data.decode('latin-1')
    header_lines = header_text.splitlines()
    request      = header_lines[0].split()
    op      = request[0]
    url     = request[1][1:]
    self.process_request(op,url)

# 处理请求
def process_request(self,op,url):
    self.responding = True
    if op == "GET":
        if not os.path.exists(url):
            self.send_error(404,"File %s not found\r\n" % url)
        else:
            type, encoding = mimetypes.guess_type(url)
            size = os.path.getsize(url)
            self.push_text('HTTP/1.0 200 OK\r\n')
            self.push_text('Content-length: %d\r\n' % size)
            self.push_text('Content-type: %s\r\n' % type)
            self.push_text('\r\n')
            self.push(open(url,"rb").read())
    else:
        self.send_error(501,"%s method not implemented" % self.op)

# 错误处理
def send_error(self,code,message):
    self.push_text('HTTP/1.0 %s %s\r\n' % (code, responses[code]))
    self.push_text('Content-type: text/plain\r\n')
    self.push_text('\r\n')
    self.push_text(message)

# 将二进制数据添加到输出队列
def push(self,data):
    self.write_queue.append(data)

# 将文本数据添加到输出队列
def push_text(self,text):
    self.push(text.encode('latin-1'))

# 仅在响应准备好时才能写入
def writable(self):
    return self.responding and self.write_queue

# 写入响应数据
def handle_write(self):
    chunk = self.write_queue.popleft()
    bytes_sent = self.send(chunk)
    if bytes_sent != len(chunk):
```

```
        self.write_queue.appendleft(chunk[bytes_sent:])
    if not self.write_queue:
        self.close()
# 创建服务器
a = async_http(8080)
# 持续轮询
asyncore.loop()
```

另请参见： socket（21.5节），select（21.4节），http（22.2节），SocketServer（21.7节）。

21.4 select

select 模块可以访问select() 和poll() 系统调用。select() 通常用来实现轮询，也可以在不使用线程或子进程的情况下跨多个输入/输出流进行多重处理。在UNIX中，它可以用于文件、套接字、管道以及其他大部分文件类型。在Windows中，它只能用于套接字。

```
select(iwtd
, owtd
, ewtd
 [, timeout
])
```

查询一组文件描述符的输入、输出和异常状态。前3个参数都是列表，包含整数文件描述符，或者带有可返回文件描述符的fileno() 方法的对象。 *iwtd* 参数指定等待输入的对象， *owtd* 指定等待输出的对象， *ewtd* 指定等待异常情况的对象。这3个列表都可以是空列表。 *timeout* 是一个浮点数，指定以秒为单位的超时期。如果忽略timeout，该函数将等待，直到至少准备好一个文件描述符时为止。如果它为0，函数将仅执行一次轮询并立即返回。返回值是一个列表元组，其中包含准备就绪的对象。这些对象是前3个参数的子集。如果在到达超时期之前没有对象准备就绪，那么将返回3个空列表。如果发生错误，那么将触发select.error 异常。它的值与IOError 和OSError 返回的值相同。

```
poll()
```

创建利用poll() 系统调用的轮询对象。仅在支持poll() 的系统中可用。

poll() 返回的轮询对象p 支持以下方法。

```
p.register
(fd
 [, eventmask
])
```

注册新的文件描述符 *fd* 。 *fd* 要么是一个整数文件描述符，要么是一个带有可以获取描述符的fileno() 方法的对象。 *eventmask* 是以下标志的“按位或”，这些标志指示要处理的事件。

常　　量	描　　述	常　　量	描　　述
POLLIN	可用于读取的数据	POLLERR	错误情况
POLLPRI	可用于读取的紧急数据	POLLHUP	保持状态
POLLOUT	准备写入	POLLNVAL	无效请求

如果忽略eventmask ， 则检查POLLIN 、POLLPRI 和POLLOUT 事件。

```
p
.unregister(fd
)
```

从轮询对象中删除文件描述符 *fd* 。如果文件没有注册，则出现KeyError 。

```
p
```

```
.poll([timeout

])
```

对所有已注册的文件描述符事件进行轮询。 *timeout* 是以毫秒为单位的可选超时时间。返回列表元组(*fd, event*)，其中 *fd* 是文件描述符， *event* 是指示事件的位掩码。该位掩码的字段对应于常量POLLIN 、POLLOUT 等。例如，要检查POLLIN 事件，只需使用event & POLLIN 测试值即可。如果返回空列表，则表示到达超时值且没有发生任何事件。

21.4.1 高级模块功能

select() 和poll() 函数是该模块定义的函数中可移植性最高的。在Linux系统中，select 模块还提供了一个接口用作外围和级别触发器轮询（epoll）接口，该接口能显著提高性能。在BSD系统中，该模块可以访问内核队列和事件对象。关于这些编程接口，请参考http://docs.python.org/library/select 上有关select 的在线文档。

21.4.2 高级异步I/O示例

select 模块有时可用来实现基于小任务（tasklet）或协程的服务器，协程是一种在不使用线程或进程的情况下提供并发执行的技术。为了演示这个概念，在以下高级示例中，我们将为协程实现一个基于I/O的任务调度程序。提示：这是本书中最高级的示例，需要费一番脑筋才能弄懂。关于更多参考资料，可以参考我的PyCON'09教程"*A Curious Course on Coroutines and Concurrency* "（http://www.dabeaz.com/coroutines ）。

```
import select
import types
import collections
# 表示运行任务的对象
class Task(object):
    def __init__(self,target):
        self.target  = target    # 一个协程
        self.sendval = None      # 恢复时要发送的值
        self.stack   = []        # 调用栈
    def run(self):
        try:
            result = self.target.send(self.sendval)
            if isinstance(result,SystemCall):
                return result
            if isinstance(result,types.GeneratorType):
                self.stack.append(self.target)
                self.sendval = None
                self.target = result
            else:
                if not self.stack: return
                self.sendval = result
                self.target = self.stack.pop()
        except StopIteration:
            if not self.stack: raise
```

```
            self.sendval = None
            self.target = self.stack.pop()

# 表示“系统调用”的对象
class SystemCall(object):
    def handle(self,sched,task):
        pass

# 调度程序对象
class Scheduler(object):
    def __init__(self):
        self.task_queue     = collections.deque()
        self.read_waiting   = {}
        self.write_waiting  = {}
        self.numtasks       = 0

    # 通过协程新建任务
    def new(self,target):
        newtask = Task(target)
        self.schedule(newtask)
        self.numtasks += 1

    # 将任务放入任务队列
    def schedule(self,task):
        self.task_queue.append(task)

    # 让任务等待文件描述符上的数据
    def readwait(self,task,fd):
        self.read_waiting[fd] = task
# 让任务等待写入文件描述符
def writewait(self,task,fd):
    self.write_waiting[fd] = task

# 调度程序主循环
def mainloop(self,count=-1,timeout=None):
    while self.numtasks:
        # 检查要处理的I/O事件
        if self.read_waiting or self.write_waiting:
            wait = 0 if self.task_queue else timeout
            r,w,e = select.select(self.read_waiting, self.write_waiting, [],
                                  wait)
            for fileno in r:
                self.schedule(self.read_waiting.pop(fileno))
            for fileno in w:
                self.schedule(self.write_waiting.pop(fileno))

        # 运行队列上准备运行的所有任务
        while self.task_queue:
            task = self.task_queue.popleft()
            try:
                result = task.run()
                if isinstance(result,SystemCall):
                    result.handle(self,task)
                else:
                    self.schedule(task)
            except StopIteration:
                self.numtasks -= 1

        # 如果没有任务可以运行，是等待还是返回
```

```
        else:
            if count > 0: count -= 1
            if count == 0:
                return

# 实现不同的系统调用
class ReadWait(SystemCall):
    def __init__(self,f):
        self.f = f
    def handle(self,sched,task):
        fileno = self.f.fileno()
        sched.readwait(task,fileno)

class WriteWait(SystemCall):
    def __init__(self,f):
        self.f = f
    def handle(self,sched,task):
        fileno = self.f.fileno()
        sched.writewait(task,fileno)

class NewTask(SystemCall):
    def __init__(self,target):
        self.target = target
    def handle(self,sched,task):
        sched.new(self.target)
        sched.schedule(task)
```

本例中的代码实现了一个微型“操作系统”。以下是关于其操作的一些细节问题。

- 所有工作都由协程函数完成。必须强调的是，协程使用yield 语句的方式与生成函数类似，但它不采用迭代方法，你必须使用send(*value*) 方法向其发送值。
- Task 类表示运行的任务，它在协程的基础上做了简单封装。Task 对象 *task* 只有一个操作——task.run() 。该操作恢复任务并持续运行，直到抵达下一个yield 语句为止，此时任务将挂起。运行任务时，task.sendval 属性包含将发送到任务对应的yield 表达式的值。在遇到下一条yield 语句之前，任务将一直运行。yield 生成的值控制接下来的任务。
  - 如果返回值是另一个协程（type.GeneratorType ），表示任务需要临时将控制权转移到该协程。Task 对象的stack 属性表示这时已经构建了协程的调用栈。下一次运行任务时，控制权将转移到这个新的协程。
  - 如果返回值是SystemCall 实例，表示任务需要调度程序代表它执行某些操作（如启动新任务、等待I/O等）。该对象的目的已经简单介绍过。
  - 如果返回值是任何其他的值，则可能有两种情况。如果当前执行的协程作为子协程运行，那么它将从任务调用栈中弹出并保存值，以便发送给调用方。调用方将在下一次执行任务时收到该值。如果该协程是唯一执行的协程，则丢弃返回值。
  - 对StopIteration 处理的是处理已经终止的协程。发生这种情况时，控制权将返回到上一个协程（如果有），或者将异常传播到调度程序，通知它任务已经结束。
- SystemCall 类表示调度程序中的系统调用。运行的任务需要调度程序代表它执行操作时，它将生成一个SystemCall 实例。该对象被称为“系统调用”是因为它可以模拟程序请求多任务操作系统（如UNIX或Windows）服务。具体来说，如果某个程序需要操作系统服务，它将获得控制权并向系统提供一些指示信息。从这个角度上看，生

成SystemCall 类似于执行某种系统“陷阱”。

- Scheduler 类表示被管理的Task 对象集合。调度程序的核心部分是围绕任务队列（task_queue 属性）构建的，任务队列负责跟踪准备运行的任务。与任务队列有关的基本操作有4个：new() 新建一个协程，使用Task 对象包装，并将其放在工作队列；schedule() 将现有Task 放回工作队列；mainloop() 以循环方式运行调度程序，逐一处理完所有任务；readwait() 和writewait() 方法将Task 对象放入临时分阶段区域，它将在该区域中等待I/O事件。在这种情况下，Task 不会运行，但是也不会终止——只是等待属于它的时间来临。
- mainloop() 方法是调度程序的核心。该方法首先查看是否有任务在等待I/O事件。如果有，它将安排调用select() 来轮询I/O事件。如果有需要的事件，则相关任务将被放到任务队列中运行。接下来，mainloop() 方法将任务移出任务队列并调用其run() 方法。如果任务退出（StopIteration ），则将其丢弃。如果任务被放弃，那么它将重新返回任务队列以便再次运行。这个过程将一直持续到没有任何任务，或者阻塞了所有任务以等待更多I/O事件时为止。另外还有一个可以选择的操作，即mainloop() 函数的count 参数可用于指定在给定次数的I/O轮询操作之后返回。如果要将调度程序集成到另一个事件循环，那么这个选项会非常有用。
- 调度程序最微妙的方面也许是处理mainloop() 方法的SystemCall 实例了。如果任务生成SystemCall 实例，那么调度程序将调用它的handle() 方法，传递相关的Scheduler 和Task 对象作为参数。系统调用的目的是执行某些与任务或者调度程序本身有关的内部操作。ReadWait() 、WriteWait() 和NewTask() 类都是可以挂起任务进行I/O操作或者新建任务的系统调用。例如，ReadWait() 获取一个任务并对调度程序调用readwait() 方法。然后，调度程序将该任务放入相应的保存区域。这里又有一个关键的对象解耦操作。任务生成SystemCall 对象请求服务，但不会与调度程序直接交互。SystemCall 对象可以对任务和调度程序执行操作，但是不会尝试任何特定的调度程度或任务实现。因此，理论上你可以写入完全不同的调度程序实现（可以使用线程）——只需要将其插入整个框架即可运行。

以下是一个使用这种I/O任务调度程序实现的网络时间服务器示例。该示例将演示上表中介绍的很多概念：

```
from socket import socket, AF_INET, SOCK_STREAM
def time_server(address):
    import time
    s = socket(AF_INET,SOCK_STREAM)
    s.bind(address)
    s.listen(5)
    while True:
        yield ReadWait(s)
        conn,addr = s.accept()
        print("Connection from %s" % str(addr))
        yield WriteWait(conn)
        resp = time.ctime() + "\r\n"
        conn.send(resp.encode('latin-1'))
        conn.close()

sched = Scheduler()
sched.new(time_server(('',10000)))   # 端口10000上的服务器
sched.new(time_server(('',11000)))   # 端口11000上的服务器
sched.run()
```

在该例中，有两台不同的服务器并发运行——每台服务器都侦听不同的端口号（使用telnet连接和测试）。`yield ReadWait()` 和`yield WriteWait()` 语句导致每台服务器上运行的协程挂起，直到相关套接字上可以进行I/O操作为止。这些语句返回时，代码立即进行I/O操作，如`accept()` 或`send()` 。

使用`ReadWait` 和`WriteWait` 看起来特别偏底层，但幸运的是，我们在设计时将这些操作隐藏在库函数和方法背后——只要它们是协程。下例中的对象包装了套接字，并模拟其接口：

```
class CoSocket(object):
    def __init__(self,sock):
        self.sock = sock
    def close(self):
        yield self.sock.close()
    def bind(self,addr):
        yield self.sock.bind(addr)
    def listen(self,backlog):
        yield self.sock.listen(backlog)
    def connect(self,addr):
        yield WriteWait(self.sock)
        yield self.sock.connect(addr)
    def accept(self):
        yield ReadWait(self.sock)
        conn, addr = self.sock.accept()
        yield CoSocket(conn), addr
    def send(self,bytes):
        while bytes:
            evt = yield WriteWait(self.sock)
            nsent = self.sock.send(bytes)
            bytes = bytes[nsent:]
    def recv(self,maxsize):
        yield ReadWait(self.sock)
        yield self.sock.recv(maxsize)
```

下面是使用`CoSocket` 类实现的时间服务器：

```
from socket import socket, AF_INET, SOCK_STREAM
def time_server(address):
        import time
        s = CoSocket(socket(AF_INET,SOCK_STREAM))
        yield s.bind(address)
        yield s.listen(5)
        while True:
            conn,addr = yield s.accept()
            print(conn)
            print("Connection from %s" % str(addr))
            resp = time.ctime()+"\r\n"
            yield conn.send(resp.encode('latin-1'))
            yield conn.close()

sched = Scheduler()
sched.new(time_server(('',10000)))   # 端口10000上的服务器
sched.new(time_server(('',11000)))   # 端口11000上的服务器
sched.run()
```

在本例中，CoSocket 对象的编程接口与普通的套接字类似。唯一的区别在于每个操作都必须以yield 开始（因为所有方法都被定义为协程）。首先，这看起来很疯狂，你可能会问这些乱七八糟的东西能给你带来什么好处？如果你运行上面的服务器，将会发现它不需要使用线程或子进程就可以并发运行。不仅如此，只要忽略所有的yield 关键字，你会发现它有着“正常”的控制流。

下例是一个并发处理多个客户端连接的异步Web服务器，但是它没有使用回调函数、线程或进程。可以比较asynchat 和asyncore 模块中的示例进行学习。

```
import os
import mimetypes
try:
    from http.client import responses    # Python 3
except ImportError:
    from httplib import responses        # Python 2
from socket import *

def http_server(address):
    s = CoSocket(socket(AF_INET,SOCK_STREAM))
    yield s.bind(address)
    yield s.listen(50)

    while True:
        conn,addr = yield s.accept()
        yield NewTask(http_request(conn,addr))
        del conn, addr

def http_request(conn,addr):
    request = b""
    while True:
        data = yield conn.recv(8192)
        request += data
        if b'\r\n\r\n' in request: break

    header_data     = request[:request.find(b'\r\n\r\n')]
    header_text     = header_data.decode('latin-1')
    header_lines    = header_text.splitlines()
    method, url, proto = header_lines[0].split()
    if method == 'GET':
        if os.path.exists(url[1:]):
            yield serve_file(conn,url[1:])
        else:
            yield error_response(conn,404,"File %s not found" % url)
    else:
        yield error_response(conn,501,"%s method not implemented" % method)
    yield conn.close()

def serve_file(conn,filename):
    content,encoding = mimetypes.guess_type(filename)
    yield conn.send(b"HTTP/1.0 200 OK\r\n")
    yield conn.send(("Content-type: %s\r\n" % content).encode('latin-1'))
    yield conn.send(("Content-length: %d\r\n" %
                     os.path.getsize(filename)).encode('latin-1'))
    yield conn.send(b"\r\n")
    f = open(filename,"rb")
    while True:
        data = f.read(8192)
```

```
        if not data: break
        yield conn.send(data)

def error_response(conn,code,message):
    yield conn.send(("HTTP/1.0 %d %s\r\n" %
                     (code, responses[code])).encode('latin-1'))
    yield conn.send(b"Content-type: text/plain\r\n")
    yield conn.send(b"\r\n")
    yield conn.send(message.encode('latin-1'))
sched = Scheduler()
sched.new(http_server(('',8080)))
sched.mainloop()
```

深入学习该例，你将了解很多高级第三方模块使用的协程和并发编程技术。但是要小心，如果过度使用这些技术，下一次代码评审之后你可能会被解雇。

21.4.3 异步联网的时机

正如上一个示例所示，使用异步I/O（`asyncore` 和`asynchat` ）、轮询和协程是Python开发中最微妙的地方。但是，这些技术的使用之频繁超乎你的想象。使用异步I/O常见的一个原因是最小化大量线程可能带来的编程开销，尤其是在管理大量客户端以及存在全局解释器锁有关的限制时（请参考第20章）。

历史上，`asyncore` 模块是支持异步I/O的第一批库模块之一。后来出现了`asynchat` 模块，旨在简化编程。但是，这些模块都将I/O作为事件处理。例如，发生I/O事件时，将触发回调函数。然后回调对I/O事件做出反应并进行某些处理。如果用这种处理方式构建大型应用程序，你将会发现事件处理几乎会对应用程序的所有部分产生影响（例如，I/O事件触发回调，这又将触发更多回调，然后又触发其他回调，无休无止）。最常用的联网包之一Twisted（http://twistedmatrix.com ）采用的就是这种方法。

协程是更现代的方法，但是知名度和使用范围都不高，因为它们是在Python 2.5版本中首次引入的。协程有一个重要的特征，即你可以编写出整个控制流更像线程程序的程序。例如，示例中的Web服务器不使用任何回调函数，与使用线程时要编写的内容极其相似——只需要习惯`yield` 语句的使用。Stackless Python（http://www.stackless.com ）进一步发展了这一思路。

通常来说，你应该控制对大部分网络应用程序使用异步I/O技术的欲望。例如，如果你需要编写通过成百上千网络连接即时传输数据的服务器程序，线程的性能是无可比拟的。这是因为`select()` 的性能随着它必须监视的连接数增加而递减。在Linux上，可以使用特殊函数（如`epoll()` ）缓解这种情况，但这又限制了代码的可移植性。异步I/O的主要好处体现在网络需要集成到其他事件循环（如GUI）的应用程序中，或者在需要执行大量CPU处理的代码添加网络的应用程序中。在这些情况下，使用异步联网可以加快响应时间。

为了演示说明，我们编写了以下程序，它执行歌曲“10 million bottles of beer on the wall（墙上的1 000万瓶美酒）”中描述的任务[②]：

```
bottles = 10000000
```

```
def drink_beer():
    remaining = 12.0
    while remaining > 0.0:
          remaining -= 0.1

def drink_bottles():
    global bottles
    while bottles > 0:
          drink_beer()
          bottles -= 1
```

现在，假设你需要在代码中添加远程监视功能，使应用程序能够连接并了解还剩下多少瓶酒。一种方法是在客户端自己的线程中启动服务器，然后让它随主应用程序一起运行，如下所示：

```
def server(port):
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.bind(('',port))
    s.listen(5)
    while True:
        client,addr = s.accept()
        client.send(("%d bottles\r\n" % bottles).encode('latin-1')
        client.close()
# 启动监视服务器
thr = threading.Thread(target=server,args=(10000,))
thr.daemon=True
thr.start()
drink_bottles()
```

另一种方法是根据I/O轮询编写服务器程序，直接在主计算循环中嵌入轮询操作。下例使用上文介绍的协程调度程序：

```
def drink_bottles():
    global bottles
    while bottles > 0:
          drink_beer()
          bottles -= 1
          scheduler.mainloop(count=1,timeout=0)  # 轮询连接

# 基于协程的异步服务器。
def server(port):
    s = CoSocket(socket.socket(socket.AF_INET,socket.SOCK_STREAM))
    yield s.bind(('',port))
    yield s.listen(5)
    while True:
        client,addr = yield s.accept()
        yield client.send(("%d bottles\r\n" % bottles).encode('latin-1')
        yield client.close()

scheduler = Scheduler()
scheduler.new(server(10000))
drink_bottles()
```

如果编写一个独立的程序，让它周期性连接酒瓶程序并测量接收状态消息所需的响应时间，结果会让你大吃一惊。在我自己的机器上（双核2 GHZ MacBook），基于协程的服务器的平均响应时间（测量了1 000多次请求）为1ms，基于线程的则为5 ms。之所以会产生这种差别，是因为基于协程的代码能够在检测到连接时立即响应，而线程服务器必须通过操作系统的调度才能运行。如果存在CPU绑定线程和Python全局解释器锁，服务器将延迟到CPU绑定线程超过分配时间片段后运行。在很多系统中，这个时间片段将近10 ms，因此上述线程响应时间测量实际上就是等待操作系统抢占CPU绑定任务的平均时间。

轮询的缺点在于：如果轮询发生得过于频繁，它的开销将会很大。例如，尽管本例中的响应时间很短，但使用轮询的应用程序的整个操作完成时间可能会延长50%。如果你将代码修改为仅在每6个啤酒包之后轮询，那么响应时间将缩短到1.2 ms，而程序的运行时间只比没有轮询的情况增加了3%。不幸的是，除了测量应用程序之外，往往没有其他简便方法确定最佳轮询频率。

虽然响应时间缩短似乎是一个好处，但是在自己实现并发性时仍然有很多相关问题。例如，在执行各种锁定操作的任务要特别小心。在Web服务器示例中，有一个可以打开文件并读取数据的代码片段。该操作发生时，整个程序将被冻结——如果文件访问涉及磁盘搜索的话，这个时间可能很长。修复该问题只有一个办法，即额外实现异步文件访问，并将其添加为调度程序的功能。对于执行数据库查询等更加复杂的操作，指出如何以异步方式完成工作将变得更加复杂。解决的方法是在单独的线程中完成工作，并在结果可用时反馈到任务调度程序——谨慎使用消息队列可以实现类似操作。某些系统有一些用于异步I/O的底层系统调用（如UNIX上的`aio_*` 函数系列）。在编写本书时，Python库还无法访问这些函数，但是你可以通过第三方模块找到绑定包。以作者的经验，使用这种功能比想象的要棘手得多，不值得将它引入程序，因为它只会让程序更加复杂，最好使用线程库来处理这种问题。

21.5 `socket`

`socket` 模块可以访问标准的BSD套接字接口。虽然该模块是基于UNIX的，但是它可以在所有平台上使用。套接字接口被设计成通用型的，支持各种网络协议（Internet、TIPC、蓝牙等）。但是，最常用的协议是Internet Protocol（IP），包括TCP和UDP。Python支持IPv4和IPv6，不过IPv4更常用。

应该注意的是，该模块比较偏底层，可以直接访问操作系统提供的网络函数。如果要编写网络应用程序，使用第22章介绍的模块或者本章最后一节介绍的`SocketServer` 将更加简单。

21.5.1 地址族

有些`socket` 函数需要指定地址族。 该族指定要使用的网络协议。以下常量就是为这个目的定义的。

常　　量	描　　述	常　　量	描　　述
AF_BLUETOOTH	蓝牙协议	AF_PACKET	链接级别数据包

AF_INET	IPv4协议（TCP、UDP）	AF_TIPC	透明进程间通信协议
AF_INET6	IPv6协议（TCP、UDP）	AF_UNIX	UNIX域协议
AF_NETLINK	Netlink进程间通信		

在这些协议中，AF_INET 和AF_INET6 是最常用的，因为它们表示标准的网络连接。AF_BLUETOOTH 只能用于支持蓝牙的系统（通常是嵌入式系统）。只有Linux系统支持AF_NETLINK 、AF_PACKET 和AF_TIPC 。AF_NETLINK 用于用户应用程序和Linux内核之间的快速进程间通信。AF_PACKET 用于直接处理数据链接层（如原始的Ethernet数据包）。AF_TIPC 协议用于Linux群集上的高性能IPC（http://tipc.sourceforge.net/ ）。

21.5.2 套接字类型

有些socket 函数还需要指定套接字类型。套接字类型指定要在给定协议族中使用的通信类型（数据流或数据包）。以下常量用于该目的。

常　　量	描　　述	常　　量	描　　述
SOCK_STREAM	面向连接的可靠字节流（TCP）	SOCK_RDM	可靠数据报
SOCK_DGRAM	数据报（UDP）	SOCK_SEQPACKET	序列化连接模式记录传输
SOCK_RAW	原始套接字		

最常用的套接字类型是SOCK_STREAM 和SOCK_DGRAM ，因为它们对应于Internet Protocol套件中的TCP和UDP。SOCK_RDM 是一种可靠的UDP形式，保证交付数据报但不保证顺序（收到数据报的顺序可能与发送的顺序不同）。SOCK_SEQPACKET 使用面向数据流的连接发送数据包，保证顺序和数据包边界。对SOCK_RDM 或SOCK_SEQPACKET 的支持不太广泛，如果你关注可移植性，那么最好不要使用它们。SOCK_RAW 用来提供对原始协议的底层访问，在需要执行某些特殊操作时使用，如发送控制消息（如ICMP消息）。SOCK_RAW 通常仅限于高级用户或管理员运行的程序使用。

并非每个协议族都支持所有这些套接字类型。例如，如果使用AF_PACKET 嗅探Linux上的Ethernet数据包，那么将无法使用SOCK_STREAM 建立面向数据流的连接。你必须使用SOCK_DGRAM 或SOCK_RAW 才行。AF_NETLINK 套接字只支持SOCK_RAW 类型。

21.5.3 寻址

为了在套接字上执行通信，必须指定目标地址。地址的形式取决于套接字的地址族。

1．AF_INET（IPv4）

对于使用IPv4的网络应用程序，地址的形式为元组(*host, port*)。以下是两个示例：

```
('www.python.org', 80)
('66.113.130.182', 25)
```

如果host 为空字符串，则它的含义与INADDR_ANY 相同，即表示任何地址。服务器通常在创建所有客户端都可以连接的套接字时使用。如果host 设置为"<broadcast>"，它的含义与套接字API中的INADDR_BROADCAST 常量相同。

注意，使用"www.python.org" 之类的主机名时，Python使用DNS将主机名解析为一个IP地址。由于DNS配置的不同，每次获得的IP地址可能不同。如果需要避免这种行为，可以使用"66.113.130. 182" 之类的IP地址。

2．AF_INET6（IPv6）

在IPv6中，地址形式为4元组(*host, port, flowinfo, scopeid*)。使用IPv6，*host* 和 *port* 组件的工作方式与IPv4相同，但是IPv6主机地址的数值形式通常是由8个以分号分隔的十六进制数组成的字符串，
如'FEDC:BA98:7654:3210:FEDC:BA98:7654:3210' 或'080A::4:1' （在这种情况下，双冒号用0填充中间的地址组件）。

flowinfo 参数是一个32位数字，包括一个24位流标签（低24位），一个4位属性优先级（接下来的4位）以及4个保留位（高4位）。流标签通常只在发送方需要支持路由器的特殊处理时使用。否则， *flowinfo* 设置为0。

scopeid 参数是一个32位数字，只有在处理本地链接和本地站点的地址时使用。本地链接地址带有前缀'FE80:... '，在同一个LAN的机器之间使用（路由器不用转发链接本地数据包）。在这种情况下， *scopeid* 是用于指定主机上特定网络接口的接口索引。在UNIX上可以使用'ifconfig' 等命令查看该信息，在Windows上使用'ipv6 if' 。本地站点地址总是前缀为'FEC0:... '，它用于同一个站点上的机器之间通信（如特定子网络的所有机器）。在这种情况下， *scopeid* 是站点标识符数字。

如果没有为 *flowinfo* 或 *scopeid* 指定任务数据，那么可以像IPv4一样，以元组(*host, port*)的形式给定IPv6地址。

3．AF_UNIX

对于UNIX域套接字，该地址是包含路径名称的字符串，如'/tmp/myserver' 。

4．AF_PACKET

对于Linux数据包协议，地址是一个元组(*device, protonum* [, *pkttype* [, *hatype* [, *addr*]]])，其中 *device* 是指定设备名称的字符串，如'eth0' 。*protonum* 是一个整数，用于指定<linux/if_ether.h> 报头文件中定义的Ethernet协议

编号（如IP数据包的0x0800 ）。 *packet_type* 是指定数据包类型的整数，它是以下常量之一。

常　　量	描　　述
PACKET_HOST	本地主机的数据包地址
PACKET_BROADCAST	物理层广播数据包
PACKET_MULTICAST	物理层多播
PACKET_OTHERHOST	为另一个主机指定的数据包，但在混杂模式下是由设备驱动程序捕获的数据包
PACKET_OUTGOING	源自机器的数据包，但又会循环回数据包套接字的数据包

hatype 是一个整数，指定ARP协议和<linux/if_arp.h> 报头文件中使用的硬件地址类型。 *addr* 是包含硬件地址的字节字符串，它的结构取决于 *hatype* 的值。对于Ethernet， *addr* 将是保存硬件地址的6字节字符串。

5．AF_NETLINK

对于Linux Netlink协议，地址是一个元组(*pid, groups*)，其中 *pid* 和 *groups* 都是无符号整数。pid 是单播套接字地址，通常与创建套接字进程的进程ID相同，对于内核其值则为0。 *groups* 是一个位掩码，用来指定要加入的多播组。更多信息请参考Netlink文档。

6．AF_BLUETOOTH

蓝牙地址取决于要使用的协议。对于L2CAP，地址是一个元组(*addr, psm*)，其中 *addr* 是一个字符串，如'01:23:45:67:89:ab' 。 *psm* 是一个未指定的整数。对于RFCOMM，地址是一个元组(*addr, channel*)，其中 *addr* 是一个地址字符串，*channel* 是一个整数。对于HCI，地址是一个1元组(deviceno,)，其中 *deviceno* 是一个整数设备号。对于SCO，地址是一个字符串 *host* 。

常量BDADDR_ANY 表示任何地址，是字符串'00:00:00:00:00:00' 。常量BDADDR_LOCAL 是字符串'00:00:00:ff:ff:ff' 。

7．AF_TIPC

对于TIPC套接字，它的地址是一个元组(*addr _type, v1, v2, v3* [*, scope*])，其中所有字段都是无符号整数。 *addr_type* 是以下值之一，这些值也决定了 *v1* 、*v2* 和 *v3* 的值。

地址类型	描　　述
TIPC_ADDR_NAMESEQ	*v1* 是服务器类型， *v2* 是端口标识符， *v3* 是0
TIPC_ADDR_NAME	*v1* 是服务器类型， *v2* 是较低的端口号， *v3* 是较高的端口号
TIPC_ADDR_ID	*v1* 是节点， *v2* 是引用， *v3* 是0

可选的 *scope* 字段有如下可选值TIPC_ZONE_SCOPE 、TIPC_CLUSTER_SCOPE 和 TIPC_NODE_SCOPE 。

21.5.4 函数

socket 模块定义了以下函数。

```
create_connection(address
 [, timeout
])
```

建立与address 的SOCK_STREAM 连接并返回已经连接的套接字对象。 *address* 的形式是元组(*host, port*)， *timeout* 指定一个可选的超时期。该函数首先调用getaddrinfo()，然后尝试连接所有已经返回的元组。

```
fromfd(fd
, family
, socktype
 [, proto
])
```

通过整数文件描述符创建套接字对象 *fd* 。地址族、套接字类型和协议编号与socket() 中的相同。文件描述符必须引用之前创建的套接字。该方法返回 SocketType 实例。

```
getaddrinfo(host
, port
 [,family
 [, socktype
 [, proto
 [, flags
]]]])
```

给定关于主机的 *host* 和 *port* 信息，该函数返回由元组组成的列表，元组中包含打开套接字连接所需的信息。 *host* 是包含主机名或数字IP地址的字符串。 *port* 是表示服务名称的数字或字符串（如"http"、"ftp"、"smtp"）。返回的每个元组都包括5个元素（ *family* 、 *socktype* 、 *proto* 、 *canonname* 、 *sockaddr* ）。 *family* 、 *socktype* 和 *proto* 项目的值与传递给socket()函数的值相同。 *canonname* 是表示主机规范名称的字符串。 *sockaddr* 是一个元组，其中包含上文中介绍的与Internet地址有关的套接字地址。示例如下：

```
>>> getaddrinfo("www.python.org",80)

[(2,2,17,'',('194.109.137.226',80)), (2,1,6,'',('194.109.137.226'),80))]
```

在本例中，getaddrinfo()返回与两个套接字连接有关的信息。第一个（proto=17）是UDP连接，第二个（proto=6）是TCP连接。getaddrinfo()的其他参数可以用来缩小选择范围。例如，该例返回与建立IPv4 TCP连接有关的信息：

```
>>> getaddrinfo("www.python.org",80,AF_INET,SOCK_STREAM)

[(2,1,6,'',('194.109.137.226',80))]
```

地址族可以使用特殊的常量AF_UNSPEC查询所有类型的连接。例如，以下代码提供有关类似TCP连接的信息，可能返回有关IPv4或IPv6的信息：

```
>>> getaddrinfo("www.python.org","http", AF_UNSPEC, SOCK_STREAM)

[(2,1,6,'',('194.109.137.226',80))]
```

getaddrinfo() 的用途很广，适用于所有受支持的网络协议（IPv4、IPv6等）。如果你关心兼容性以及对将来协议的支持，尤其是如果你希望支持IPv6，那么应该使用该函数。

```
getdefaulttimeout()
```

返回默认的套接字超时期（以秒为单位）。Note 值表示不设置任何超时期。

```
getfqdn([name
])
```

返回完全限定域名 *name* 。如果忽略 *name* ，那么假定为本地机器。例如，getfqdn("foo") 返回"foo.quasievil.org" 。

```
gethostbyname(hostname
)
```

将主机名（如"www.python.org" ）转换为IPv4地址。IP地址将以字符串形式返回，如"132.151. 1.90" 。不支持IPv6。

```
gethostbyname_ex(hostname
)
```

将主机名转换为IPv4地址，但返回元组(*hostname, aliaslist, ipaddrlist*)，其中 *hostname* 是主机名， *aliaslist* 是同一个地址的可选主机名列表，*ipaddrlist* 是同一个主机上同一个接口的IPv4地址列表。例如，gethostbyname_ex('www.python.org') 返回('fang.python.org', ['www.

python.org '], ['194.109.137.226']) 。该函数不支持IPv6。

```
gethostname()
```

返回本地机器的主机名。

```
gethostbyaddr(ip_address
)
```

返回的信息与gethostbyname_ex() 相同，但要提供'132.151.1.90' 之类的IP地址。如果 *ip_address* 是IPv6地址，
如'FEDC:BA98:7654:3210:FEDC:BA98:7654:3210' ，那么将返回关于IPv6的信息。

```
getnameinfo(address
, flags
)
```

给定套接字地址address ，该函数将地址转换为由 *flags* 的值决定的2元组(*host, port*)。 *address* 参数是指定地址的元组，如('www.python.org',80) 。 *flags* 是以下常量的按位或：

常　　量	描　　述
NI_NOFQDN	不为本地主机使用完全限定名
NI_NUMERICHOST	以数字形式返回地址
NI_NAMEREQD	要求提供主机名。如果 *address* 没有任何DNS项，则返回一个错误
NI_NUMERICSERV	返回的端口将以包含端口号的字符串形式返回

NI_DGRAM	指定要查询的服务是数据报服务（UDP）而不是TCP（默认值）

该函数的主要目的是获取与地址有关的详细信息。示例如下：

```
>>> getnameinfo(('194.109.137.226',80),0)


('fang.python.org', 'http')
>>> getnameinfo(('194.109.137.226',80),NI_NUMERICSERV)


('fang.python.org','80')

getprotobyname

(
protocolname



)
```

将Internet协议名称（如'icmp'）转换为协议编号（如IPPROTO_ICMP 的值），以便传递到socket() 函数的第3个参数。如果无法识别协议名称，则会出现socket.error 错误。通常，该函数仅供原始套接字使用。

```
getservbyname(servicename

 [, protocolname

])
```

将Internet服务名称和协议名称转换为该服务的端口号。例如，getservbyname('ftp', 'tcp') 返回21。提供的协议名称应该是'tcp' 或'udp' 。如果 *servicename* 不匹配任何已知服务，则出现socket.error 错误。

```
getservbyport(port

 [, protocolname

])
```

该函数与getservbyname() 相反。如果给定数字端口号 *port* ，该函数将返回表示服务名称（如果有）的字符串。例如，getservbyport(21, 'tcp') 返回'ftp' 。提供的协议名称应该是'tcp' 或'udp' 。如果没有任何服务名称可用于 *port* ，则出现 socket.error 错误。

has_ipv6

布尔值常量，支持IPv6时为True 。

htonl(x)

将主机的32位整数转换为网络字节顺序（大尾）。

htons(x)

将主机的16位整数转换为网络字节顺序（小尾）。

inet_aton(*ip_string*

)

将以字符串形式提供的IPv4地址（如'135.128.11.209' ）转换为32位二进制分组格式，用作地址的原始编码。返回值是由包含二进制编码的4个字符组成的字符串。在将地址传递给C或者地址必须打包到传递给其他程序的数据结构中时，该函数非常有用。它不支持IPv6。

inet_ntoa(*packedip*

)

将二进制的IPv4地址转换为使用标准句点表示的字符串（如'135.128.11.209'）。 *packedip* 是由包含IP地址原始32位编码的4个字符组成的字符串。在从C接收地址或者从数据结构解包地址时，该函数非常有用。它不支持IPv6。

```
inet_ntop(address_family
, packed_ip
)
```

将表示IP网络地址的二进制分组字符串 *packed_ip* 转换为类似'123.45.67.89'的字符串。 *address_family* 是地址族，通常是AF_INET或AF_INET6。该函数可以用来从原始字节缓冲区获取网络地址字符串（例如，从底层网络数据包的内容中获取）。

```
inet_pton(address_family
, ip_string
)
```

将类似'123.45.67.89'的IP地址转换为分组字节字符串。 *address_family* 是地址族，通常是AF_INET或AF_INET6。在尝试将网络地址编码为原始二进制数据包时，可以使用该函数。

```
ntohl(x
)
```

将来自网络的32位整数（大尾）转换为主机字节顺序。

```
ntohs(x
)
```

将来自网络的16位整数（小尾）转换为主机字节顺序。

```
setdefaulttimeout(timeout
)
```

为新建的套接字对象设置默认超时期。 *timeout* 是以秒为单位指定的浮点数。可以提供None 值指示没有超时期（这是默认值）。

```
socket(family
, type
 [, proto
])
```

使用给定的地址族、套接字类型和协议编号新建套接字。 *family* 是一个地址族，*type* 是21.5.2节中讨论的套接字类型。要打开TCP连接，使用socket(AF_INET, SOCK_STREAM) 。要打开UDP连接，使用socket(AF_INET, SOCK_DGRAM) 。该函数返回SocketType 实例（前面介绍过）。

通常省略协议编号（默认值为0）。该函数通常仅与原始套接字（SOCK_RAW ）一起使用，并根据要使用的地址族设置为对应常量。下表列出了Python根据它们在主机系统上的可用性为AF_INET 和AF_INET6 定义的所有协议编号。

常　　量	描　　述
IPPROTO_AH	IPv6验证报头
IPPROTO_BIP	Banyan VINES（虚拟综合网络服务）
IPPROTO_DSTOPTS	IPv6目标选项
IPPROTO_EGP	外部网关协议

IPPROTO_EON	ISO CNLP（Connectionless Network Protocol，无连接网络协议）
IPPROTO_ESP	封装安全负载的IPv6
IPPROTO_FRAGMENT	IPv6报头片段
IPPROTO_GGP	网关对网关协议（RFC823）
IPPROTO_GRE	通用路由封装（RFC1701）
IPPROTO_HELLO	Fuzzball HELLO协议
IPPROTO_HOPOPTS	IPv6逐跳选项（hop-by-hop options）
IPPROTO_ICMP	IPv4 ICMP
IPPROTO_ICMPV6	IPv6 ICMP
IPPROTO_IDP	XNS IDP
IPPROTO_IGMP	组管理协议
IPPROTO_IP	IPv4
IPPROTO_IPCOMP	IP负载压缩协议
IPPROTO_IPIP	IP内部IP
IPPROTO_IPV4	IPv4报头
IPPROTO_IPV6	IPv6报头
IPPROTO_MOBILE	IP移动性
IPPROTO_ND	Netdisk协议
IPPROTO_NONE	IPv6没有下一个报头

IPPROTO_PIM	独立于协议的多播
IPPROTO_PUP	Xerox PARC Universal Packet（PUP，帕洛阿尔研究中心通用包）
IPPROTO_RAW	原始IP数据包
IPPROTO_ROUTING	IPv6路由报头
IPPROTO_RSVP	资源保留
IPPROTO_TCP	TCP
IPPROTO_TP	OSI 传输协议（TP-4）
IPPROTO_UDP	UDP
IPPROTO_VRRP	虚拟路由冗余协议
IPPROTO_XTP	快速传送协议

以下协议编号将与AF_BLUETOOTH 一起使用。

常　　量	描　　述
BTPROTO_L2CAP	逻辑链接控制和适配协议
BTPROTO_HCI	主机/控制程序接口
BTPROTO_RFCOMM	电缆复位协议（Cable replacement protocol）
BTPROTO_SCO	面向异步连接的链接

```
socketpair([family
 [, type
 [, proto
 ]]])
```

使用给定的 *family* 、type 和 *proto* 选项创建一对连接的套接字对象，这些参数的含义与socket() 函数的参数相同。该函数仅适用于UNIX域套接字（*family=AF _UNI*） 。 *type* 可以是SOCK_DGRAM 或SOCK_STREAM 。如果 *type* 是SOCK_STREAM ，将创建一个名为数据流管道 的对象。 *proto* 通常是0（默认值）。该函数主要用于设置os.fork() 创建的进程之间的通信。例如，父进程调用socketpair() 创建一对套接字并调用os.fork() 。然后，父进程和子进程就可以使用这些套接字相互通信了。

套接字使用类型SocketType 的实例表示。套接字 *s* 具有以下方法。

```
s
.accept()
```

接受连接并返回(conn, address) ，其中conn 是新的套接字对象，可以用来通过连接发送和接收数据，address 是另一个连接端的套接字地址。

```
s
.bind(address)
```

将套接字绑定到地址。 *address* 的格式取决于地址族。在大部分情况下，它是元组形式(*hostname, port*) 。对于IP地址，空字符串表示INADDR_ANY ，字符串'<broadcast>' 表示INADDR_BROADCAST 。INADDR_ANY 主机名（空字符串）用来表示服务器可以连接系统上的任何网络接口。这常常在多宿主服务器上使用。INADDR_BROADCAST 主机名（'<broadcast>' ）在使用套接字发送广播消息时使用。

```
s
.close()
```

关闭套接字。套接字被垃圾回收时也会关闭。

```
s
.connect(address
)
```

连接到 *address* 处的远程套接字。 *address* 的格式取决于地址族，但是它通常是一个元组(*hostname, port*)。如果有错误则引发socket.error 。如果要连接同一台计算机上的服务器，则可以使用'localhost' 作为 *hostname* 。

```
s
.connect_ex(address
)
```

与connect(*address*) 类似，但是成功时返回0，失败时返回errno 的值。

```
s
.fileno()
```

返回套接字的文件描述符。

```
s
.getpeername()
```

返回连接套接字的远程地址。返回值通常是一个元组(*ipaddr, port*)，具体取决于使用的地址族。并非所有系统都支持该函数。

```
s
.getsockname()
```

返回套接字自己的地址。通常是一个元组(*ipaddr, port*)。

```
s
.getsockopt(level
, optname
 [, buflen
])
```

返回套接字选项的值。 *level* 定义选项的级别，套接字级别的选项用SOL_SOCKET表示，与协议相关的选项用IPPROTO_IP 之类的协议编号表示。 *optname* 选择特定的选项。如果忽略 *buflen* ，则假定使用整数选项并返回其整数值。如果提供 *buflen* ，则表示用来接收选项的最大长度。缓冲区作为字节字符串返回，由调用方决定使用 *struct* 模块还是其他方法来解码其内容。

下面列出了Python定义的套接字选项。这些选项的绝大部分被视为高级套接字接口（Advanced Sockets API）的一部分，用于控制网络的底层细节。需要参考其他文档了解更详细的介绍。如果在“值”列中列出了类型名称，则表示该名称和与该值相关的标准C数据结构的名称相同，标准套接字编程接口中也使用了这些数据结构。并非所有机器都可以使用这些选项。

以下是级别SOL_SOCKET 常用的选项名称。

选项名称	值	描　　述
SO_ACCEPTCONN	0,1	确定套接字是否接受连接
SO_BROADCAST	0,1	允许发送广播数据报

SO_DEBUG	0,1	确定是否记录调试信息
SO_DONTROUTE	0,1	绕过路由表查询
SO_ERROR	int	获取错误状态
SO_EXCLUSIVEADDRUSE	0,1	防止其他套接字被强行绑定到同一个地址和端口。这将禁用SO_REUSEADDR选项
SO_KEEPALIVE	0,1	周期性探测连接的另一端，如果连接是半开的则将其终止
SO_LINGER	linger	如果发送缓冲区包含数据则延迟close() 。linger 是包含32位整数的分组二进制字符串(*onoff, seconds*)
SO_OOBINLINE	0,1	将带外数据放入输入队列
SO_RCVBUF	int	接收缓冲区的大小（以字节为单位）
SO_RCVLOWAT	int	在select() 返回可读套接字之前读取的字节数
SO_RCVTIMEO	timeval	接收调用的超时期（以秒为单位）。timeval 是包含两个32位未指定整数的分组二进制字符串(*seconds, microseconds*)
SO_REUSEADDR	0,1	支持重用本地地址
SO_REUSEPORT	0,1	只要在所有进程中设置了该套接字选项，即可支持将多个进程绑定到一个地址
SO_SNDBUF	int	发送缓冲区的大小（以字节为单位）
SO_SNDLOWAT	int	在select() 返回可写套接字之前发送的可用字节数
SO_SNDTIMEO	timeval	发送调用的超时期（以秒为单位）。有关timeval 的描述，请参见SO_RCVTIMEO
SO_TYPE	int	获取套接字类型
SO_USELOOPBACK	0,1	路由套接字获取它发送的副本

级别IPPROTO_IP 具有以下选项。

选项名称	值	描　　述
IP_ADD_MEMBERSHIP	ip_mreg	加入多播组（仅支持集合）。ip_mreg 是包含两个32位IP地址的分组二进制字符串(*multiaddr, localaddr*)，其中 *multiaddr* 是多播地址，localaddr 是要使用的本地接口IP
IP_DROP_MEMBERSHIP	ip_mreg	离开多播组（仅支持集合）。 *ip_mreg* 含义同上
IP_HDRINCL	int	与数据包含在一起的IP报头
IP_MAX_MEMBERSHIPS	int	多播组的最大数字
IP_MULTICAST_IF	in_addr	传出接口。in_addr 是包含32位IP地址的分组二进制字符串
IP_MULTICAST_LOOP	0,1	回路
IP_MULTICAST_TTL	uint8	存活时间。uint8 是包含一个1字节无符号char 的分组二进制字符串
IP_OPTIONS	ipopts	IP报头选项。 *ipopts* 是不超过44个字节的分组二进制字符串。有关该字符串的内容，请参见RFC 791中的描述
IP_RECVDSTADDR	0,1	与数据报一起接收IP目标地址
IP_RECVOPTS	0,1	与数据报一起接收所有IP选项
IP_RECVRETOPTS	0,1	与响应一起接收IP选项
IP_RETOPTS	0,1	与IP_RECVOPTS 类似，但选项都未进行处理，不填充时间戳或路由记录选项
IP_TOS	int	服务类型
IP_TTL	int	生存时间

级别IPPROTO_IPV6 具有以下选项。

选项名称	值	描　　述
IPV6_CHECKSUM	0,1	让系统计算校验和

IPV6_DONTFRAG	0,1	数据包超出MTU大小时不要将其分段
IPV6_DSTOPTS	*ip6_dest*	目标选项。 *ip6* _dest 是形式为(*next* , *len* , *options*) 的分组二进制字符串，其中 *next* 是一个8位整数，提供下一个报头的选项类型； *len* 是一个8位整数，指定报头的长度（单位为8字节），但不包括最前面的8个字节； *options* 是编码选项
IPV6_HOPLIMIT	int	跳数限制
IPV6_HOPOPTS	*ip6_hbh*	逐跳寻址选项。 *ip6_hbh* 的编码与 *ip6_dest* 相同
IPV6_JOIN_GROUP	*ip6_mreq*	加入多播组。 *ip6_mreg* 是包含(*multiaddr* , *index*) 的分组二进制字符串，其中 *multiaddr* 是128位IPv6多播地址，index 是本地接口的32位无符号整数接口索引
IPV6_LEAVE_GROUP	*ip6_mreq*	离开多播组
IPV6_MULTICAST_HOPS	int	多播数据包的跳数限制
IPV6_MULTICAST_IF	int	传出多播数据包的接口索引
IPV6_MULTICAST_LOOP	0,1	将传出多播数据包传回本地应用程序
IPV6_NEXTHOP	*sockaddr_in6*	设置传出数据包的下一个跳动地址。 *sockaddr_in6* 是包含C *sockaddr_in6* 结构的分组二进制字符串，通常在<netinet/in.h> 中定义。
IPV6_PKTINFO	*ip6_pktinfo*	数据包信息结构。 *ip6_pktinfo* 是包含(*addr, index*) 的分组二进制字符串，其中 *addr* 是一个128位IPv6地址， *index* 是一个带有接口索引的32位无符号整数
IPV6_RECVDSTOPTS	0,1	接收目标选项
IPV6_RECVHOPLIMIT	0,1	接收跳数限制
IPV6_RECVHOPOPTS	0,1	接收逐跳寻址选项
IPV6_RECVPKTINFO	0,1	接收数据包信息
IPV6_RECVRTHDR	0,1	接收路由报头
IPV6_RECVTCLASS	0,1	接收通信类

IPV6_RTHDR	*ip6_rthdr*	路由报头。 *ip6_rthdr* 是包含(*next, len, type, segleft , data*) 的分组二进制字符串，其中 *next* 、len 、type 和 *segleft* 都是8位无符号整数， *data* 是路由数据。请参见RFC 2460
IPV6_RTHDRDSTOPTS	*ip6_dest*	路由选项报头之前的目标选项报头
IPV6_RECVPATHMTU	0,1	支持接收IPV6_PATHMTU辅助数据项
IPV6_TCLASS	int	通信类
IPV6_UNICAST_HOPS	int	多播数据包的跳数限制
IPV6_USE_MIN_MTU	-1,0,1	路径MTU显示。1禁用所有目标。-1只禁用多播目标
IPV6_V6ONLY	0,1	只连接其他IPV6节点

级别SOL_TCP具有以下选项。

选项名称	值	描　　述
TCP_CORK	0,1	不发出部分数据帧（如果已设置）
TCP_DEFER_ACCEPT	0,1	仅在数据到达套接字时唤醒侦听程序
TCP_INFO	*tcp_info*	返回包含套接字信息的结构。 *tcp_info* 与实现有关
TCP_KEEPCNT	int	放弃连接之前TCP应该发送的最大keepalive探测消息数量
TCP_KEEPIDLE	int	如果设置了TCP_KEEPALIVE 选项，指定TCP开始发送keepalive探测消息之前，连接应该处于空闲状态的时间（以秒为单位）
TCP_KEEPINTVL	int	进行keepalive探测间隔的秒数
TCP_LINGER2	int	不常用的FIN_WAIT2 状态套接字的生命周期
TCP_MAXSEG	int	传出TCP数据包的最大片段大小
TCP_NODELAY	0,1	如果设置，则禁用Nagle算法

TCP_QUICKACK	0,1	如果设置，则立即发送ACK。禁用TCP延迟ACK算法
TCP_SYNCNT	int	退出连接请求之前的SYN中继数量
TCP_WINDOW_CLAMP	int	在公告的TCP窗口大小上设置上限

```
s
.gettimeout()
```

返回当前超时期的值（如果有）。返回浮点值数字（秒），如果没有设置超时期，则返回None 。

```
s
.ioctl(control
, option
)
```

受限访问Windows上的WSAIoctl接口。唯一支持的 *control* 值是SIO_RCVALL ，它可以捕获网络上收到的所有IP数据包。使用该函数需要登录管理员账户。以下值可用于 *options* 。

选　　项	描　　述
RCVALL_OFF	禁止套接字接收所有IPv4或IPv6数据包
RCVALL_ON	启用混杂模式，允许套接字接收网络上的所有IPv4或IPv6数据包。接收的数据包类型取决于套接字地址族。该选项不捕获与其他网络协议（如ARP）有关的数据包
RCVALL_IPLEVEL	接收网络上收到的所有IP数据包，但是不启用混杂模式。该选项捕获已配置IP地址的主机发送的所有IP数据包

```
s
.listen(backlog
)
```

开始监听传入连接。 *backlog* 指定在拒绝连接之前，操作系统应该列队的最大挂起连接数量。该值至少为1，大部分应用程序设置为5就足够了。

```
s
.makefile([mode
 [, bufsize
]])
```

创建与套接字关联的文件对象。 *mode* 和 *bufsize* 的含义与内置的open() 函数相同。文件对象使用套接字文件描述符的复制版本，它是使用os.dup() 创建的，因此文件对象和套接字对象可以独立关闭或进行垃圾回收。套接字 *s* 应该有一个超时期，而且不能配置为非阻塞模式。

```
s
.recv(bufsize
 [, flags
])
```

接收套接字的数据。数据以字符串形式返回。 *bufsize* 指定要接收的最大数据量。*flags* 提供有关消息的其他信息，通常可以忽略（这种情况下默认为0）。如果使用该参数，它通常设置为以下常量之一（与系统无关）。

常　量	描　述

MSG_DONTROUTE	绕过路由表查询（仅在发送数据时使用）
MSG_DONTWAIT	非阻塞操作
MSG_EOR	指示消息是记录中的最后一条。通常仅在SOCK_SEQPACKET 套接字上发送数据时使用
MSG_PEEK	查看数据但是不丢弃（仅在接收时使用）
MSG_OOB	接收/发送带外数据
MSG_WAITALL	在读取请求的字节数之前不要返回（仅在接收时使用）

```
s.recv_into(buffer
 [, nbytes
 [, flags
]])
```

与recv() 类似，但是数据写入到支持缓冲区接口的对象 *buffer* 。 *nbytes* 是要接收的最大字节数。如果忽略该参数，则最大大小等于缓冲区的大小。 *flags* 的含义与recv() 中的相同。

```
s
.recvfrom(bufsize
 [, flags
])
```

与recv() 类似，但是返回值是(*data, address*) 对，其中 *data* 是包含接收数据的字符串， *address* 是发送数据的套接字地址。可选的 *flags* 参数的含义与recv() 中的相同。该函数主要在通过UDP协议连接时使用。

```
s
.recvfrom_info(buffer
 [, nbytes
 [, flags
]])
```

与recvfrom()类似，但是接收的数据存储在缓冲对象 *buffer* 中。 *nbytes* 指定要接收的最大字节数。如果忽略该参数，则最大大小为 *buffer* 的大小。 *flags* 的含义与recv()中的相同。

```
s
.send(string
 [, flags
])
```

将 *string* 中的数据发送到连接的套接字。可选 *flags* 参数的含义与recv()中相同（上文已经介绍）。返回要发送的字节数量，该数量可能小于 *string* 中的字节数。如果有错误则抛出异常。

```
s
.sendall(string
 [, flags
])
```

将 *string* 中的数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功则返回None；失败则抛出异常。 *flags* 的含义与send()中的相同。

```
s
.sendto(string
 [, flags
], address
)
```

将数据发送到套接字。 *flags* 的含义与recv() 中的相同。 *address* 是形式为 (*host, port*) 的元组，指定远程地址。在此之前，套接字不应该被连接。返回发送的字节数。该函数主要在通过UDP协议连接时使用。

```
s
.setblocking(flag
)
```

如果 *flag* 为0，则将套接字设置为非阻塞模式。否则，套接字将设置为阻塞模式（默认值）。在非阻塞模式下，如果recv() 调用没有发现任何数据，或者send() 调用无法立即发送数据，那么将引发socket.error 异常。在阻塞模式下，这些调用在处理之前都将被阻塞。

```
s
.setsockopt(level
, optname
, value
)
```

设置给定套接字选项的值。 *level* 和 *optname* 的含义与getsockopt() 中的含义相同。该值可以是一个整数，也可以是表示缓冲区内容的字符串。在后一种情况下，由调

用方确保字符串包含正确的数据。有关套接字选项的名称、值和说明，请参见getsockopt()。

```
s
.settimeout(timeout
)
```

设置套接字操作的超时期。 *timeout* 是一个浮点数（秒）。值为None 表示没有超时期。如果发生超时，则引发socket.timeout 异常。一般来说，超时期应该在刚创建套接字的时候设置，因为它们可能会应用于建立连接的操作（如connect()）。

```
s
.shutdown(how
)
```

关闭一个或两个连接。如果 *how* 为0，则可以继续接收。如果 *how* 为1，则不允许后续发送。如果 *how* 为2，则不允许后续发送和接收。

除了这些方法之外，套接字实例s还具有以下只读属性，这些属性与传递给socket()函数的参数对应。

属　　性	描　　述
s .family	套接字地址族（如AF_INET ）
s .proto	套接字协议
s .type	套接字类型（如SOCK_STREAM ）

21.5.5 异常

socket 模块定义了以下异常。

error

该异常表示与套接字或地址有关的错误。它返回一个(*errno, mesg*)对以及底层系统调用返回的错误。继承自IOError 。

herror

表示与地址有关的错误。返回一个包含错误编号和错误消息的元组(*herrno, hmesg*)。继承自error 。

gaierror

表示getaddrinfo() 和getnameinfo() 函数中与地址有关的错误。错误值是一个元组(*errno, mesg*)，其中 *errno* 是错误编号，mesg 是包含消息的字符串。 *errno* 设置为socket 模块中定义的以下常量之一。

常　　量	描　　述	常　　量	描　　述
EAI_ADDRFAMILY	不支持地址族	EAI_NODATA	没有与节点名称相关的地址
EAI_AGAIN	名称解析暂时失败	EAI_NONAME	未提供节点名称或服务名称
EAI_BADFLAGS	标志无效	EAI_PROTOCOL	不支持该协议
EAI_BADHINTS	提示不当	EAI_SERVICE	套接字类型不支持该服务名称
EAI_FAIL	不可恢复的名称解析失败	EAI_SOCKTYPE	不支持该套接字类型
EAI_FAMILY	主机不支持的地址族	EAI_SYSTEM	系统错误
EAI_MEMORY	内存分配失败		

timeout

套接字操作超时时出现的异常。此异常只有在使用套接字对象的`setdefaulttimeout()`函数，或`settimeout()`方法设置了超时时才会发生。异常值是字符串`'timeout'`，继承自`error`。

21.5.6 示例

本章前面提供了一个简单的TCP连接示例。下面将提供一个简单的UDP回显服务器示例：

```
# UDP消息服务器
# 从任何地方接收小型数据包并打印
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(("",10000))
while True:
     data, address = s.recvfrom(256)
     print("Received a connection from %s" % str(address))
     s.sendto(b"echo:" + data, address)
```

下面是将消息发送到上例服务器的客户端：

```
# UDP消息客户端
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(b"Hello World", ("", 10000))
resp, addr = s.recvfrom(256)
print(resp)
s.sendto(b"Spam", ("", 10000))
resp, addr = s.recvfrom(256)
print(resp)
s.close()
```

21.5.7 注意

- 并非所有常量和套接字选项都可用于所有平台。如果以可移植性为目标，那么最好使用主要参考资料中介绍的选项，例如，你可以参考本节开头介绍的《UNIX网络编程》（W.Richard Stevens著）。
- 值得注意的是，`socket`模块中忽略了`recvmsg()`和`sendmsg()`系统调用，它们常用于处理辅助数据以及与数据包报头、路由和其他细节有关的高级网络选项。若需要该功能，你必须安装第三方模块，如PyXAPI（http://pypi.python.org/pypi/PyXAPI）。
- 非阻塞套接字操作和涉及超时的操作之间存在细微的差别。在非阻塞模式下使用套接

字函数时，如果操作被阻塞，它将立即返回一个错误。设置超时期时，只有在操作没有在指定超时期内完成时函数才返回错误。

21.6 ssl

ssl 模块使用SSL（Secure Sockets Layer，安全套接字层）包装套接字对象，SSL提供数据加密和对等验证功能。Python使用OpenSSL库（http://www.openssl.org ）实现该模块。本书不打算全面讨论有关SSL的理论和操作，只介绍使用该模块时必须了解的内容，并假定你了解SSL配置、密钥、证书以及其他相关问题。

```
wrap_socket(sock
 [, **opts
])
```

包装现有套接字sock （使用socket 模块创建）与SSL，返回SSLSocket 实例。该函数应该在执行后续connect() 或accept() 操作之前使用。 *opts* 表示用于指定其他配置数据的密钥参数的数量。

密钥参数	描　　述
server_side	布尔标志，指示是否以服务器（True ）或客户端（False ）的形式操作套接字。默认情况下，该参数为False
keyfile	用来标识连接本地端的密钥文件。应该是一个PEM格式的文件，通常只有在使用不包含密钥的certfile 指定文件时才包含该参数
certfile	用来标识连接本地端的证书文件。应该是PEM格式的文件
cert_reqs	指定连接的另一端是否需要证书，以及它是否进行了验证。值为CERT_NONE 表示忽略证书；CERT_OPTIONAL 表示不需要证书，但如果给定，则需要验证；CERT_REQUIRED 表示需要证书，也需要验证。如果要验证证书，则必须提供ca_certs 参数
ca_certs	文件名，该文件包含用于验证的授权证书
ssl _version	使用的SSL协议版本。可能的值包括：PROTOCOL_TLSv1 ，PROTOCOL_SSLv2 ，PROTOCOL_SSLv23 或PROTOCOL_SSLv3 。默认协议是PROTOCOL_SSLv3
do_handshake_on_connect	布尔标志，指定是否在连接时自动执行SSL握手。默认情况下，该参数为True

suppress_ragged_eofs	指定read() 如何处理连接上的异常EOF。如果为True （默认值），则发出普通EOF信号；如果为False ，则发出异常

SSLSocket 的实例 *s* 继承自socket.socket ，还支持以下操作。

```
s
.cipher()
```

返回一个元组(*name, version, secretbits*)，其中 *name* 是要使用的密码名称。 *version* 是SSL协议， *secretbits* 是要使用的密码位数。

```
s
.do_handshake()
```

执行SSL握手。一般情况下是自动进行的，除非使用 *wrap_socket()* 函数将do_handshake_on_connect 选项设置为False 。如果底层套接字 *s* 是非阻塞的，那么在无法完成操作时将出现SSLError 异常。SSLError 异常 *e* 的 *e* .args[0] 属性具有SSL_ERROR_WANT_READ 或SSL_ERROR_WANT_WRITE 值，具体取决于需要执行的操作。如果在读取和写入可以继续操作后要继续握手进程，只要再次调用 *s* .do_handshake() 即可。

```
s
.getpeercert([binary
_form
])
```

返回另一个连接端的证书（如果有）。如果没有证书，则返回None 。如果有证书但未经验证，则返回一个空字典。如果收到了已经验证的证书，则返回带有键'subject'

和'notAfter' 的字典。如果binary_form 置为True ， 则以DER编码字节序列的形式返回证书。

```
s
.read([nbytes
])
```

最多读取 *nbytes* 个数据并返回。如果忽略 *nbytes* ， 则最多返回1 024个字节。

```
s
.write(data
)
```

写入字节字符串 *data* 。返回写入的字节数。

```
s
.unwrap()
```

关闭SSL连接并返回可以执行进一步未加密通信的底层套接字对象。

该模块还定义了以下实用函数。

```
cert_time_to_seconds(timestring
)
```

将字符串 *timestring* 从证书使用的格式转换为兼容time.time()函数的浮点数。

```
DER_cert_to_PEM_cert(derbytes
)
```

给定包含DER编码证书的字节字符串 *derbytes* ，返回PEM编码证书的字符串版本。

```
PEM_cert_to_DER_cert(pemstring
)
```

给定包含PEM编码字符串版本证书的字符串 *pemstring* ，返回证书DER编码的字节字符串版本。

```
get_server_certificate(addr
 [, ssl_version
 [, ca_certs
]])
```

检索SSL服务器证书并以PEM编码字符串的形式返回。addr是形式为(*hostname, port*)的地址。 *ssl_version* 是SSL版本号， *ca_certs* 是包含证书授权的文件名，与 *wrap_socket()* 函数中的相同。

```
RAND_status()
```

返回True或False，表示SSL层认为伪随机数字生成器生成的数字是否足够随机。

```
RAND_egd(path
)
```

从后台函数收集程序（Entropy Gathering Demon）中读取256个随机字节，并添加到伪随机数字生成器。path 是后台程序UNIX域套接字的名称。

```
RAND_add(bytes
, entropy
)
```

将字节字符串 *bytes* 中的字节添加到伪随机数生成器。 *entropy* 是非负浮点数，限定函数的下界。

示例

下例展示了如何使用该模块打开SSL客户端连接：

```
import socket, ssl

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_s = ssl.wrap_socket(s)
ssl_s.connect(('gmail.google.com',443))
print(ssl_s.cipher())
# 发送请求
ssl_s.write(b"GET / HTTP/1.0\r\n\r\n")
# 获得响应
while True:
    data = ssl_s.read()
    if not data: break
    print(data)
ssl_s.close()
```

下面是使用受SSL保护的时间服务器示例：

```
import socket, ssl, time

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)
s.bind(('',12345))
s.listen(5)
```

```
while True:
    client, addr = s.accept()      # 进行连接
    print "Connection from", addr
    client_ssl = ssl.wrap_socket(client,
                                 server_side=True,
                                 certfile="timecert.pem")

    client_ssl.sendall(b"HTTP/1.0 200 OK\r\n")
    client_ssl.sendall(b"Connection: Close\r\n")
    client_ssl.sendall(b"Content-type: text/plain\r\n\r\n")
    resp = time.ctime() + "\r\n"
    client_ssl.sendall(resp.encode('latin-1'))
    client_ssl.close()
    client.close()
```

要运行该服务器，文件timecert.pem 中必须有一个签名的服务器证书。如果要进行测试，可以使用以下UNIX命令创建一个：

```
% openssl req -new -x509 -days 30 -nodes -out timecert.pem -keyout timecert.pem
```

要测试该服务器，可以使用"https://localhost:1234 " 之类的URL连接浏览器。如果能够实现，则浏览器将发出一个关于你正在使用服务端证书（self-signed certificate）的警告消息。如果同意该消息，会看到服务器的输出。

21.7 SocketServer

在Python 3中该模块称为SocketServer 。SocketServer 模块包括很多可以简化TCP、UDP和UNIX域套接字服务器实现的类。

21.7.1 处理程序

要使用该模块，必须定义一个继承自基类BaseRequestHandler 的处理程序类。BaseRequest-Handler 的实例h 可以实现以下方法。

```
h

.finish()
```

完成handle() 方法之后调用该方法执行清除操作。默认情况下，它不执行任何操作。如果setup() 或handle() 方法生成异常，则不会调用该方法。

h

.handle()

调用该方法执行实际的请求操作。调用该函数可以不带任何参数，但是有几个实例变量包含有用的值。 *h* .request 包含请求， *h* .client_address 包含客户端地址， *h* .server 包含调用处理程序的实例。对于TCP之类的数据流服务， *h* .request 属性是套接字对象。对于数据报服务，它是包含收到数据的字节字符串。

h

.setup()

在handle() 方法之前调用该方法进行初始化操作。默认情况下，它不执行任何操作。如果希望服务器实现更多连接设置（如建立SSL连接），那么可以在这里实现。

下例中的处理程序类实现了一个可以操作数据流或数据报的简单时间服务器：

```
try:
    from socketserver import BaseRequestHandler   # Python 3
except ImportError:
    from SocketServer import BaseRequestHandler   # Python 2
import socket
import time

class TimeServer(BaseRequestHandler):
    def handle(self):
        resp = time.ctime() + "\r\n"
        if isinstance(self.request,socket.socket):
            # 面向数据流的连接
            self.request.sendall(resp.encode('latin-1'))
        else:
            # 面向数据报的连接
            self.server.socket.sendto(resp.encode('latin-1'),self.client_address)
```

如果你知道处理程序只会操作面向数据流的连接（如TCP），那么它应该从 StreamRequest-Handler 而不是BaseRequestHandler 继承。该类设置两个属性： *h* .wfile 是将数据写入客户端的类文件对象； *h* .rfile 是从客户端读取数据的类文件对象。示例如下：

```
try:
```

```
    from socketserver import StreamRequestHandler # Python 3
except ImportError:
    from SocketServer import StreamRequestHandler # Python 2
import time
class TimeServer(StreamRequestHandler):
    def handle(self):
        resp = time.ctime() + "\r\n"
        self.wfile.write(resp.encode('latin-1'))
```

如果要编写仅对数据包进行操作的处理程序并将响应持续返回发送方，那么它应该继承自DatagramRequestHandler 而不是BaseRequestHandler 。它提供的类文件接口与StreamRequest-Handler 提供的相同。例如：

```
try:
    from socketserver import DatagramRequestHandler # Python 3
except ImportError:
    from SocketServer import DatagramRequestHandler # Python 2
import time

class TimeServer(DatagramRequestHandler):
    def handle(self):
        resp = time.ctime() + "\r\n"
        self.wfile.write(resp.encode('latin-1')
```

在本例中，写入self.wfile 的所有数据都被收集到一个数据包中，这个数据包在handle() 方法返回之后返回。

21.7.2 服务器

要使用处理程序，必须将其插入服务器对象。这里定义了4个基本的服务器类。

```
TCPServer(address

, handler

)
```

支持使用IPv4的TCP协议的服务器。 *address* 是一个形式为(*host, port*) 的元组。 *handler* 是BaseRequestHandler 类（见后文）的子类的实例。

```
UDPServer(address

, handler

)
```

支持使用IPv4的UDP协议的服务器。address 和handler 与TCPServer() 中的相同。

```
UnixStreamServer(address
, handler
)
```

使用UNIX域套接字实现面向数据流协议的服务器。继承自TCPServer 。

```
UnixDatagramServer(address
, handler
)
```

使用UNIX域套接字实现数据报协议的服务器。该方法继承自UDPServer 。

所有4个服务器类的实例具有以下基本方法。

```
s
.fileno()
```

返回服务器套接字的整数文件描述符。该方法的存在使得通过轮询操作（如select() 函数）使用服务器实例成为可能。

```
s
.serve_forever()
```

处理无限的请求数。

```
s
.shutdown()
```

停止serve_forever()循环。

以下属性提供有关运行服务器配置的基本信息。

```
s
.RequestHandlerClass
```

由用户提供的请求处理程序类，会被传递给服务器构造函数。

```
s
.server_address
```

监听服务器的地址，如元组('127.0.0.1', 80)。

```
s
.socket
```

用于处理传入请求的套接字对象。

下面是以TCP服务器的形式运行`TimeHandler`的示例：

```
from SocketServer import TCPServer

serv = TCPServer(('',10000,TimeHandler)
serv.serve_forever()
```

下例以UDP服务器的形式运行处理程序：

```
from SocketServer import UDPServer

serv = UDPServer(('',10000,TimeHandler)
serv.serve_forever()
```

`SocketServer`模块的一个核心特征是处理程序与服务器之间是解耦的。也就是说，编写了处理程序之后，无需更改实现即可插入到各种服务器中。

21.7.3 定义自定义服务器

服务器往往需要特殊的配置来处理不同的网络地址族、超时期、并发和其他功能。可以通过继承上一节中介绍的4个基本服务器执行自定义。可以定义以下类属性来自定义底层网络套接字的基本设置。

```
Server

.address_family
```

服务器套接字使用的地址族。默认值是`socket.AF_INET`。如果要使用IPv6则可以使用`socket.AF_INET6`。

```
Server

.allow_reuse_address
```

布尔标志，指示套接字是否应该重用地址。在程序终止之后，如果需要在同一个端口立即重启服务器，那么该设置会很有用（其他情况下，你必须等待几分钟）。默认值

为False 。

Server

.request_queue_size

传递给套接字listen() 方法的请求队列大小。默认值为5。

Server

.socket_type

服务器使用的套接字类型，如socket.SOCK_STREAM 或socket.SOCK_DGRAM 。

Server

.timeout

服务器等待新请求的超时期（以秒为单位）。超时期结束后，服务器调用handle_timeout() 方法（见下文）并继续等待。该超时期不能 用于设置套接字超时期。但是，如果已经设置了套接字超时期，则使用该超时期而不是此值。

下例演示如何创建支持端口重用的服务器：

```
from SocketServer import TCPServer

class TimeServer(TCPServer):
    allow_reuse_address = True

serv = TimeServer(('',10000,TimeHandler)
serv.serve_forever()
```

如果需要，以下方法对于扩展继承自服务器的类是最有用的。如果你在自己的服务器中定义了这些方法，则必须确保在超类中调用相同的方法。

```
Server
.activate()
```

对服务器执行listen() 操作的方法。服务器套接字以self.socket 形式引用。

```
Server
.bind()
```

此方法对服务器执行bind() 操作。

```
Server
.handle_error(request
, client_address
)
```

此方法处理操作过程中发生的未处理异常。要获取有关上一个异常的信息，使用traceback 模块中的sys.exc_info() 或其他函数。

```
Server
.handle_timeout()
```

服务器发生超时时调用的方法。通过重新定义该方法并调整超时设置，你可以将其他处理整合到服务器事件循环中。

```
Server
.verify_request(request
, client_address
)
```

在进一步处理之前，如果需要验证连接，则可以重新定义该方法。如果你希望实现防火墙或执行某些验证，则应该定义该方法。

最后，还可以通过混合类获得更多服务器功能。这也是通过线程或进程分支添加并发性的方法。为了实现这些功能，该模块定义了以下类。

```
ForkingMixIn
```

将UNIX进程分支添加到服务器的混合类，可以让服务器服务多个客户端。类变量max_children 控制子进程的最大数量，timeout 变量确定收集僵尸进程的操作间隔时间。实例变量active_children 跟踪正在运行多少个活动进程。

```
ThreadingMixIn
```

修改服务器的混合类，可以让服务器通过线程服务多个客户端。对于可创建的线程数量没有任何限制。默认情况下，线程是非守护程序类，除非将类变量daemon_threads 设置为True 。

要向服务器添加这些功能，可以使用多重继承，其中首先列出混合类。例如，下例是一个分支时间服务器：

```
from SocketServer import TCPServer, ForkingMixIn

class TimeServer(ForkingMixIn, TCPServer):
    allow_reuse_address = True
    max_children = 10

serv = TimeServer(('',10000,TimeHandler)
serv.serve_forever()
```

由于并发服务器很常见，因此SocketServer 预定义了以下服务器类。

- ForkingUDPServer(address, handler)
- ForkingTCPServer(address, handler)
- ThreadingUDPServer(address, handler)
- ThreadingTCPServer(address, handler)

这些类实际上就是按照混合类和服务器类定义的。例如，下面是ForkingTCPServer 的定义：

```
class ForkingTCPServer(ForkingMixIn, TCPServer): pass
```

21.7.4 自定义应用服务器

其他库模块常常使用SocketServer 类实现应用级别协议（如HTTP和XML-RPC）的服务器。要自定义这些服务器，还可以继承并扩展为基本服务器操作定义的方法。例如，下例是一个分支XML-RPC服务器，仅接受回路接口上发起的连接。

```
try:
     from xmlrpc.server import SimpleXMLRPCServer       # Python 3
     from socketserver import ForkingMixIn
except ImportError:                                    # Python 2
     from SimpleXMLRPCServer import SimpleXMLRPCServer
     from SocketServer import ForkingMixIn

class MyXMLRPCServer(ForkingMixIn,SimpleXMLRPCServer):
    def verify_request(self, request, client_address):
        host, port = client_address
        if host != '127.0.0.1':
            return False
        return SimpleXMLRPCServer.verify_request(self,request,client_address)
# 用法示例
def add(x,y):
    return x+y
server = MyXMLRPCServer(("",45000))
server.register_function(add)
server.serve_forever()
```

需要使用xmlrpclib 模块来测试上例。运行上面的服务器，然后启动一个独立的Python进程：

```
>>> import xmlrpclib


>>> s = xmlrpclib.ServerProxy("http://localhost:45000

")
>>> s.add(3,4)
7
```

```
>>>
```

要测试是否拒绝连接，可以使用相同的代码，但必须是从网络中的另一台机器执行才可以。为此，你需要将“`localhost`”替换为运行服务器的机器的主机名。

---

① 中文版《UNIX网络编程（卷1：套接字联网API）》已由人民邮电出版社出版。——编者注

② 任务的内容是：“墙上有1 000万瓶美酒，拿下来，分出去，墙上还有999万瓶美酒。墙上有999万瓶美酒，拿下来，分出去，墙上还有998万瓶美酒……”

第22章　网络应用程序编程

本章介绍与网络应用程序协议有关的模块，包括HTTP、XML-RPC和SMTP。我们将在第23章介绍编写CGI脚本等Web编程主题，在第24章中将介绍用于处理通用网络相关数据格式的模块。

在组织与网络有关的库模块方面，Python 2和Python 3有着很大的区别。为了让你了解未来的发展趋势，本章使用Python 3库的组织形式，因为它更符合逻辑。但是，在本书编写之际，这两个Python版本的库模块所提供的功能基本上是相同的。因此，如果存在对应的Python 2模块名称，我也会在每一节中标出。

22.1　ftplib

ftplib 模块实现了FTP协议的客户端。我们一般不需要直接使用该模块，因为urllib 包提供了一个高级接口。但是，如果你想更好地控制FTP连接的底层细节，这个模块还是很有用的。要使用这个模块，最好先了解Internet RFC 959中描述的一些FTP细节。

下面定义的类用于建立FTP连接：

```
FTP([host
 [, user
 [, passwd
 [, acct
 [, timeout
]]]]])
```

创建表示FTP连接的对象。 *host* 是表示主机名称的字符串。 *user* 、 *passwd* 和 *act* 是可选的，分别表示用户名、密码和账户。如果没有指定参数，则必须显式调用 *connect()* 和 *login()* 方法初始化实际的连接。如果指定了 *host* ，那么将自动调用 *connect()* 。如果指定了 *user* 、 *passwd* 和 *act* ，则将调用login()。 *timeout* 表示超时期，以秒为单位。

FTP 的实例 *f* 具有以下方法。

```
f
```

```
.abort()
```

试图终止正在进行的文件传输过程。该方法可能依赖远程服务器。

```
f
.close()
```

关闭FTP连接。调用该方法之后，不可以再对FTP对象 *f* 执行其他操作。

```
f
.connect(host [, port
 [, timeout
]])
```

根据指定的主机和端口打开FTP连接。 *host* 是指定了主机名称的字符串。 *port* 是FTP服务器端口号（整数），默认值为端口21。 *timeout* 是超时期，以秒为单位。如果已经向FTP() 提供了主机名称，则不需要再调用该方法。

```
f
.cmd(pathname
)
```

将服务器上的当前工作目录改为 *pathnanme* 。

```
f
.delete(filename
)
```

从服务器删除文件 *filename* 。

```
f
.dir([dirname
 [,...[, callback
]]])
```

生成一个目录列表，该方法与'LIST' 命令的效果一样。 *dirname* 是可选的，可提供要列出的目录名称。另外，如果提供了其他参数，它们将作为附加参数传递给'LIST' 。如果最后一个参数 *callback* 是一个函数，则它将作为回调函数处理返回的目录列表数据。该回调函数的工作方式与retrlines() 方法所使用的回调是一样的。默认情况下，该方法将目录列表打印到sys.stdout 中。

```
f.login([user, [passwd[, acct]]])
```

使用指定的用户名、密码和账户登录服务器。 *user* 是提供用户名的字符串，默认值是'anonymous' 。 *passwd* 是包含密码的字符串，默认值是空字符串" 。acct 也是一个字符串，默认值是空字符串。如果已经向FTP() 提供了这些信息，则无需调用该方法。

```
f
.mkd(pathname
)
```

在服务器上新建一个目录。

```
f
.ntransfercmd(command
[, rest
])
```

该方法与transfercmd() 几乎一样，唯一的差别在于此方法返回的是元组(*sock, size*)，其中sock 是对应于数据连接的套接字对象， *size* 是期望的数据大小（以字节为单位），如果大小无法确定，则为None 。

```
f
.pwd()
```

返回包含服务器上当前工作目录的字符串。

```
f
.quit()
```

将'QUIT' 命令发送到服务器，关闭FTP连接。

```
f
.rename(oldname
```

```
, newname
)
```

重命名服务器上的文件。

```
f
.retrbinary(command
, callback
[, blocksize
[, rest
]])
```

使用二进制传输模式返回在服务器上执行命令的结果。 *command* 是指定相应文件检索命令的字符串，一般都是'RETR *filename* '。 *callback* 是每次接收数据块时调用的回调函数。该回调函数带有一个参数，用于接收字符串形式的数据。 *blcokzize* 是使用的最大数据块大小，默认值是8192字节。 *rest* 是可选的文件偏移量。如果提供了该参数，它将指定要从文件中的哪个位置开始传输内容。但是，并非所有的FTP服务器都支持该参数，因此可能会导致error_reply 异常。

```
f
.retrlines(command
[, callback
])
```

使用文本传输模式返回在服务器上执行命令的结果。 *command* 是指定命令的字符串，通常类似于'RETR *filename* '。 *callback* 是回调函数，每收到一行数据时都将调用该函数。该回调函数带有一个参数，是包含已收到数据的字符串。如果忽略 *callback* ，则返回的数据将打印到sys.stdout 中。

```
f
.rmd(pathname
)
```

从服务器删除目录。

```
f
.sendcmd(command
)
```

将一个简单的命令发送到服务器并返回服务器的响应。 *command* 是一个包含命令的字符串。该方法只能用于不涉及数据传输的命令。

```
f
.set_pasv(pasv
)
```

设置被动模式。pasv 是一个布尔值标志，为True 表示打开被动模式，为False 表示关闭被动模式。默认情况下，被动模式为打开状态。

```
f
.size(filename
)
```

返回filename 的大小（单位是字节）。如果出于某些原因无法确定该大小，则返回None 。

```
f
.storbinary(command
, file
 [, blocksize
])
```

在服务器上执行一个命令并使用二进制传输模式传送数据。 *command* 是指定底层命令的字符串。它几乎总是设置为'STOR *filename* '，其中 *filename* 是你希望放在服务器上的文件名。 *file* 是一个打开的文件对象，将使用file.read(*blocksize*) 读取其数据并将该数据传输到服务器。 *blocksize* 是传输中使用的数据块大小。默认情况下，它的值是8192字节。

```
f
.storlines(command
, file
)
```

在服务器上执行一个命令并使用文本传输模式传输数据。 *command* 是指定底层命令的字符串。它通常是'STOR filename' 。file 是一个打开的文件对象，将使用 *file* .readline() 读取其数据并将该数据发送到服务器。

```
f
.transfercmd(command
 [, rest
])
```

通过FTP数据连接发起传输。如果使用主动模式，则该方法将发送'PORT' 或者'EPRT' 命令，并接受来自服务器的最终连接。如果使用被动模式，则该方法将发送'EPSV' 或者'PASV' 命令，然后建立与服务器的连接。无论是哪种情况，在建立了数据连接之后，会发出FTP命令 *command* 。该函数将返回对应于打开的数据连接的套接字对象。可选的 *rest* 参数指定服务器上被请求文件的起始字节偏移量。但是，并非所有服务器都支持该参数，因此可能导致error_reply 异常。

示例

下例展示了如何使用该模块将文件上传到FTP服务器：

```
host     = "ftp.foo.com"
username = "dave"
password = "1235"
filename = "somefile.dat"

import ftplib
ftp_serv = ftplib.FTP(host,username,password)
# 打开要发送的文件
f = open(filename,"rb")
# 将文件发送到FTP服务器
resp = ftp_serv.storbinary("STOR "+filename, f)
# 关闭连接
ftp_serv.close
```

要从FTP服务器获取文档，可以使用urllib 包。例如：

```
try:
    from urllib.request import urlopen  # Python 3
except ImportError:
    from urllib2 import urlopen        # Python 2

u = urlopen("ftp://username:password@somehostname/somefile")
contents = u.read()
```

22.2 http 包

http包中包括了编写HTTP客户端和服务器的模块以及对状态管理的支持（cookie）。超文本传输协议（HTTP）是一个基于文本的简单协议，它的工作方式如下。

（1）客户端连接到HTTP服务器，然后发送以下形式的请求报头：

```
GET /document.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.61 [en] (X11; U; SunOS 5.6 sun4u)
Host: rustler.cs.uchicago.edu:8000
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
```

```
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8

Optional

 data


...
```

第一行定义请求类型、文档（选择符）和协议版本。请求行后面是一系列报头行，包括各种有关客户端的信息，如密码、cookie、缓存首选项和客户端软件。报头行后面是一个空白行，表示报头行结束。报头之后，是发送表单的信息或者上传文件的事件中可能出现的数据。报头中的每一行都应该使用回车符和换行符（'\r\n'）终止。

（2）服务器发送以下形式的响应：

```
HTTP/1.0 200 OK
Content-type: text/html
Content-length: 72883 bytes
...
Header: data



Data


...
```

服务器响应的第一行表示HTTP协议版本、成功代码和返回消息。响应行之后是一系列报头字段，包含返回文档的类型、文档大小、Web服务器软件、cookie等方面的信息。通过空白行结束报头，后面是所请求文档的原始数据。

下列请求方法最为常见。

方　　法	描　　述
GET	获取文档
POST	将数据发布到表单
HEAD	仅返回报头信息
PUT	将数据上传到服务器

表22-1详细介绍了服务器最常返回的响应代码。符号常量列是http.client 中预定义变量的名称，预定义变量保存响应代码整数值，而该值可以用于代码中，以提高程序的可读性。

表**22-1**　服务器常返回的响应代码

代　　码	描　　述	符号常量
成功代码（**2xx**）		
200	成功	OK
201	创建	CREATED
202	接受	ACCEPTED
204	无内容	NO_CONTENT
重定向（**3xx**）		
300	多种选择	MULTIPLE_CHOICES
301	永久移动	MOVED_PERMANENTLY
302	临时移动	MOVED_TEMPORARILY
303	不修改	NOT_MODIFIED
客户端错误（**4xx**）		
400	请求错误	BAD_REQUEST
401	未授权	UNAUTHORIZED
403	禁止	FORBIDDEN
404	未找到	NOT_FOUND
服务器错误（**5xx**）		

500	内部服务器错误	INTERNAL_SERVER_ERROR
501	未实现	NOT_IMPLEMENTED
502	网关错误	BAD_GATEWAY
503	服务不可用	SERVICE_UNAVAILABLE

请求和响应中出现的所有报头都以众所周知的RFC-822格式进行编码。每个报头的一般形式为 *Headername:data* ，可以在RFC中找到更为详细的说明。你一般不需要解析这些报头，因为Python通常都会在合适的时候帮你完成这项工作。

22.2.1 http.client（httplib）

http.client 模块为HTTP的客户端提供底层支持。在Python 2中，该模块称为httplib 。也可以使用urllib 包中的函数。该模块既支持HTTP/1.0，也支持HTTP/1.1，如果构建Python时支持OpenSSL，那么它还可以通过SSL进行连接。通常不需要直接使用该包，而应该考虑使用urllib 包。但是由于HTTP是非常重要的协议，在某些情况下，仅使用urllib 无法轻松处理一些底层细节问题。例如，有时可能需要使用GET 或POST 以外的命令发送请求。有关HTTP的更多详情，请参考RFC 2616（HTTP/1.1）和RFC 1945（HTTP/1.0）。

可以使用以下类建立与服务器的HTTP连接。

```
HTTPConnection(host
 [,port
])
```

创建HTTP连接。host 是主机名，port 是远程端口号。默认端口是80。返回一个HTTPConnection 实例。

```
HTTPSConnection(host
 [, port
 [, key_file=kfile
 [, cert_file=cfile
 ]]])
```

创建HTTP连接，但使用安全套接字。默认端口是443。key_file 和cert_file 是可选关键字参数，指定客户端验证所需要的客户端PEM格式私有密钥和证书链文件。但是，该方法不会执行服务器证书的验证。返回HTTPSConnection 实例。

HTTPConnection 或HTTPSConnection 的实例h 支持以下方法。

h

.connect()

初始化到提供给HTTPConnection() 或HTTPSConnection() 的主机和端口的连接。如果尚未实现连接，则其他方法将自动调用该方法。

h

.close()

关闭连接。

h

.send(*bytes*

)

将字节字符串(*bytes*) 发送到服务器。建议不要直接使用该函数，因为它可能破坏底层的响应/请求协议。通常在调用 *h* .endheaders() 之后向服务器发送数据时使用这个方法。

```
h
.putrequest(method
, selector
 [, skip_host
 [, skip_accept_encoding
]])
```

向服务器发送请求。 *method* 是一个HTTP方法，如'GET'或'POST'。 *selector* 指定要返回的对象，如'/index.html'。 *skip_host* 和 *skip_accept_encoding* 参数是禁止在HTTP请求中发送Host:和Accept-Encoding:报头的标志。默认情况下，这两个参数都为False。由于HTTP/1.1协议支持通过一个连接发送多个请求，因此如果连接的当前状态为禁止发布新请求，就会引发CannotSendRequest异常。

```
h
.putheader(header
, value
, ...
)
```

将RFC-822样式的报头发送到服务器。该方法将一行内容发送到服务器，包括报头、一个冒号、一个空格以及一个值。其他参数可以编码为报头文件中连续的行。如果 *h* 的状态为不允许发送报头，则会引发CannotSendHeader异常。

```
h
.endheaders()
```

将空白行发送到服务器，表示报头行结束。

```
h
.request(method
, url
 [, body
 [, headers
]])
```

将完整的HTTP请求发送到服务器。 *method* 和 *url* 对于 *h* .putrequest() 来说含义是相同的。 *body* 是包含数据的可选字符串，发送请求之后这些数据将上传到服务器。如果提供了 *body* ，Context-length: 报头将自动被设为合适的值。headers 是一个字典，包括了提供给 *h* .putheader() 方法的 *header* :*value* 对。

```
h
.getresponse()
```

从服务器获取响应，并返回可用来读取数据的HTTPResponse 实例。如果h 的状态为无法收到响应，则会引发ResponseNotReady 异常。

getresponse() 方法返回的HTTPResponse 实例r 支持以下方法。

```
r
.read([size
])
```

从服务器读取最多 *size* 个字节。如果忽略 *size* ，则返回该请求的所有数据。

```
r
```

```
.getheader(name
 [,default
])
```

获取响应报头。 *name* 是报头的名称。 *default* 是未找到报头的情况下返回的默认值。

```
r
.getheaders()
```

返回(*header, value*) 元组列表。

HTTPResponse 实例*r* 还具有以下属性。

```
r
.version
```

服务器使用的HTTP版本。

```
r
.status
```

服务器返回的HTTP状态代码。

```
```

```
r
.reason
```

服务器返回的HTTP错误消息。

```
r
.length
```

响应中剩余的字节数。

1. 异常

处理HTTP连接时可能出现以下异常。

异　　常	描　　述	异　　常	描　　述
HTTPException	所有关于HTTP错误的基类	UnknownTransferEncoding	未知的传输编码
NotConnected	进行了请求但是未连接	UnimplementedFileMode	文件模式未实现
InvalidURL	给定的URL或端口号错误	IncompleteRead	收到的数据不完整
UnknownProtocol	未知的HTTP协议号	BadStatusLine	收到未知的状态码

以下异常与HTTP/1.1连接的状态有关。因为HTTP/1.1可以通过一个连接发送多个请求/响应，所以关于何时可以发送请求和收到响应还要应用其他规则。按照错误的顺序执行操作将生成异常。

异　　常	描　　述	异　　常	描　　述
ImproperConnectionState	所有HTTP连接状态错误的基类	CannotSendHeader	无法发送报头
CannotSendRequest	无法发送请求	ResponseNotReady	无法读取响应

2. 示例

以下示例展示了如何使用HTTPConnection 类通过使用POST请求执行文件上传且内存利用十分高效，这种操作在urllib 框架中很难实现。

```
import os
try:
    from httplib import HTTPConnection     # Python 2
except ImportError:
    from http.client import HTTPConnection  # Python 3

BOUNDARY = "$Python-Essential-Reference$"
CRLF     = '\r\n'

def upload(addr, url, formfields, filefields):
    # 为表单字段创建区
    formsections = []
    for name in formfields:
        section = [
            '--'+BOUNDARY,
            'Content-disposition: form-data; name="%s"' % name,
            '',
            formfields[name]
            ]
        formsections.append(CRLF.join(section)+CRLF)

    # 收集要上传的所有文件信息
    fileinfo = [(os.path.getsize(filename), formname, filename)
                    for formname, filename in filefields.items()]

    # 为每个文件创建HTTP报头
    filebytes = 0
    fileheaders = []
    for filesize, formname,filename in fileinfo:
        headers = [
            '--'+BOUNDARY,
            'Content-Disposition: form-data; name="%s"; filename="%s"' % \
                        (formname, filename),
            'Content-length: %d' % filesize,
            ''
            ]
        fileheaders.append(CRLF.join(headers)+CRLF)
        filebytes += filesize

    # 关闭marker
    closing = "--"+BOUNDARY+"--\r\n"

    # 确定整个请求的长度
    content_size = (sum(len(f) for f in  formsections) +
                   sum(len(f) for f in  fileheaders) +
                   filebytes+len(closing))

    # 上传
    conn = HTTPConnection(*addr)
    conn.putrequest("POST",url)
    conn.putheader("Content-type", 'multipart/form-data; boundary=%s' % BOUNDARY)
```

```
    conn.putheader("Content-length", str(content_size))
    conn.endheaders()
    # 发送所有表单区
    for s in formsections:
        conn.send(s.encode('latin-1'))

    # 发送所有文件
    for head,filename in zip(fileheaders,filefields.values()):
        conn.send(head.encode('latin-1'))
        f = open(filename,"rb")
        while True:
            chunk = f.read(16384)
            if not chunk: break
            conn.send(chunk)
        f.close()
    conn.send(closing.encode('latin-1'))
    r = conn.getresponse()
    responsedata = r.read()
    conn.close()
    return responsedata

# 示例：上传一些文件。表单字段'name'、'email'
# 'file_1'、'file_2'等是远程服务器
# 所需要的（很明显这不是固定不变的）。
server     = ('localhost', 8080)
url        = '/cgi-bin/upload.py'
formfields = {
    'name' : 'Dave',
    'email' : 'dave@dabeaz.com'
}
filefields = {
    'file_1' : 'IMG_1008.JPG',
    'file_2' : 'IMG_1757.JPG'
}
resp = upload(server, url,formfields,filefields)
print(resp)
```

22.2.2 http.server（BaseHTTPServer、CGIHTTPServer和SimpleHTTPServer）

http.server模块提供了实现HTTP服务器的各种类。在Python 2中，该模块的内容分为3个库模块：BaseHTTPServer、CGIHTTPServer和SimpleHTTPServer。

1. HTTPServer

以下类实现了基本的HTTP服务器。在Python 2中，它位于BaseHTTPServer模块中。

```
HTTPServer(server_address

, request_handler

)
```

新建HTTPServer 对象。 *server_address* 是服务器侦听的表单元组(*host, port*)。request_handler 是派生自BaseHTTPRequestHandler 的处理程序类，我们将在后面介绍它。

HTTPServer 直接继承socketserver 模块中定义的TCPServer 。因此，如果希望以某种方式自定义HTTP服务器的操作，你可以从HTTPServer 继承并进行扩展。以下是定义多线程HTTP服务器仅接受特定子网络连接的方式：

```
try:
    from http.server import HTTPServer        # Python 3
    from socketserver import ThreadingMixIn
except ImportError:
    from BaseHTTPServer import HTTPServer     # Python 2
    from SocketServer import ThreadingMixIn

class MyHTTPServer(ThreadingMixIn,HTTPServer):
    def __init__(self,addr,handler,subnet):
        HTTPServer.__init__(self,addr,handler)
        self.subnet = subnet
    def verify_request(self, request, client_address):
        host, port = client_address
        if not host.startswith(subnet):
            return False
        return HTTPServer.verify_request(self,request,client_address)

# 服务器运行方式的示例
serv = MyHTTPServer(('',8080), SomeHandler, '192.168.69.')
serv.serve_forever()
```

HTTPServer 类只处理底层HTTP协议。要让服务器真正执行某些操作，你必须提供处理程序类。有两个内置的处理程序和基类可以用来自定义处理方式。我们将在下一节讨论。

2. SimpleHTTPRequestHandler 和CGIHTTPRequestHandler

如果想快速建立简单的独立Web服务器，你可以使用两个预置的Web服务器处理程序类。这些类不需要安装任何第三方Web服务器即可独立运行，如Apache。

```
CGIHTTPRequestHandler(request

, client_address

, server

)
```

提供当前目录及其所有子目录的文件。另外，如果某个文件位于特殊的CGI目录中（默认由设置为['/cgi-bin', '/htbin']的cgi_directories 类变量定义），处理程序将把该文件作为CGI脚本运行。处理程序支持GET 、HEAD 和POST 方法。但是，它不支持HTTP重定向（HTTP代码302），因此它只能用于更加简单的CGI应用程序。出于安全的考虑，CGI脚本使用nobody UID执行。在Python 2中，该类在CGIHTTPServer 模块中定义。

```
SimpleHTTPRequestHandler(request
, client_address
, server
)
```

提供当前目录及其所有子目录的文件。该类支持HEAD 和GET 请求。所有的IOError 异常都将导致“404 File not found”错误。试图访问目录将导致“403 Directory listing not supported”错误。在Python 2中，该类在SimpleHTTPServer 模块中定义。

这两个处理程序都定义了以下类变量，如果需要，可以通过继承更改。

```
handler
.server_version
```

向客户端返回服务器版本字符串。默认情况下，该变量设为'SimpleHTTP/0.6' 之类的字符串。

```
handler
.extensions_map
```

将后缀映射到MIME类型的字典。未识别的文件类型将被视为'application/octet-stream' 类型。

以下示例将使用这些处理程序类运行一个独立的Web服务器，该服务器能够运行CGI

脚本：

```
try:
    from http.server import HTTPServer, CGIHTTPRequestHandler  # Python 3
except ImportError:
    from BaseHTTPServer import HTTPServer            # Python 2
    from CGIHTTPServer import CGIHTTPRequestHandler
import os

# 进入文档根目录
os.chdir("/home/httpd/html")
# 在端口8080上启动CGIHTTP服务器
serv = HTTPServer(("",8080),CGIHTTPRequestHandler)
serv.serve_forever()
```

3. BaseHTTPRequestHandler

BaseHTTPRequestHandler 类是一个基类，定义自己的自定义HTTP服务器处理时可使用它。SimpleHTTPRequestHandler 和CGIHTTPRequestHandler 等预置处理程序都继承自该类。在Python 2中，该类在BaseHTTPServer 模块中定义。

```
BaseHTTPRequestHandler(request

, client_address

, server

)
```

处理程序基类，用来处理HTTP请求。收到连接时，将解析请求和HTTP报头。然后根据请求类型尝试执行do_*REQUEST* 形式的方法。例如，'GET' 方法调用do_GET()，'POST' 方法调用do_POST 。默认情况下，该类不执行任何操作，所以这些方法应该在子类中定义。

BaseHTTPRequestHandler 中定义了以下类变量，可以在子类中重定义这些变量。

```
BaseHTTPRequestHandler.server_version
```

指定服务器向客户端报告的服务器软件版本字符串，如'ServerName/1.2' 。

```
BaseHTTPRequestHandler.sys_version
```

Python系统版本，如'Python/2.6' 。

BaseHTTPRequestHandler.error_message_format

用于构建发送到客户端的错误消息的格式字符串。格式字符串适用于包含属性code 、message 和explain 的字典。例如：

```
'''<head>
   <title>Error response</title>
   </head>
   <body>
   <h1>Error response</h1>
   <p>Error code %(code)d.
   <p>Message: %(message)s.
   <p>Error code explanation: %(code)s = %(explain)s.
   </body>'''
```

BaseHTTPRequestHandler.protocol_version

响应中使用的HTTP协议版本。默认值是'HTTP/1.0' 。

BaseHTTPRequestHandler.responses

将HTTP整数错误码映射为描述该问题的双元素元组(*message, explain*)。例如，整数码404映射为("Not Found", "Nothing matches the given URI")。根据上文介绍的error_message_format 属性在创建错误消息时使用该映射中的整数码和字符串。

创建后用来处理连接时，BaseHTTPRequestHandler 的实例 *b* 具有以下属性。

属　性	描　述

b .client_address	元组(host, port) 形式的客户端地址
b .command	请求类型，如'GET'、'POST'、'HEAD'等
b .path	请求路径，如'/index.html'
b .request_version	请求中的HTTP版本字符串，如'HTTP/1.0'
b .headers	映射对象中存储的HTTP报头。要测试或提取报头的内容，可以使用 *headername in b.headers* 或 *headerval = b.headers[headername]* 等字典操作
b .rfile	读取可选输入数据的输入流。该属性在客户端上传数据时使用（如在POST 请求过程中）
b .wfile	将响应写回客户端的输出流

通常使用以下方法，或者在子类中重定义这些方法。

```
b
.send_error(code
 [, message
])
```

为失败的请求发送响应。 *code* 是HTTP数字响应码。 *message* 是可选的错误消息。调用log_error() 来记录错误。该方法使用error_message_format 类变量创建完整的错误响应，将其发送到客户端，然后关闭连接。调用该方法后不应该再执行其他操作。

```
b
.send_response(code
 [, message
])
```

为成功的请求发送响应。发送HTTP响应行，然后是Server 和Date 报头。 *code* 是HTTP响应码， *message* 是可选的消息。调用log_request() 来记录请求。

```
b
.send_header(keyword
, value
)
```

将MIME报头项写入输出流。keyword 是报头关键字， *value* 是它的值。该方法仅在send_response() 之后调用。

```
b
.end_headers()
```

发送一个空白行表示MIME报头结束。

```
b
.log_request([code
 [, size
]])
```

记录成功的请求。 *code* 是HTTP代码， *size* 是响应的大小，以字节为单位（如果可用）。默认情况下，调用log_message() 进行日志记录。

```
b
.log_error(format
, ...)
```

记录错误消息。默认情况下，调用log_message()进行日志记录。

```
b
.log_message(format
, ...)
```

向sys.stderr记录任意消息。 *format* 是对任何传递的附加参数所应用的格式字符串。每个消息的前面将附加客户端地址和当前时间。

下例创建了一个运行在独立线程中的自定义HTTP服务器，并且可以监视字典的内容，将请求路径解释为一个键。

```
try:
    from http.server import BaseHTTPRequestHandler, HTTPServer      # Python 3
except ImportError:
    from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer   # Python 2

class DictRequestHandler(BaseHTTPRequestHandler):
    def __init__(self,thedict,*args,**kwargs):
        self.thedict=thedict
        BaseHTTPRequestHandler.__init__(self,*args,**kwargs)

    def do_GET(self):
        key = self.path[1:]       # 跳过前导'/'
        if not key in self.thedict:
            self.send_error(404, "No such key")
        else:
            self.send_response(200)
            self.send_header('content-type','text/plain')
            self.end_headers()
            resp  = "Key  : %s\n" % key
            resp += "Value: %s\n" % self.thedict[key]
            self.wfile.write(resp.encode('latin-1'))

# 用法示例
d = {
    'name' : 'Dave',
```

```
    'values' : [1,2,3,4,5],
    'email' : 'dave@dabeaz.com'
}
from functools import partial
serv = HTTPServer(("",9000), partial(DictRequestHandler,d))

import threading
d_mon = threading.Thread(target=serv.serve_forever)
d_mon.start()
```

要测试该示例，运行服务器然后在浏览器中输入http://localhost:9000/name 或http://localhost:9000/values 之类的URL。如果成功，你将在浏览器中看到字典的内容。

本例还说明了如何使用额外的参数让服务器初始化处理程序类。一般来说，服务器使用传递给`__init__()`的预定义参数集创建处理程序。如果想添加更多的参数，可以如上例所示使用`functools.partial()`函数。这将创建包含额外参数的可调用对象，但却保留了服务器所需的调用签名。

22.2.3 `http.cookies`（`Cookie`）

`http.cookies`模块支持在服务器端处理HTTP cookie。在Python 2中，该模块称为`Cookie`。

`cookie`用于服务器实现会话、用户登录及相关功能时进行状态管理。要在用户浏览器上安装cookie，HTTP服务器通常向HTTP响应添加类似以下内容的HTTP报头：

```
Set-Cookie: session=8273612; expires=Sun, 18-Feb-2001 15:00:00 GMT; \
           path=/; domain=foo.bar.com
```

也可以在HTML文档的`<head>`部分嵌入JavaScript来添加cookie：

```
<SCRIPT LANGUAGE="JavaScript">
document.cookie = "session=8273612; expires=Sun, 18-Feb-2001 15:00:00 GMT; \
    Feb 17; Path=/; Domain=foo.bar.com;"
</SCRIPT>
```

`http.cookies`模块提供了一个类似字典的特殊对象，其中存储并管理着称为`Morsel`的`cookie`值集合，因此简化了生成`cookie`值的任务。每个`Morsel`都有名称、值，以及一组包含了可应用到浏览器的元数据的可选属性`{expires, path, comment, domain, max-age, secure, version, httponly}`。名称通常是一个简单的标识符，如`"name"`，不得与任何元数据名称（如`"expires"`或`"path"`）相同。值通常是一个简短的字符串。要创建`cookie`，只需像以下这样创建`cookie`对象即可。

```
c = SimpleCookie()
```

接下来，只需使用普通的字典赋值设置`cookie`值（`Morsel`）：

```
c["session"] = 8273612
c["user"] = "beazley"
```

Morsel的其他属性设置如下：

```
c["session"]["path"] = "/"
c["session"]["domain"] = "foo.bar.com"
c["session"]["expires"] = "18-Feb-2001 15:00:00 GMT"
```

要创建以HTTP报头集合形式表示的cookie 数据输出，可以使用c.output() 方法。例如：

```
print(c.output())
# 生成两行输出
# Set-Cookie: session=8273612; expires=...; path=/; domain=...
# Set-Cookie: user=beazley
```

浏览器将cookie 发回HTTP服务器时，它使用 *key=value* 字符串的编码形式，如"session= 8273612;user=beazley" 。不返回expires 、path 和domain 等可选属性。cookie 字符串通常位于HTTP_COOKIE 环境变量中，可供CGI应用程序读取。要恢复cookie 值，可以使用以下的代码：

```
c = SimpleCookie(os.environ["HTTP_COOKIE"])
session = c["session"].value
user    = c["user"].value
```

下文将更加详细地介绍SimpleCookie 对象。

```
SimpleCookie([input

])
```

定义cookie 对象，将cookie 值存储为简单的字符串。

cookie 实例c 提供了以下方法。

```
c

.output([attrs

 [,header
```

```
 [,sep

]]])
```

生成可用于在HTTP报头中设置cookie 值的字符串。 *attrs* 是一个可选列表，由要包含的可选属性（"expires" 、"path" 、"domain" 等）组成。默认情况下包含所有cookie 属性。 *header* 是要使用的HTTP报头（默认值是'Set-Cookie:' ）。 *sep* 是用来连接报头的字符，默认是换行符。

```
c

.js_output([attrs

])
```

生成包含JavaScript代码的字符串，在支持JavaScript的浏览器上执行时可用来设置cookie 。 *attrs* 是由要包含的属性组成的可选列表。

```
c

.load(rawdata

)
```

使用rawdata 中的数据加载cookie 实例 *c* 。如果 *rawdata* 是一个字符串，则默认其格式与CGI程序中HTTP_COOKIE 环境变量的格式相同。如果 *rawdata* 是字典，则通过设置 *c[key]=value* 解释所有 *key-value* 对。

在内部，用来存储cookie 值的key/value 对是Morsel 类的实例。Morsel 的实例 *m* 与字典类似，可以设置可选的"expires" 、"path" 、"comment" 、"domain" 、"max-age" 、"secure" 、"version"
和"httponly" 键。另外，Morsel 的实例m 具有以下方法和属性。

```
m
```

```
.value
```

包含cookie原始值的字符串。

```
m
.coded_value
```

包含可以通过浏览器发送和接收的cookie 编码值的字符串。

```
m
.key
```

cookie名称。

```
m
.set(key
,value
,coded_value
)
```

设置上文所示的 *m* .key、 *m*. value 和 *m* .coded_value 的值。

```
m
```

```
.isReservedKey(k
)
```

测试 *k* 是否为保留关键字，如"expires"、"path"、"domain"等。

```
m
.output([attrs
 [,header
]])
```

生成该Morsel的HTTP报头字符串。 *attrs* 是要包含的其他属性（"expires"、"path"等）组成的可选列表。 *header* 是要使用的报头字符串（默认值是'Set-Cookie:'）。

```
m
.js_output([attrs
])
```

输出执行时设置cookie的JavaScript代码。

```
m
.OutputString([attrs
])
```

返回不带任何HTTP报头或JavaScript代码的cookie字符串。

异常

如果在解析或生成cookie 值时发生错误，则引发CookieError 异常。

22.2.4 http.cookiejar（cookielib）

http.cookiejar 模块支持在客户端存储和管理HTTP cookie。在Python 2中，该模块称为cookielib 。

该模块的主要功能是提供可存储cookie 的对象，以便与访问网络文档时使用的urllib 包配合使用。例如，可以使用http.cookiejar 模块捕获cookie 并在后续连接请求时重新发送。它还可以用来处理包含cookie数据的文件，如各种浏览器创建的文件。

该模块定义了以下对象。

```
CookieJar()
```

管理HTTP cookie值、存储HTTP请求生成的cookie 、向传出HTTP请求添加cookie 的对象。整个cookie 都存储在内存中，对CookieJar 实例进行垃圾回收后cookie 也将丢失。

```
FileCookieJar(filename
 [, delayload
 ])
```

创建FileCookieJar 实例，获取cookie信息并将该信息存储到文件。 *filename* 是文件名。 *delayload* 为True 时支持延迟访问文件。也就是说，只有需要时才会读取文件或在文件中存储数据。

```
MozillaCookieJar(filename
 [, delayload
 ])
```

创建与Mozilla cookies.txt 文件兼容的FileCookieJar 实例。

```
LWPCookieJar(filename
 [, delayload
 ])
```

创建与libwww-perl Set-Cookie3 文件格式兼容的FileCookieJar 实例。

很少直接使用这些对象的方法和属性。如果需要了解它们的底层编程接口，请参考在线文档。相反，我们经常实例化某个cookie jar 对象，并将其插入到需要处理cookie 的某个对象中。22.4.1节提供了一个示例。

22.3 smtplib

smtplib 模块提供了一个底层SMTP客户端接口，该接口可以使用RFC 821和RFC 1869中描述的SMTP协议发送邮件。该模块包含很多底层函数和方法，在线文档中对此有详细介绍。但我们在此只介绍该模块最有用的部分，具体内容如下。

```
SMTP([host
 [, port
]])
```

创建表示SMTP服务器连接的对象。如果给定host ，它指定SMTP服务器的名称。port 是可选的端口号。默认端口是25。如果提供了host ，则将自动调用connect() 方法。否则，你需要对返回的对象手动调用connect() 才能建立连接。

SMTP 的实例 *s* 具有以下方法。

```
s
.connect([host
 [, port
]])
```

连接 *host* 上的SMTP服务器。如果忽略 *host* ，则连接本地主机('127.0.0.1')。 *port* 是可选的端口号，如果忽略，则默认为25。如果向SMTP()提供主机名，则不需要调用connect()。

```
s
.login(user
, password
)
```

如果需要验证，则登录服务器。 *user* 是用户名， *password* 是密码。

```
s
.quit()
```

向服务器发送'QUIT'命令，终止会话。

```
s
.sendmail(fromaddr
, toaddrs
, message
)
```

将邮件消息发送到服务器。 *fromaddr* 是包含发件人电子邮箱地址的字符串。 *toaddrs* 是包含收件人电子邮箱地址的字符串列表。 *message* 是一个字符串，包含完

全符合RFC-822规定的格式化消息。`emil` 包通常用于创建此类消息。有一点需要注意的，尽管 *`message`* 的形式是文本字符串，但它只能包含值位于0到127之间的有效ASCII字符，否则会出现编码错误。如果需要发送不同编码（如UTF-8）的消息，则首先将其编码到字节字符串中，然后将该字节字符串作为 *`message`* 提供。

示例

下例演示了如何使用该模块发送消息：

```
import smtplib
fromaddr = "someone@some.com"
toaddrs = ["recipient@other.com"]
msg = "From: %s\r\nTo: %s\r\n\r\n" % (fromaddr, ",".join(toaddrs))
msg += """
Refinance your mortgage to buy stocks and Viagra!
"""

server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

22.4 `urllib` 包

`urllib` 包提供了一个高级接口，用于编写需要与HTTP服务器、FTP服务器和本地文件交互的客户端。典型的应用程序包括从网页抓取数据、自动化、代理、Web爬虫等。这是可配置程度最高的库模块之一，因而此处不会介绍每一个细节，而是介绍这个包最常见的用法。

在Python 2中，`urllib` 功能分散在几个不同的库模块中，包括`urllib` 、`urllib2` 、`urlparse` 和`robotparser` 。在Python 3中，所有这些功能都合并且统一放置在`urllib` 包中。

22.4.1 `urllib.request`（`urllib2`）

`urllib.request` 模块提供很多可打开URL并从中获取数据的函数和类。在Python 2中，该功能位于模块`urllib2` 模块。

该模块最常见的用途是通过HTTP从Web服务器获取数据。例如，以下代码演示了获取网页最简单的方法：

```
try:
    from urllib.request import urlopen  # Python 3
except ImportError:
    from urllib2 import urlopen         # Python 2

u = urlopen("http://docs.python.org/3.0/library/urllib.request.html

")
data = u.read()
```

当然，与实际服务器交互时情况会复杂得多。例如，你必须考虑代理服务器、验证、cookie、用户代理以及其他问题。此模块支持所有这些功能，只是代码会更加复杂（参见后文）。

1. urlopen() 和请求

最简单明了的请求方式是用urlopen() 函数。

```
urlopen(url
 [, data
 [, timeout
]])
```

打开URL url 并返回类文件对象，可使用该对象读取返回的数据。简言之，url 可以是包含URL的字符串，也可以是Request 类实例。data 是URL编码的字符串，包含要上传到服务器的表单数据。如果提供了data ，那么实际使用的就是HTTP 'POST' 方法，而不是'GET' （默认值）。通常使用urllib.parse.urlencode() 之类的函数创建数据。timeout 是可选的超时期（以秒为单位），内部存在阻塞操作时使用。

urlopen() 返回的类文件对象u 支持以下方法。

方　法	描　述
u.read([nbytes])	以字节字符串形式读取nbytes 个数据
u.readline()	以字节字符串形式读取单行文本
u.readlines()	读取所有输入行并返回列表
u.fileno()	返回整数文件描述符
u.close()	关闭连接
u.info()	返回映射对象，该对象带有与URL关联的元信息。对HTTP来说，返回的服务器响应包含HTTP报头。对于FTP来说，返回的报头包含'content-length' 。对于本地文件来说，返回的报头包含'content-length' 和'content-type' 字段

u.getcode()	返回整数形式的HTTP响应代码，例如，成功时返回200，未找到文件时返回404
u.geturl()	返回所返回数据的实际URL，考虑可能发生的任何重定向问题

需要特别强调的是，类文件对象u 以二进制模式操作。如果需要以文本形式处理响应数据，则需要使用codecs 模块或类似方式解码数据。

如果在下载期间出现错误，会引发URLError 异常。这包括与HTTP协议本身有关的错误，如禁止访问或请求验证。对于此类错误，服务器返回的内容通常会提供额外的描述信息。要获取该内容，可将异常实例本身作为可以读取的类文件对象操作。例如：

```
try:
    u = urlopen("http://www.python.org/perl.html")
    resp = u.read()
except HTTPError as e:
    resp = e.read()
```

如果通过代理服务器访问网页，那么在使用urlopen() 时会遇到一个很常见的错误。例如，如果你的公司通过代理路由所有Web流量，那么请求将会失败。如果代理服务器不需要任何验证，那么只需要设置os.environ 字典中的HTTP_PROXY 环境变量（如os.environ['HTTP_PROXY'] = 'http://example.com:12345 '）就可以解决该问题。

对于简单的请求来说，urlopen() 的url 参数就是一个字符串，如'http://www.python.org' 。如果需要执行更加复杂的操作，如修改HTTP请求报头，可创建Request 实例并使用其作为url 参数。

```
Request(url
 [, data
 [, headers
 [, origin_req_host
 [, unverifiable
]]]])
```

新建Request 实例。 *url* 指定URL（如'http://www.foo.bar/spam.html '）。 *data* 是URL编码的数据，要通过HTTP请求上传到服务器。提供该参数时，它将HTTP请求类型从'GET' 改为'POST' 。headers 是一个字典，包含了可表示HTTP报头内

容的键值映射。 *origin_req_host* 设为事务的请求主机，通常是发出请求的主机名称。如果请求的是无法验证的URL，则 *unverifiable* 设为True 。无法验证 的URL的非正式定义是：不是用户直接输入的URL，如加载图像的页面中嵌入的URL。 *unverifiable* 的默认值是False 。

Request 的实例 *r* 具有以下方法。

```
r.add_data(data
)
```

向请求添加数据。如果请求是HTTP请求，则方法改为'POST' 。与前文所述的Request() 一样，data 是URL编码的数据。该方法不会将data 追加到之前设置的任何数据上，它使用data 替换之前的数据。

```
r.add_header(key
, val
)
```

向请求添加报头信息。 *key* 是报头名称， *val* 是报头值。两个参数的值都是字符串。

```
r
.add_unredirected_header(key
, val
)
```

向请求添加报头信息，但不会添加到重定向请求中。 *key* 和 *val* 的含义与add_header() 中的含义相同。

```
```

```
r
.get_data()
```

返回请求的数据（如果有）。

```
r
.get_full_url()
```

返回请求的完整URL.

```
r
.get_host()
```

返回将接收该请求的主机。

```
r
.get_method()
```

返回HTTP方法，可能是'GET' 或'POST' 。

```
r
.get_origin_req_host()
```

返回发起事务的请求主机。

```
r
.get_selector()
```

返回URL的选择器部分（如'/index.html'）。

```
r
.get_type()
```

返回URL类型（如'http'）。

```
r
.has_data()
```

如果数据是请求的一部分，则返回True。

```
r
.is_unverifiable()
```

如果请求是无法验证的，则返回True。

```
r
```

```
.has_header(header

)
```

如果请求具有报头 *header* ，则返回True 。

```
r

.set_proxy(host

, type

)
```

准备请求连接代理服务器。这将使用 *host* 替换原来的主机，使用 *type* 替换原来的请求类型。URL选择器部分设为原始的URL。

下例使用Request 对象更改urlopen() 使用的'User-Agent' 报头。如果你希望服务器认为你是通过Internet Explorer、Firefox或其他浏览器进行连接的，则可以使用该方法。

```
headers = {
    'User-Agent':
        'Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727)'
}

r = Request("http://somedomain.com/

",headers=headers)
u = urlopen(r)
```

2. 自定义**opener**

基本的urlopen() 函数不支持验证、cookie 或者其他高级HTTP功能。要支持这些功能，必须使用build_opener() 函数创建自己的自定义opener 对象：

```
build_opener([handler1

 [, handler2

, ... ]])
```

构建用于打开URL的自定义opener 对象。参数 *handler1* 、handler2 等都是特殊处理程序对象的实例。这些处理程序的目的是向得到的opener 对象添加各种功能。下表列出了所有可用的处理程序对象。

处理程序	描　　述
CacheFTPHandler	具有持久FTP连接的FTP处理程序
FileHandler	打开本地文件
FTPHandler	通过FTP打开URL
HTTPBasicAuthHandler	基本的HTTP验证处理
HTTPCookieProcessor	处理HTTP cookie
HTTPDefaultErrorHandler	通过引发HTTPError异常来处理HTTP错误
HTTPDigestAuthHandler	HTTP摘要验证处理
HTTPHandler	通过HTTP打开URL
HTTPRedirectHandler	处理HTTP重定向
HTTPSHandler	通过安全HTTP打开URL
ProxyHandler	通过代理重定向请求
ProxyBasicAuthHandler	基本的代理验证
ProxyDigestAuthHandler	摘要代理验证
UnknownHandler	处理所有未知URL的处理程序

默认情况下，始终使用处理程序ProxyHandler 、UnknownHandler 、HTTPHandler 、HTTPSHandler 、HTTPDefaultErrorHandler 、HTTPRedirectHandler 、FTPHandler 、FileHandler 和HTTPErrorProcessor 创建opener。这些处理程序提供基本的功能。其他处理程序可以作为额外的参数提供。但是，如果其他处理程序的类型与默认类型相同，则优先使用后提供的程序。例如，如果添加一个HTTPHandler 实例或者继承自HTTPHandler 的某个类，则应该使用该处理程序而不是默认值。

build_opener() 返回的对象具有open(*url* [, *data* [,*timeout*]]) 方法，其作用是根据各种处理程序提供的规则打开URL。open() 的参数与传递给urlopen() 函数的参数相同。

```
install_opener(opener

)
```

安装不同的opener 对象作为urlopen() 使用的全局URL opener 。opener 通常是build_opener() 创建的opener 对象。

以下几节将介绍如何为使用urlib.request 模块时出现的更加普遍的场景创建自定义opener 。

3. 密码验证

要处理涉及密码验证的请求，你可以创建一个opener ，往其中添加HTTPBasicAuthHandler 、HTTPDigest AuthHandler 、ProxyBasicAuthHandler 或ProxyDigestAuthHandler 的组合。这些处理程序都具有以下方法，可用来设置密码。

```
h

.add_password(realm

, uri

, user

, passwd

)
```

为给定的域和URI添加用户和密码信息。所有参数都是字符串。 *uri* 可以是URI序列，在这种情况下，用户和密码信息适用于序列中的所有URI。 *realm* 是与验证相关联的名称或描述信息。它的值取决于远程服务器。但是，它通常是与相关网页关联的常用名称。 *uri* 是与验证相关联的基础URL。 *realm* 和uri 的常见值类似于 ('Administrator', 'http://www.somesite.com ')。 *user* 和 *password* 分别指定用户名和密码。

下例演示如何设置带有基本验证方法的opener：

```
auth = HTTPBasicAuthHandler()
auth.add_password("Administrator","http://www.secretlair.com

","drevil","12345")

# 使用添加的验证创建opener
opener = build_opener(auth)

# 打开URL
u = opener.open("http://www.secretlair.com/evilplan.html

")
```

4. HTTP Cookie

如果要管理HTTP cookie，需要创建添加了HTTPCookieProcessor 处理程序的opener对象。例如：

```
cookiehand = HTTPCookieProcessor()
opener = build_opener(cookiehand)
u = opener.open("http://www.example.com/

")
```

默认情况下，HTTPCookieProcessor 使用http.cookiejar 模块中的CookieJar 对象。将不同类型的CookieJar 对象作为HTTPCookieProcessor 的参数提供，可以支持不同类型的cookie 处理方法。例如：

```
cookiehand = HTTPCookieProcessor(
               http.cookiejar.MozillaCookieJar("cookies.txt")
             )
opener = build_opener(cookiehand)
u = opener.open("http://www.example.com/

")
```

5. 代理

如果需要通过代理重定向请求，可创建ProxyHandler 实例。

```
ProxyHandler([proxies

])
```

创建通过代理路由请求的代理处理程序。参数 *proxies* 是一个字典，将协议名称（如'http'、'ftp' 等）映射到相应代理服务器的URL。

下例演示了如何使用该方法：

```
proxy = ProxyHandler({'http': 'http://someproxy.com:8080/

'}
auth  = HTTPBasicAuthHandler()
auth.add_password("realm","host", "username", "password")
opener = build_opener(proxy, auth)

u = opener.open("http://www.example.com/doc.html

")
```

22.4.2 urllib.response

这是一个内部模块，实现了urllib.request 模块中函数返回的类文件对象。没有公共API。

22.4.3 urllib.parse

urllib.parse 模块用于操作URL字符串，如"http://www.python.org" 。

1. URL解析（Python 2中的urlparse 模块）

URL的一般形式为"*scheme://netloc/path;parameters?query#fragment* "。另外，URL的 *netloc* 部分可能包含一个端口号，如"*hostname:port* "，也可能包含用户验证信息，如"*user:pass@hostname* "。以下函数用于解析URL：

```
urlparse(urlstring

 [, default_scheme

 [, allow_fragments

]])
```

在urlstring 中解析URL并返回ParseResult 实例。如果URL中没有任何内容，则 *default_scheme* 指定要使用的方案（"http"、"ftp" 等）。如果 *allow_fragments* 为0，则不支持片段标识符。ParseResult 实例 *r* 是一个指定的元组，形式为(*scheme, netloc, path, parameters, query, fragment*)。该实例还定义了以下只读属性。

属　　性	描　　述
r.scheme	URL方案说明符（如'http' ）
r.netloc	netloc说明符（如'www.python.org' ）
r.path	层次结构路径（如'/index.html' ）
r.params	用于最后一个路径元素的参数
r.query	查询字符串（如'name=Dave&id=42' ）
r.fragment	不带前置'#' 的片段标识符
r.username	如果netloc说明符的形式为'username:password@hostname' ，则该属性表示用户名（username） 部分
r.password	netloc说明符的密码（password ）部分
r.hostname	netloc说明符的主机名称（hostname ）部分
r.port	如果netloc说明符的形式为'hostname:port' ，则该属性表示端口号

使用r.geturl() 可将ParseResult 实例转换回URL字符串。

urlunparse(*parts*

)

将urlparse() 返回的元组表示形式的URL转换为URL字符串。parts 必须是元组或者具有6个组件的可迭代内容。

```
urlsplit(url
 [, default_scheme
 [, allow_fragments
]])
```

该方法与urlparse() 类似，不同之处在于URL的 *parameters* 部分在路径中保持不变。该方法可以解析参数附加在单个路径组件上的URL，如'*scheme://netloc/path1;param1path2; param2/ path3?query#fragment* '。返回的结果是SplitResult的 实例，是包含(*scheme, netloc, path, query, fragment*)的指定元组。还定义了以下只读属性。

属　　性	描　　述
r.scheme	URL方案说明符（如'http'）
r.netloc	Netloc说明符（如'www.python.org'）
r.path	层次结构路径（如'/index.html'）
r.query	查询字符串（如'name=Dave&id=42'）
r.fragment	不带前置'#'的片段标识符
r.username	如果netloc说明符的形式为'username:password@hostname'，则该属性表示用户名部分
r.password	netloc说明符的密码部分
r.hostname	netloc说明符的主机名部分
r.port	如果netloc说明符的形式为'hostname:port'，则该属性表示端口号

使用 *r* .geturl() 可将SplitResult 实例转换回URL字符串。

```
urlunsplit(parts
)
```

将urlsplit()创建的元组形式的URL转换为URL字符串。 *Parts* 是一个元组或者具有5个URL组件的可迭代内容。

```
urldefrag(url
)
```

返回元组(*newurl, fragment*)，其中 *newurl* 是去除了片段的 *url* 部分，*fragment* 是包含片段部分的字符串（如果有）。如果 *url* 中没有任何片段，则 *newurl* 与 *url* 相同， *fragment* 为空字符串。

```
urljoin(base
, url
 [, allow_fragments
])
```

通过组合基础URL base 和相对URL构造绝对URL。 *url.allow_fragments* 的含义与urlparse()中的含义相同。如果基础URL的最后一个组件不是目录，则去除最后的组件。

```
parse_qs(qs
 [, keep_blank_values
[, strict_parsing
]])
```

解析URL编码的（MIME类型为application/x-www-form-urlencoded ）查询字符串 *qs* ，并返回字典，其中键是查询变量名称，值是为每个名称定义的值列表。 *keep_blank_values* 是一个布尔值标志，控制如何处理空白值。如果为True ，则它们包含在字典中，值设置为空字符串；如果为False （默认值），则将其丢弃。 *strict_parsing* 是一个布尔值标志，如果为True ，则将解析错误转换为ValueError 异常。默认情况下会忽略错误。

```
parse_qsl(qs
 [, keep_blank_values
 [, strict_parsing
]])
```

该方法与parse_qs() 类似，不同之处在于返回的结果是一个(name, value) 对列表，其中name 是查询变量的名称，value 是值。

2．URL编码（**Python 2**中的**urllib** 模块）

以下函数可对组成URL的数据进行编码和解码。

```
quote(string
 [, safe
 [, encoding
 [, errors
]]])
```

使用适合URL内容的转义序列替换 *string* 中的特殊字符。字母、数字和下划线（_ ）、逗号（, ）、句号（. ）、连字符（- ）都保持不变。所有其他字符都转换为'%xx' 形式的转义序列。safe 提供由其他不应该带有引号的字符组成的字符串，默认为'/' 。encoding 指定对非ASCII字符使用的编码。默认情况下是utf-8。 *errors* 指定遇到编码错误时的操作，默认值是'strict' 。encoding 和errors 参数只能在Python 3中使用。

```
quote_plus(string
 [, safe
```

```
 [, encoding

 [, errors

]]])
```

调用quote()并使用加号替换所有空格。 *string* 、safe、quote()中的含义相同 *encoding* 和 *errors* 。

```
quote_from_bytes(bytes

 [, safe

])
```

该方法与quote()类似，但是接受字节字符串且不执行编码。返回的值为文本字符串。仅用于Python 3中。

```
unquote(string

 [, encoding

 [, errors

]])
```

使用转义序列对应的单字符替换'%xx'形式的转义序列。 *encoding* 和 *errors* 指定'%xx'转义中编码数据的编码和错误处理。默认编码是'utf-8'，默认的errors策略是'replace'。 *encoding* 和 *errors* 仅用于Python 3中。

```
unquote_plus(string

 [, encoding

 [, errors

]])
```

该方法与unquote() 类似，但使用加号替换空格。

```
unquote_to_bytes(string
)
```

该方法与unquote() 类似，但是不执行任何编码并返回字节字符串。

```
urlencode(query
 [, doseq
])
```

将query 中的查询值转换为一个URL编码的字符串，该字符串可以作为URL的query参数包括在内，或者可以作为POST 请求的一部分上传。query 是字典，或者是一个 (*key, value*) 对序列。得到的字符串是以'&' 字符分隔的'key=value' 对序列，其中key 和value 都使用quote_plus() 引用。 *doseq* 参数是一个布尔值标志，如果 *query* 中的 *value* 是序列，则应该设置为True ，表示同一个键有多个值。在这种情况下，将为 *value* 中的每个v 创建一个单独的'key=v' 字符串。

3. 示例

下例演示了如何将查询变量的字典转换为适于在HTTP GET 请求中使用的URL，以及如何解析URL：

```
try:
    from urllib.parse import urlparse, urlencode, parse_qsl   # Python 3
except ImportError:
    from urlparse import urlparse, parse_qsl                   # Python 2
    from urllib import urlencode

# 使用合适的编码查询变量创建URL示例
form_fields = {
    'name'  : 'Dave',
    'email' : 'dave@dabeaz.com',
    'uid'   : '12345'
}
form_data = urlencode(form_fields)
url = "http://www.somehost.com/cgi-bin/view.py?"+form_data
```

```
# 分解URL的示例
r = urlparse(url)
print(r.scheme)      # 'http'
print(r.netloc)      # 'www.somehost.com'
print(r.path)        # '/cgi-bin/view.py'
print(r.params)      # ''
print(r.query)       # 'uid=12345&name=Dave&email=dave%40dabeaz.com'
print(r.fragment)    # ''

# 提取查询数据
parsed_fields = dict(parse_qsl(r.query))
assert form_fields == parsed_fields
```

22.4.4 urllib.error

urllib.error 模块定义了urllib 包使用的异常。

```
ContentTooShort
```

在下载数据量小于预期的数据量（由'Content-Length' 报头定义）时引发。Python 2在urllib 模块中定义。

```
HTTPError
```

在HTTP协议发生问题时引发。该错误可以用来提醒需要验证之类的事件。该异常还可以用作文件对象，以读取服务器返回的与错误有关的数据。它是URLError 的子类。Python 2在urllib2 模块中定义。

```
URLError
```

处理程序检测到问题时引发的错误。它是IOError 的子类。异常实例的reason 属性包含相关问题的更多信息。Python 2在urllib2 模块中定义。

22.4.5 urllib.robotparser（robotparser）

`urllib.robotparser` 模块（Python 2中的`robotparser`）用于获取和解析用来指示Web爬虫的`'robots.txt'` 文件的内容。如需更多的使用信息，请参考在线文档。

22.4.6 注意

- `urllib` 包的高级用户能够以你可以想象的任何方式自定义其行为。这包括创建新的opener、处理程序、请求、协议等。该主题超出了本文的介绍范围，读者可参考在线文档了解更多的详细信息。
- Python 2的用户应该注意`urllib.urlopen()` 函数，这个函数的使用非常广泛，它从Python 2.6开始已经废弃，Python 3中不存在该函数。你应该使用`urllib2.urlopen()` 替代`urllib.urlopen()`，该函数与本文所述的`urllib.request.urlopen()` 函数功能相同。

22.5 `xmlrpc` 包

`xmlrpc` 包中包含了实现XML-RPC服务器和客户端的模块。XML-RPC是一种远程过程调用机制，使用XML进行数据编码，使用HTTP作为传输机制。底层协议不是特定于Python的，因此使用这些模块的程序可以与使用其他语言编写的程序交互。如需有关XML-RPC的更多信息，请访问http://www.xmlrpc.com 。

22.5.1 `xmlrpc.client`（`xmlrpclib`）

`xmlrpc.client` 模块用于编写XML-RPC客户端。在Python 2中，该模块称为`xmlrpclib` 。若要用作客户端，可以创建`ServerProxy` 实例：

```
ServerProxy(uri
 [, transport
 [, encoding
 [, verbose
 [, allow_none
 [, use_datetime
]]]]
```

`uri` 是远程XML-RPC服务器的位置，如`"http://www.foo.com/RPC2 "` 。如有必要，可以使用格式`"http://user:pass@host:port/path "` 将基本的验证信息添加到URI，其中`user:pass` 是用户名和密码。该信息使用Base-64位编码，放入传输的`'Authorization:'` 报头中。如果Python配置了OpenSSL支持，则还可以使用HTTPS。`transport` 指定一个工厂函数，可以创建底层通信使用的内部传输对象。该参数仅用在通过HTTP或HTTPS以外的某种连接使用XML-RPC时。在一般情况下无需提供该

参数（更多细节请参考在线文档）。 *encoding* 指定编码，默认是UTF-8。 *verbose* 如果为True，则显示一些调试信息。 *allow_none* 如果为True，则可以对远程服务器发送None 值。默认情况下，该参数是禁用的，因为对它的支持很少。 *use_datetime* 是一个布尔值标志，如果设置为True，则使用 *datetime* 模块表示日期和时间。默认情况下是False。

ServerProxy 的实例 *s* 透明地公开了远程服务器的所有方法。这些方法可以作为 *s* 的属性进行访问。例如，下面的代码从提供该服务的远程服务器获取当前时间：

```
>>> s = ServerProxy("http://www.xmlrpc.com/RPC2

")


>>> s.currentTime.getCurrentTime()


<DateTime u'20051102T20:08:24' at 2c77d8>
>>>
```

大部分情况下，调用RPC就像调用普通的Python函数一样。但是，XML-RPC协议仅支持有限的参数类型和返回值。

XML-RPC 类型	Python 对应类型
boolean	True 和False
integer	int
float	float
string	string 或unicode （仅包含XML中有效的字符）
array	包含有效XML-RPC 类型的任何序列
structure	包含字符串键和有效类型值的字典
dates	日期和时间（xmlrpc.client.DateTime）
binary	二进制数据（xmlrpc.client.Binary）

收到日期时，会将其存储到xmlrpc.client.DateTime 实例 *d* 中。 *d* .value 属

性以ISO 8601时间/ 日期字符串的形式存储日期。要将其转换为兼容`time` 模块的时间元组，可以使用 *d* `.timetuple()` 。收到二进制数据时，数据将存储到`xmlrpc.client.Binary` 实例 *b* 中。 *b* `.data` 属性以字节字符串的形式存储数据。注意，该字符串被视为Unicode编码，你必须使用正确的编码形式。如果它们包含ASCII，则可以发送Python 2原始字节字符串，但其他情况则不行。要解决这个问题，可以首先将其转换为Unicode字符串。

如果使用包含无效类型的参数进行RPC调用，则会收到`TypeError` 或`xmlrpclib.Fault` 异常。

如果远程XML-RPC服务器支持内省（introspection），则可用以下方法。

```
s
.system.listMethods()
```

返回字符串列表，列出XML-RPC服务器提供的所有方法。

```
s
.methodSignatures(name
)
```

给定方法名称`name` ，返回该方法可能使用的调用签名列表。每个签名都是逗号分隔的字符串类型列表（如`'string, int, int'` ），其中第一项是返回类型，其余项是参数类型。如果出现重载，可能会返回多个签名。在Python实现的XML-RPC服务器中，签名通常为空，因为函数和方法是动态类型的。

```
s
.methodHelp(name
)
```

给定方法的名称name，返回描述该方法用途的文档字符串。文档字符串可能包含HTML标记。如果没有文档可用，则返回空字符串。

xmlrpclib 模块中提供了以下实用工具函数。

```
boolean(value

)
```

通过 *value* 创建一个XML-RPC布尔值对象。该函数早于现有的Python布尔值类型出现，因此在老代码中可能会见到该方法。

```
Binary(data

)
```

创建包含二进制数据的XML-RPC对象。data 是包含原始数据的字符串。返回Binary 实例。返回的Binary 实例在发送过程中使用Base-64透明地进行编码/ 解码。要从Binary 实例 *b* 中提取二进制数据，可以使用 *b* .data 。

```
DateTime(daytime

)
```

创建包含日期的XML-RPC对象。 *daytime* 可以是ISO 8601格式的日期字符串，或者是time.local-time() 返回的时间元组或结构，或者是datetime 模块的datetime 实例。

```
dumps(params [, methodname

 [, methodresponse

 [, encoding

 [, allow_none

]]]])
```

将 *params* 转换为XML-RPC请求或响应，其中params 是一个参数元组，或者是Fault 异常实例。 *methodname* 是字符串形式的方法名称。 *methodresponse* 是一个布尔值标志。如果为True ，则结果是XML-RPC响应。在这种情况下，params 中只可以提供一个值。encoding 指定了所生成的XML中的文本编码，默认为UTF-8。 *allow_none* 是一个标志，指定参数类型是否支持None 。XML-RPC规范中没有明确提到None ，但是很多服务器支持它。默认情况下， *allow_none* 为False 。

loads(*data*

)

将包含XML-RPC请求或响应的data 转换为元组(*params, methodname*)，其中 *params* 是参数的元组， *methodname* 是包含方法名称的字符串。如果请求表示的是错误状态而不是实际值，则引发Fault 异常。

MultiCall(*server*

)

创建MultiCall 对象，允许将多个XML-RPC请求打包到一起并作为一个请求发送。如果同一台服务器上需要进行很多不同的RPC请求，那么该方法可以有效地优化性能。 *server* 是 *ServerProxy* 的实例，表示到远程服务器的连接。返回的MultiCall 对象的使用方式与ServerProxy 完全相同。但是，该方法不会立即执行远程方法，会将方法调用进行排队，直到MultiCall 作为函数被调用为止。一旦发生这种情况，将发出RPC请求。该操作的返回值是一个生成器，生成队列中每个RPC操作的返回结果。注意，只有在远程服务器提供system.multicall() 方法时，MultiCall() 才可用。

以下示例演示了MultiCall 的用法：

```
multi = MultiCall(server)
multi.foo(4,6,7)                  # 远程方法 foo
multi.bar("hello world")          # 远程方法 bar
multi.spam()                      # 远程方法 spam
# 现在实际发送XML-RPC请求并获取返回结果
foo_result、bar_result、spam_result = multi()
```

异常

xmlrpc.client 中定义了以下异常。

```
Fault
```

表示XML-RPC错误。faultCode 属性包含带有错误类型的字符串。faultString 属性包含与错误有关的描述性消息。

```
ProtocolError
```

表示与底层网络有关的错误，如URL错误或某种连接问题。url 属性包含引发该错误的URI。errcode 属性包含错误代码。errmsg 属性包含描述性字符串。headers 属性包含引发该错误的请求的所有HTTP报头。

22.5.2 xmlrpc.server （SimpleXMLRPCServer 和DocXMLRPCServer ）

xmlrpc.server 模块包含了实现不同种类XML-RPC服务器的类。在Python 2中，该功能位于两个不同的模块中：SimpleXMLRPCServer 和DocXMLRPCServer 。

```
SimpleXMLRPCServer(addr
 [, requestHandler
 [, logRequests
]])
```

创建侦听套接字地址 *addr* 的XML-RPC服务器（如('localhost',8080)）。*requestHandler* 是一个工厂函数，在收到连接时创建处理程序请求对象。默认情况下，它设置为SimpleXMLRPCRequest Handler，这是当前唯一可用的处理程序。*logRequests* 是一个布尔值标志，指示是否记录所传入的请求。默认值是True 。

```
DocXMLRPCServer(addr
 [, requestHandler
```

```
 [, logRequest
]]
```

创建文档XML-RPC，它还可以响应HTTP GET 请求（通常由浏览器发送）。如果收到请求，服务器将根据所有注册方法和对象中的文档字符串生成文档。其中参数的含义与SimpleXMLRPCServer 的含义相同。

SimpleXMLRPCServer 或DocXMLRPCServer 的实例 *s* 具有以下方法。

```
s
.register_function(func
 [, name
])
```

向XML-RPC服务器注册新函数func 。name 是用于该函数的可选名称。如果提供了name ，客户端将使用该名称访问该函数。该名称包含的字符可能不是有效的Python标识符，包括句号（.）。如果没有提供name ，那么将使用func 的实际函数名称。

```
s
.register_instance(instance
 [, allow_dotted_names
])
```

注册一个对象，用于解析没有使用register_function() 方法注册的方法名称。如果实例instance 定义了方法_dispatch(*self, methodname, params*)，则调用该方法处理请求。 *methodname* 是方法的名称， *params* 是包含参数的元组。_dispatch() 的返回值返回到客户端。如果没有定义_dispatch() 方法，则查看实例，检索方法名称是否匹配为instance 定义的方法名称。如果匹配，则直接调用该方法。 *allow_dotted_names* 参数是一个标志，表示检查方法名称时是否应该执行层次搜

索。例如，如果收到对方法foo.bar.spam 的请求，则该参数确定是否要搜索instance.foo.bar.spam 。默认情况下，该参数是False 。只有在验证了客户端的情况下才能设置为True 。否则，它将造成安全漏洞，让入侵者能够执行恶意Python代码。注意，一次只能注册一个实例。

```
s
.register_introspection_functions()
```

向XML-RPC服务器添加XML-RPC内省函数system.listMethods() 、system.methodHelp() 和system.methodSignature() 。system.methodHelp() 返回方法的文档字符串（如果有）。system. methodSignature() 函数只返回一个消息指出不支持的操作。（因为Python是动态类型的，因此类型消息可用。）

```
s
.register_multicall_functions()
```

将system.multicall() 函数添加到服务器，添加XML-RPC多调用函数支持。

DocXMLRPCServer 实例还提供以下方法。

```
s
.set_server_title(server_title
)
```

在HTML文档中设置服务器标题。该字符串位于HTML <title> 标记中。

```
s
.set_server_name(server_name
```

```
)
```

在HTML文档中设置服务器名称。该字符串出现在页面顶部的<h1> 标记中。

```
s
.set_server_documentation(server_documentation
)
```

将描述性段落添加到生成的HTML输出中。该字符串添加到服务器名称之后，在XML-RPC函数的描述之前。

尽管XML-RPC服务器常常作为独立的进程运行，但它也可以在CGI脚本中运行。以下类用于这种情况。

```
CGIXMLRPCRequestHandler([allow_none
 [, encoding
]])
```

CGI请求处理程序，操作方式与SimpleXMLRPCServer 相同。参数的含义与SimpleXMLRPCServer 中的含义相同。

```
DocCGIXMLRPCRequestHandler()
```

CGI请求处理程序，操作方式与DocXMLRPCServer 相同。注意，编写本书时，调用参数与CGIXMLRPCRequestHandler() 的不同。这可能是一个bug，因此你应该参考新版本的在线文档。

任何一个CGI处理程序的实例c 都拥有与普通XML-RPC相同的注册函数和实例方法。

但是，它们还定义了以下方法：

```
c

.handle_request([request_text

])
```

处理XML-RPC请求。默认情况下，从标准输入读取请求。如果提供了 *request_text* ，它将包含通过HTTP POST 请求收到的表单中的请求数据。

1. 示例

下面这个例子是一个独立服务器程序。它添加一个简单的函数add 。另外，它还将math 模块的所有内容作为实例添加，公开它包含的所有函数。

```
try:
    from xmlrpc.server import SimpleXMLRPCServer     # Python 3
except ImportError:
    from SimpleXMLRPCServer import SimpleXMLRPCServer # Python 2
import math

def add(x,y):
    "Adds two numbers"
    return x+y

s = SimpleXMLRPCServer(('',8080))
s.register_function(add)
s.register_instance(math)
s.register_introspection_functions()
s.serve_forever()
```

以下是用CGI脚本实现的同一个功能：

```
try:
    from xmlrpc.server import CGIXMLRPCRequestHandler     # Python 3
except ImportError:
    from SimpleXMLRPCServer import CGIXMLRPCRequestHandler # Python 2
import math

def add(x,y):
    "Adds two numbers"
    return x+y

s = CGIXMLRPCRequestHandler()
s.register_function(add)
s.register_instance(math)
s.register_introspection_functions()
```

```
s.handle_request()
```

要从其他Python程序访问XML-RPC函数，可以使用xmlrpc.client 或xmlrpclib 模块。下面是一个演示其工作原理的简短交互式会话：

```
>>> from xmlrpc.client import ServerProxy

>>> s = ServerProxy("http://localhost:8080

")

>>> s.add(3,5)

8
>>> s.system.listMethods()

['acos', 'add', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'exp',

'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf',

'pow', 'radians', 'sin', 'sinh', 'sqrt', 'system.listMethods',

'system.methodHelp', 'system.methodSignature', 'tan', 'tanh']

>>> s.tan(4.5)

4.6373320545511847
>>>
```

2．自定义高级服务器

XML-RPC服务器模块可以轻松地用于基本类型的分布式计算。例如，可将XML-RPC用作更好地控制网络上其他系统的协议，前提是这些系统都运行了合适的XML-RPC服务器。如果使用pickle 模块，还可以在系统之间传递更有趣的对象。

XML-RPC的问题之一是安全性。默认情况下，XML-RPC服务器作为网络上的开放服务运行，因此任何知道服务器地址和端口的人都可以连接它（除非受到防火墙的阻止）。另外，XML-RPC服务器对请求可以发送的数据量没有限制。攻击者可以发送载荷很大（可耗尽内存）的HTTP POST 请求让服务器崩溃。

如果想解决这些问题，就需要自定义XML-RPC服务器类或请求处理程序。所有的服务器类都继承自socketserver 模块中的TCPServer 。因此，可以使用与其他套接字服

务器类相同的方式自定义服务器（如添加线程、分支或验证客户端地址）。验证包装器可以围绕请求处理程序放置，方法是继承SimpleXMLRPCRequestHandler 或DocXMLRPCRequestHandler 并扩展do_POST() 方法。以下是限制进入请求大小的示例：

```
try:
    from xmlrpc.server import (SimpleXMLRPCServer,
                               SimpleXMLRPCRequestHandler)
except ImportError:
    from SimpleXMLRPCServer import (SimpleXMLRPCServer,
                                    SimpleXMLRPCRequestHandler)
class MaxSizeXMLRPCHandler(SimpleXMLRPCRequestHandler):
    MAXSIZE = 1024*1024 # 1MB
    def do_POST(self):
        size = int(self.headers.get('content-length',0))
        if size >= self.MAXSIZE:
            self.send_error(400,"Bad request")
        else:
            SimpleXMLRPCRequestHandler.do_POST(self)

s = SimpleXMLRPCServer(('',8080),MaxSizeXMLRPCHandler)
```

如果要添加基于HTTP的验证，也可以使用类似的方式实现。

第23章　Web编程

我们在建网站时经常使用Python，它在这方面的用途很多。首先，Python脚本可以简单有效地生成一组可供Web服务器交付的静态HTML页面。例如，脚本可以用来携带原始内容，并基于这些内容来创建网站上常见的其他功能（导航栏、侧栏、广告、样式表等）。这主要是文件处理和文本处理方面的问题，本书的其他部分已经对此进行了介绍。

其次，Python脚本还可以用来生成动态内容。例如，使用标准Web服务器（如Apache）的网站可以使用Python脚本动态处理某些请求。Python在这方面的使用主要涉及表单处理。例如，可能存在包含以下表单的HTML页面：

```
<FORM ACTION='/cgi-bin/subscribe.py' METHOD='GET'>
Your name : <INPUT type='Text' name='name' size='30'>
Your email address: <INPUT type='Text' name='email' size='30'>
<INPUT type='Submit' name='submit-button' value='Subscribe'>
</FORM>
```

在这个表单中，`ACTION`属性指定了一个Python脚本`subscribe.py`，提交该表单时将在服务器上执行这个脚本。

另一个涉及生成动态内容的常见情况是使用AJAX（Asynchronous JavaScript and XML，异步JavaScript和XML）。使用AJAX，可以将JavaScript事件处理程序与页面上特定的HTML元素关联起来。例如，鼠标悬停在某个文档元素时，可能会执行某个JavaScript函数，并向负责处理的Web服务器（可能通过Python脚本进行处理）发送HTTP请求。收到相关响应之后，将执行另一个JavaScript函数处理响应数据并显示结果。返回结果的方式有很多。例如，服务器可能以纯文本、XML、JSON或者其他格式返回数据。以下是实现悬停弹出菜单的HTML文档示例，鼠标移动到选定元素上时，将出现一个弹出窗口。

```
<html>
  <head>
  <title>ACME Officials Quiet After Corruption Probe</title>
  <style type="text/css">
    .popup { border-bottom:1px dashed green; }
    .popup:hover { background-color: #c0c0ff; }
  </style>
  </head>
  <body>
     <span id="popupbox"
          style="visibility:hidden; position:absolute; background-color:
#ffffff;">
      <span id="popupcontent"></span>
    </span>
    <script>
         /* 获取对弹出框元素的引用 */
         var popup = document.getElementById("popupbox");
         var popupcontent = document.getElementById("popupcontent");

         /* 从服务器获取弹出数据，并在收到数据时进行显示 */
         function ShowPopup(hoveritem,name) {
```

```
            var request = new XMLHttpRequest();
            request.open("GET","cgi-bin/popupdata.py?name="+name, true);
            request.onreadystatechange = function() {
                var done = 4, ok = 200;
                if (request.readyState == done && request.status == ok) {
                    if (request.responseText) {
                        popupcontent.innerHTML = request.responseText;
                        popup.style.left = hoveritem.offsetLeft+10;
                        popup.style.top  = hoveritem.offsetTop+20;
                        popup.style.visibility = "Visible";
                    }
                }
            };
            request.send();
        }

        /* 隐藏弹出框 */
        function HidePopup() {
            popup.style.visibility = "Hidden";
        }
    </script>

    <h3>ACME Officials Quiet After Corruption Probe</h3>
    <p>
    Today, shares of ACME corporation
     (<span class="popup" onMouseOver="ShowPopup(this,'ACME');"
           onMouseOut="HidePopup();">ACME</span>)
     plummetted by more than 75% after federal investigators revealed that
     the board of directors is the target of a corruption probe involving
     the Governor, state lottery officials, and the archbishop.
   </p>

  </body>
</html>
```

在本例中，JavaScript函数`ShowPopup()`向服务器上的Python脚本`popupdata.py`发出一个请求。该脚本的返回结果是一个HTML片段，稍后将在弹出窗口中显示。图23-1展示了它在浏览器中的效果。

图23-1　可能出现的浏览器显示效果，背景文本只是一个普通的HTML文档，弹出窗口是由`popupdata.py`脚本动态生成的

最后，如果使用了Python编写的框架，整个网站可能都运行在Python的控制之下。我们常常开玩笑说Python“拥有的Web编程框架比语言关键字还要多”。Web框架的主题不在本书讨论的范围内，建议访问http://wiki.python.org/moin/WebFrameworks 了解更多信息。

本章后面部分将介绍与底层接口相关的内置模块，Python将使用这些模块与Web服务器和框架进行交互。相关主题包括：CGI，一种通过第三方Web服务器访问Python的技巧；WSGI，一个中间件层，用于编写集成Python各种Web框架的组件。

23.1　`cgi`

cgi 模块用于实现CGI脚本，也就是Web服务器在处理来自表单的用户输入或者生成某种动态内容时通常会调用的程序。

提交对应于CGI脚本的请求时，Web服务器在子过程中执行CGI程序。CGI程序接收来自以下两个源的输入：sys.stdin 和服务器设置的环境变量。下表详细介绍了Web服务器设置的通用环境变量。

变　量	描　述	变　量	描　述
AUTH_TYPE	验证方法	QUERY_STRING	查询字符串
CONTENT_LENGTH	传入sys.stdin 的数据长度	REMOTE_ADDR	客户端的远程IP地址
CONTENT_TYPE	查询数据的类型	REMOTE_HOST	客户端的远程主机名
DOCUMENT_ROOT	文档根目录	REMOTE_IDENT	发出请求的用户
GATEWAY_INTERFACE	CGI修订版本字符串	REMOTE_USER	经过验证的用户名
HTTP_ACCEPT	客户端接受的MIME类型	REQUEST_METHOD	方法（'GET' 或'POST' ）
HTTP_COOKIE	网景持久性cookie值	SCRIPT_NAME	程序名称
HTTP_FROM	客户端电子邮件地址（一般禁用）	SERVER_NAME	服务器主机名
HTTP_REFERER	引用URL	SERVER_PORT	服务器端口号
HTTP_USER_AGENT	客户端浏览器	SERVER_PROTOCOL	服务器协议
PATH_INFO	传递的其他路径信息	SERVER_SOFTWARE	服务器软件的名称和版本
PATH_TRANSLATED	PATH_ INFO 的翻译版		

CGI程序将输出写入标准输出sys.stdout 。有关CGI编程的详细内容，请参考由Shishir编写的*CGI Programming with Perl Second Edition* ① 等图书。我们只需要知道两件事。第一，HTML表单的内容通过被称为查询字符串 的文本序列传递给CGI程序。在Python中，我们使用FieldStorage 类访问查询字符串的内容。例如：

```
import cgi
form = cgi.FieldStorage()
```

```
name = form.getvalue('name')         # 获取表单'name'字段的值
email = form.getvalue('email')       # 获取表单'email'字段的值
```

第二，CGI程序的输出由两部分组成：HTTP 报头和原始数据（通常是HTML ）。一般使用空白行分隔这两个部分。下面是一个简单的HTTP 报头：

```
print 'Content-type: text/html\r'     # HTML输出
print '\r'                             # 空白行（必须添加！）
```

输出的其他部分都是原始输出。例如：

```
print '<TITLE>My CGI Script</TITLE>'
print '<H1>Hello World!</H1>'
print 'You are %s (%s)' % (name, email)
```

HTTP报头一般使用Windows行结尾约定'\r\n' 结束。这就是示例中出现'\r' 的原因。如果需要通知错误，可以在输出中包含特殊的'Status:' 报头。例如：

```
print 'Status: 401 Forbidden\r'      # HTTP错误码
print 'Content-type: text/plain\r'
print '\r'                           # 空白行（必须添加！）
print 'You're not worthy of accessing this page!'
```

如果需要将客户端重定向到其他页面，可以创建如下所示的输出：

```
print 'Status: 302 Moved\r'
print 'Location: http://www.foo.com/orderconfirm.html

\r'
print '\r'
```

cgi 模块的大部分工作都可以通过创建FieldStorage 类的实例来完成。

```
FieldStorage([input

 [, headers

 [, outerboundary

 [, environ

 [, keep_blank_values

 [,
```

```
strict_parsing


]]]]]])
```

通过读取和解析传入环境变量或标准输入的查询字符串来读取表单内容。 *input* 指定从POST请求中读取表单数据的类文件对象。默认情况下使用sys.stdin 。 *headers* 和 *outerboundary* 是供内部使用的，不应该提供。 *environ* 是一个字典，可以从中读取CGI环境变量。keep_blank_values 是一个布尔标志，用于控制是否保留空白值。默认情况下为Flase 。 *strict_parsing* 也是一个布尔标志，如果出现任何解析问题，将导致异常。默认情况下为Flase 。

FieldStorage 的实例form 与字典的工作方式类似。例如， *f = form[key]* 会根据给定的参数 *key* 提取项。用这种方式提取的实例 *f* 可能是另一个FieldStorage 实例，也可能是一个MiniFieldStorage 实例。对 *f* 定义了以下属性。

属　　性	描　　述
f .name	如果指定，则为字段名称
f .filename	上传时使用的客户端文件名
f .value	字符串值
f .file	可以读取数据的类文件对象
f .type	内容类型
f .type_options	在HTTP请求的内容类型行指定的选项字典
f .disposition	'content-disposition' 字段；如果未指定，则为None
f .disposition_options	配置选项字典
f .headers	包含所有HTTP报头内容的类字典对象

可以使用以下方法提取表单的值。

```

```

```
form.getvalue(fieldname
 [, default
])
```

返回名为 *fieldname* 的给定字段值。如果一个字段定义了两次，那么该函数将返回定义的所有值组成的列表。如果提供了 *default* 参数，它将指定字段不存在时要返回的值。该方法值得注意：如果同一个表单字段名称在请求中出现两次，则返回值将是包含这两个值的列表。若要简化编程，可以使用form.getfirst()，它只返回首先找到的值。

```
form.getfirst(fieldname
 [, default
])
```

返回名为 *fieldname* 的字段定义的第一个值。如果提供了 *default* 参数，该参数将指定字段不存在时要返回的值。

```
form.getlist(fieldname
)
```

返回为 *fieldname* 定义的所有值组成的列表。即使仅定义了一个值，也会返回一个列表；如果不存在任何值，则返回一个空列表。

此外，cgi 模块还定义了一个MiniFieldStorage 类，该类只包含属性的名称和值。该类用来表示通过查询字符串传入的表单中的每个字段，而FieldStorage 用于包含多个字段和多部分数据。

访问FieldStorage 实例的方式与访问Python字典类似，其中键是表单上的字段名称。用这种方法访问时，返回的对象本身可能是针对多部分数据（内容类型是'multipart/form-data'）或文件上传的FieldStorage 实例、针对简单字段（内容类型是'application/x-www-form-urlencoded'）的MiniFieldStorage 实例，或者是此类实例组成的列表（表单包含多个名称相同的字段时）。例如：

```
form = cgi.FieldStorage()
if "name" not in form:
    error("Name is missing")
    return
name = form['name'].value       # 获取表单'name'字段的值
email = form['email'].value     # 获取表单'email'字段的值
```

如果字段表示上传的文件，那么访问value 属性将读取整个文件并以字节字符串形式存入内存。这种方法可能会占用服务器的大量内存，因此最好分成更小的数据片段读取上传数据——直接从file 属性读取。例如，下例逐行读取上传的数据：

```
fileitem = form['userfile']
if fileitem.file:
    # 这是一个上传的文件；计算行数
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

CGI脚本常常使用以下实用工具函数。

escape(*s*

[*, quote*

])

将字符串 *s* 中的字符'&' 、'<' 和'>' 转换为HTML安全序列，如'& ' 、''< ' 和'>' 。如果可选的标志 *quote* 为True ，则双引号（" ）也将转换为'" ' 。

parse_header(*string*

)

解析HTTP报头字段（如'content-type' ）之后提供的数据。数据将被分解为一个主值和一个次要参数的字典，并以元组的形式返回。例如，命令：

```
parse_header('text/html; a=hello; b="world"')
```

将返回结果：

```
('text/html', {'a':'hello', 'b':'world'}).
parse_multipart(fp, pdict)

parse_multipart(fp

,pdict

)
```

解析类型为'multipart/form-data' 的输入，该类型在文件上传中经常使用。 *fp* 是输入文件，pdict 是包含内容类型报头参数的字典。它返回将字段名称映射为值列表的字典。该函数无法处理嵌套的多部分数据，应该使用FieldStorage 类进行处理它们。

```
print_directory()
```

使用HTML格式化当前工作目录的名称并打印出来。得到的输出将发送回浏览器，这有助于调试工作。

```
print_environ()
```

创建由所有环境变量组成的列表（使用HTML格式化），用于调试。

```
print_environ_usage()
```

打印有用环境变量组成的列表（HTML格式），用于调试。

```
print_form(form

)
```

使用HTML格式化表单上提供的数据。 *form* 必须是FieldStorage 的实例。用于调试。

```
test()
```

编写一个极简的HTTP报头，将提供给脚本的所有信息以HTML格式打印。主要用于调试，确保CGI环境的设置正确。

23.1.1 CGI编程建议

在这个Web框架盛行的时代，CGI脚本编程似乎已经过时了。但是，如果你决定使用它，可以采用我们提供的这些可以简化工作的编程技巧。

首先，如果使用了很多print 语句来生成硬编码HTML输出，那么不要使用CGI脚本。如果这样做，得到的程序可能变成一团Python和HTML组成的乱麻，既不可读，也无法维护。这种情况下使用模板可能更好，至少可以使用string.Template 对象。下面是一个解释这个概念的示例：

```
import cgi
from string import Template

def error(message):
    temp = Template(open("errormsg.html").read())
    print 'Content-type: text/html\r'
    print '\r'
    print temp.substitute({'message' : message})

form = cgi.FieldStorage()
name  = form.getfirst('name')
email = form.getfirst('email')
if not name:
    error("name not specified")
    raise SystemExit
elif not email:
    error("email not specified")
    raise SystemExit

# 执行各种处理
confirmation = subscribe(name, email)
# 打印输出页面
values = {
   'name' : name,
   'email' : email,
   'confirmation: ': confirmation,
   # 在这里添加其他值……
}
temp = Template(open("success.html").read()
```

```
print temp.substitute(values)
```

在本例中，文件"error.html" 和"success.html" 是包含所有输出的HTML页面，但是用$variable 替代CGI脚本中使用的、动态生成的值。例如，"success.html" 类似于：

```
<HTML>
   <HEAD>
      <TITLE>Success</TITLE>
   </HEAD>
   <BODY>
   Welcome $name. You have successfully subscribed to our
   newsletter. Your confirmation code is $confirmation.
   </BODY>
</HTML>
```

脚本中的temp.substitute() 操作只是为了填充该文件中的变量。这种方法有一个明显的好处，即如果你想更改输出的样式，只需修改模板文件即可，无需修改CGI文件。有很多第三方模板引擎支持Python，其数量可能比Web框架还多。它们都采用模板的理念，构建方法成千上万。更多详细信息，请参见http://wiki.python.org/moin/Templating 。

其次，如果你需要通过CGI脚本保存数据，那么可以尝试使用数据库。尽管直接将数据写入文件很容易，但是Web服务器是并发操作的，如果不能采取措施正确地锁定和同步资源，那么文件很容易遭到破坏。数据库服务器及其相关的Python接口通常没有这种问题。因此，如果需要保存数据，可以尝试使用sqlite3 之类的模块或者用于MySQL等的第三方模块。

最后，如果你发现自己编写了很多CGI脚本和代码来处理HTTP的底层细节，如cookie、验证、编码等，那么可以考虑使用Web框架。使用框架的意义在于：你不用再担心这些细节问题——至少不用担心那么多。因此别再重复造轮子了。

23.1.2 注意

- 安装CGI程序的过程很大程度上取决于使用的Web服务器类型。程序通常保存在特殊的cgi-bin 目录中。服务器也可能需要进行再次配置。更多详细信息，请参考服务器文档或咨询服务器管理员。
- 在UNIX上，Python CGI程序可能需要设置相应的执行权限，并且程序的第一行应该如下所示：

  ```
  #!/usr/bin/env python
  import cgi
  ...
  ```

- 如果要简化调试，导入cgitb 模块，如import cgitb; cgitb.enable() 。这样做可以修改异常处理，让错误在Web浏览器中显示。
- 如果通过os.system() 或os.popen() 函数调用外部程序，注意不要将从客户端收到的任何字符串传递到shell。这是一个众所周知的安全漏洞，黑客可能利用这个漏洞

在服务器上执行任意shell命令（因为传递给这些函数的命令首先由UNIX shell解释而不是直接执行）。尤其要注意，如果没有进行完整的检查来确保字符串仅包含字母数字字符、破折号、下划线和句号，则不要将任何URL部分或表单数据传递给shell命令。

- 在UNIX上，不要对CGI程序使用`setuid` 模式。这是一个安全模式，并非所有机器都支持。
- 不要对该模块使用`'from cgi import *'` 。`cgi` 模块定义了很多名称和符号，有些你可能不需要在自己的命名空间使用。

23.2 cgitb

该模块提供了另一个可供选择的异常处理程序，在发生未捕获的异常时显示详细报告。报告包括源代码、参数值和本地变量。开发该模块的最初目的是帮助调试CGI脚本，但是它可以在任何应用程序中使用。

```
enable([display
 [, logdir
 [, context
 [, format
]]]])
```

启用特殊的异常处理。 *display* 是一个标志，决定发生错误时是否显示任何信息。默认值为1。`logdir` 指定一个目录，这时错误报告将写入文件而不是打印到标准输出。给定 *logdir* 时，每个错误报告都写入`tempfile.mkstemp()` 函数创建的独特文件。*context* 是一个整数，指定围绕发生错误的行所显示的源代码行数。`format` 是指定输出格式的字符串。`"html"` 格式表示HTML（默认值）。任何其他值都会生成纯文本格式。

```
handle([info
])
```

使用`enable()` 函数的默认设置处理异常。 *info* 是一个元组(*exctype, excvalue, tb*)，其中 *exctype* 是异常类型， *excvalue* 是异常值， *tb* 是回溯对象。这个元组通常使用`sys.exc_info()` 获得。如果忽略 *info* ，则使用当前异常。

注意

> 要在CGI脚本中启用特殊异常处理，请在脚本开头添加`import cgitb; enable()`。

23.3 wsgiref

WSGI（Python Web Server Gateway Interface，Python Web服务器网关接口）是一个Web服务器和Web应用程序之间的标准化接口，用于增进应用程序在不同Web服务器和框架之间的可移植性。该标准的官方说明见PEP 333（http://www.python.org/dev/peps/pep-0333）。更多有关该标准及其用法的信息，请参见http://www.wsgi.org 。`wsgiref` 包是一个用于测试、验证和简单部署的参考实现。

23.3.1 WSGI规范

使用WSGI，Web应用程序可以实现为一个函数或者一个带有两个参数的可调用对象 *webapp*（*environ*, *start_response*）。 *environ* 是环境设置的字典，至少必须具有下表中的值，其含义和名称与CGI脚本中使用的一样。

environ 变量	描　　述	*environ* 变量	描　　述
CONTENT_LENGTH	传入的数据长度	QUERY_STRING	查询字符串
CONTENT_TYPE	查询数据的类型	REQUEST_METHOD	方法（'GET' 或'POST'）
HTTP_ACCEPT	客户端接受的MIME类型	SCRIPT_NAME	程序名称
HTTP_COOKIE	网景持久性cookie值	SERVER_NAME	服务器主机名
HTTP_REFERER	引用URL	SERVER_PORT	服务器端口号
HTTP_USER_AGENT	客户端浏览器	SERVER_PROTOCOL	服务器协议
PATH_INFO	传递的其他路径信息		

此外，environ 字典还必须包含以下特定于WSGI的值。

environ 变量	描　　述
wsgi.version	表示WSGI版本的元组（例如，(1,0) 表示WSGI 1.0）。
wsgi.url_scheme	表示URL模式组件的字符串，如'http' 或'https' 。

wsgi.input	表示输入流的类文件对象。表单数据或上传数据等其他数据都是从这里读取的。
wsgi.errors	以文本模式打开的类文件对象，用于写入错误输出。
wsgi.multithread	布尔标志，如果应用程序可以被同一个进程内的另一个线程并发执行，则为True 。
wsgi.multiprocess	布尔标志，如果应用程序可以被另一个进程并发执行，则为True 。
wsgi.run_once	布尔标志，如果应用程序在执行过程的整个生命周期中只能执行一次，则为True 。

start_response 参数是一个 *start_response* (status, headers) 形式的可调用对象，应用程序可以用它来发起响应。 *status* 是一个字符串，如'200 OK' 或者'404 Not Found' 。 *headers* 是一个元组列表，每个(*name, value*) 元组分别对应于一个将被包括在响应中的HTTP报头，如（'Content-type', 'text/ html'） 。

Web应用程序函数将响应的数据或正文作为可迭代对象返回，生成一系列字节字符串或文本字符串，其中仅包含可以编码为单字节的字符（如兼容ISO-8859-1或Latin-1字符集的字符）。例如，字节字符串列表，或者生成字节字符串的生成函数。如果应用程序需要执行某种字符编码（如UTF-8），那么它必须自己完成。

下面是一个简单的WSGI应用程序，它从表单字段读取内容并生成某些输出，类似于 cgi 模块部分展示的那个示例：

```
import cgi
def subscribe_app(environ, start_response):
    fields = cgi.FieldStorage(environ['wsgi.input'],
                              environ=environ)

    name = fields.getvalue("name")
    email = fields.getvalue("email")

    # 各种处理

    status = "200 OK"
    headers = [('Content-type','text/plain')]
    start_response(status, headers)

    response = [
        'Hi %s. Thank you for subscribing.' % name,
        'You should expect a response soon.'
        ]
    return (line.encode('utf-8') for line in response)
```

本例中有几个关键的细节。首先，WSGI应用程序组件与任何特定的框架、Web服务器或者库模块集没有联系。在上例中，我们只使用了一个库模块——cgi ，这是因为它有一些函数可以很方便地解析查询变量。该示例演示了如何使用start_response() 函数发起响应并提供报头。响应本身是由字符串列表组成的。该应用程序中的最后一条语句是

一个生成器表达式，用于将所有字符串转换为字节字符串。如果使用的是Python 3，那么这是关键的步骤——所有WSGI应用程序都要求返回编码后的字节，而不是未编码的Unicode数据。

要部署WSGI应用程序，必须向你正使用的Web编程框架注册。注册过程请参考手册。

23.3.2 wsgiref 包

wsgiref 包为WSGI标准提供了一个参考实现，它支持在独立的服务器测试应用程序，也可以将应用程序作为普通的CGI脚本执行。

1. wsgiref.simple_server

wsgiref.simple_server 模块实现了一个可以运行WSGI应用程序的简易的独立HTTP服务器。这里我们只讨论其中两个函数。

```
make_server(host
, port
, app
)
```

在给定主机名 *host* 、端口号 *port* 上创建接受连接的HTTP服务器。 *app* 是实现WSGI应用程序的函数或可调用对象。要运行服务器，使用s.serve_forever()，其中 *s* 是返回的服务器实例。

```
demo_app(environ
, start_response
)
```

一个完整的WSGI应用程序，返回带有“Hello World”消息的页面。它可以作为make_server() 的app 参数，用来验证服务器是否正常工作。

以下是运行简单WSGI服务器的示例：

```
def my_app(environ, start_response):
    # 某个应用程序
```

```
    start_response("200 OK",[('Content-type','text/plain')])
    return ['Hello World']

if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    serv = make_server('',8080, my_app)
    serv.serve_forever()
```

2. wsgiref.handlers

wsgiref.handlers 模块包含一些处理程序对象，用来设置WSGI执行环境，以便应用程序能够在其他Web服务器中运行（如Apache下的CGI脚本）。以下列举几个对象。

```
CGIHandler()
```

创建可以在标准CGI环境中运行的WSGI处理程序对象。这个处理程序从标准环境变量和I/O流中收集信息（在cgi 库模块中介绍过）。

```
BaseCGIHandler(stdin
, stdout
, stderr
, environ
 [, multithread
 [, multiprocess
]])
```

创建可以在CGI环境中操作的WSGI处理程序，但是标准I/O流和环境变量的设置方式有所不同。 *stdin* 、stdout 和 *stderr* 指定标准I/O流的类文件对象。 *environ* 是一个环境变量字典，应该已经包含标准CGI环境变量。 *multithread* 和 *multiprocess* 都是布尔标志，用于设置wsgi.multithread 和wsgi.multiprocess 环境变量。默认情况下， *multithread* 为True， *multiprocess* 为False 。

```
SimpleHandler(stdin
, stdout
```

```
, stderr

, environ

 [, multithread

 [, multiprocess

]])
```

创建类似于BaseCGIHandler的WSGI处理程序，但是让底层应用程序直接访问 *stdin* 、stdout 、stderr 和 *environ* 。这个处理程序与BaseCGIHandler 稍有不同，后者提供了更多的逻辑以便正确地处理某些问题。（例如，在BaseCGIHandler 中，响应代码转换为Status: 报头。）

所有这些处理程序都有一个run(*app*) 方法，可以在处理程序中运行WSGI应用程序。下例的WSGI应用程序以传统的CGI脚本形式运行：

```
#!/usr/bin/env python
def my_app(environ, start_response):
    # 某个应用程序
    start_response("200 OK",[('Content-type','text/plain')])
    return ['Hello World']

from wgiref.handlers import CGIHandler
hand = CGIHandler()
hand.run(my_app)
```

3. wsgiref.validate

wsgiref.validate 模块中有一个函数可以使用“验证包装器”包装WSGI应用程序，以确保该应用程序和服务器都按照标准进行操作。

```
validator(app

)
```

新建用于包装WSGI应用程序 *app* 的WSGI应用程序。新应用程序使用跟 *app* 一样的方式透明运行，但是它添加了更多的错误检查机制来确保应用程序和服务器遵守WSGI标准。出现任何违反标准的情况都将导致AssertionError 异常。

下面是一个使用验证程序的示例：

```
def my_app(environ, start_response):
    # 某个应用程序
    start_response("200 OK",[('Content-type','text/plain')])
    return ['Hello World']

if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    from wsgiref.validate import validator
    serv = make_server('',8080, validator(my_app))
    serv.serve_forever()
```

注意

本节中的内容主要针对希望创建应用程序对象的WSGI用户。另一方面，如果你打算实现另一个基于Python的Web框架，那么请参考官方资料PEP 333，了解让框架支持WSGI的细节。如果使用的是第三方Web框架，那么应该参考框架文档，了解支持WSGI对象的详细信息。由于WSGI是一个官方规范，标准库中包含参考实现，因此一般来说框架都会对其提供某种程度的支持。

23.4 webbrowser

webbrowser 模块提供了支持在Web浏览器中打开文档的实用工具函数，而且可以跨平台使用。该模块主要用于开发和测试。例如，如果你编写了一个生成HTML输出的脚本，就可以使用该模块中的这些函数自动调用系统的浏览器来查看结果。

```
open(url

 [, new

 [, autoraise

]])
```

使用系统的默认浏览器显示 *url* 。如果 *new* 为0，将尽可能在浏览器的当前窗口中打开URL。如果 *new* 为1，将新建一个浏览器窗口。如果 *new* 为2，将在浏览器中使用一个新选项卡打开URL。如果 *autoraise* 为True，将把浏览器窗口提到最上层。

```
open_new(url

)
```

在默认浏览器的新窗口中显示URL，与open(*url* , 1) 相同。

```
open_new_tab(url
)
```

在默认浏览器的新选项卡中显示URL，与open(*url* , 2) 相同。

```
get([name
])
```

返回一个操作浏览器的控制对象。 *name* 是浏览器类型的名称，通常为字符串，如'netscape' 、'mozilla' 、'kfm' 、'grail' 、'windows-default' 、'internet-config' 或 'command-line' 。返回的控制对象包含open() 和open_new() 方法，接受的参数和执行的操作与前两个函数相同。如果忽略name ，则返回默认浏览器的控制对象。

```
register(name
, constructor
[, controller
])
```

注册新的浏览器类型，以便与get() 函数一起使用。 *name* 是浏览器的名称。调用不带参数的 *constructor* 可以创建控制对象，用于在浏览器中打开页面。 *controller* 是要使用的控制对象实例。如果提供该参数，将忽略 *constructor* ，参数可以为None 。

get() 函数返回的控制对象实例 *c* 具有以下方法。

```
c.open(url
[, new
])
```

与open() 函数相同。

```
c.open_new(url
)
```

与open_new() 函数相同。

---

① 《CGI编程——使用Perl（第2版）》（Shishir Gundavaram著，中国电力出版社，2001年出版）。——译者注

第24章　网络数据处理和编码

本章介绍与处理通用网络数据格式和编码（如base 64、HTML、XML和JSON）有关的模块。

24.1　base64

使用base64 模块可通过base 64、base 32或base 16编码将二进制数据编码/解码为文本。base 64通常用于在邮件附件中嵌入二进制数据，或者将其作为HTTP协议的部分。官方详细信息见RFC-3548和RFC-1421。

base 64编码将要编码的数据分为一个或多个24位组（3个字节）。然后每个24位组都将被分为4个6位组件。每个6位值都使用下表中的可打印ASCII字符表示：

值	编　　码	值	编　　码
0～25	ABCDEFGHIJKLMNOPQRSTUVWXYZ	62	+
26～51	abcdefghijklmnopqrstuvwxyz	63	/
52～61	0123456789	填充	=

如果输入流中的字节数不是3（24位）的倍数，那么将填充数据以组成完整的24位组。额外填充的内容将在编码的最后部分使用'=' 字符表示。例如，如果编码一个16字节的字符序列，那么将得到5个3字节组，剩下1个字节。剩余的字节位将被填充，形成一个3字节组。然后，这个组会生成两个base 64 字母表中的字符（前12位，包括8位真实数据），后跟序列'==' ，表示额外填充的位。有效的base 64 编码后端只能包含0个、1个（=）或两个（==）填充字符。

base 32编码将二进制数据分为一个或多个40位（5字节）组。每个40位组又分为8个5位组件。然后使用以下字母表编码每个5位值：

值	编　　码
0～25	ABCDEFGHIJKLMNOPQRSTUVWXYZ
26～31	2-7

与使用base 64一样，如果输入流后端无法组成40位组，那么将填充为40位，并在输出中使用'='字符表示填充的位。最多只能有6个填充字符（'======'），这种情况出现在最后一个组只有1个数据字节时。

base 16编码是标准的十六进制数据编码。使用数字'0'～'9'和字母'A'～'F'表示每个4位组。base 16编码没有填充数字或字符。

```
b64encode(s
 [, altchars
])
```

使用base 64编码字节字符串 *s* 。如果指定 *altchars* ，则它应该是一个双字符字符串，指定可以用来替代base 64输出中常见字符"+"和"/"的字符。使用base 64编码文件名和URL时该函数会很有用。

```
b64decode(s
 [, altchars
])
```

解码使用base 64编码的字符串 *s* ，返回带有解码数据的字节字符串。如果指定 *altchars* ，则它应该是双字符字符串，指定可以用来替代base 64编码数据中常见字符"+"和"/"的字符。如果输出s包含无关字符或者填充错误，则会出现TypeError。

```
standard_b64encode(s
)
```

使用标准base 64编码字节字符串 *s* 。

```
standard_b64decode(s
)
```

解码使用标准base 64编码的字符串 *s* 。

```
urlsafe_b64encode(s
)
```

使用base 64编码字节字符串 *s* ，但是分别用字符"-" 和"_" 替代"+" 和"/" 。与b64encode(s, '-_') 相同。

```
urlsafe_b64decode(s
)
```

解码使用URL安全的base 64编码的字符串 *s* 。

```
b32encode(s
)
```

解码使用base 32编码的字节字符串 *s* 。

```
b32decode(s
 [, casefold
 [, map01
]])
```

解码使用base 32编码的字符串 *s* 。如果 *casefold* 为True ，则既接受大写也接受

小写。否则，只能出现大写字母（默认值）。如果有 *map01* ，则指定数字1映射到哪个字母（例如，是字母I还是字母L）。如果给定该参数，则将数字0映射到字母O。如果输出字符串包含无关字符或者填充错误，则会出现TypeError 。

```
b16encode(s
)
```

使用base 16（十六进制）编码字节字符串 *s* 。

```
b16decode(s
 [,casefold
])
```

解码使用base 16（十六进制）编码的字符串 *s* 。如果 *casefold* 为True ，则字母为大写或小写都可以。否则，十六进制字母A～F都必须为大写（默认值）。如果输出字符串包含无关字符或者格式错误，则会出现TypeError 。

以下函数是原来base 64模块接口的一部分，你仍然可以在现在的Python代码中看到它们。

```
decode(input
, output
)
```

解码base 64编码的数据。 *input* 是为进行读取而打开的文件名或文件对象。 *output* 是为进行二进制模式写入而打开的文件名或文件对象。

```
decodestring(s
)
```

解码base 64编码的字符串 *s* 。返回包含解码二进制数据的字符串。

```
encode(input
, output
)
```

使用base 64编码数据。 *input* 是为进行二进制模式读取而打开的文件名或文件对象。 *output* 是为进行写入而打开的文件名或文件对象。

```
encodestring(s
)
```

使用base 64编码字节字符串 *s* 。

24.2 binascii

binascii 模块包含一些底层函数，用来在二进制和各种ASCII编码（如base 64、BinHex和UUencod）之间转换数据。

```
a2b_uu(s
)
```

将一行UUencode编码的文本s 转换为二进制，并返回一个字节字符串。行数据通常包含45个（二进制）字节，但最后一行可能少于该数量。行数据可能后跟空格。

```
b2a_uu(data
)
```

将二进制数据字符串转换为UUencode编码的ASCII字符行。 *data* 的长度不应该大于45个字节。否则将出现Error 异常。

```
a2b_base64(string
)
```

将base 64编码的文本字符串转换为二进制，并返回一个字节字符串。

```
b2a_base64(data
)
```

将二进制数据字符串转换为一行base 64编码的ASCII字符。如果通过电子邮件传送得到的输出结果，那么data 的长度不应该大于57个字节（否则将被截断）。

```
a2b_hex(string
)
```

将十六进制数字字符串转换为二进制数据。该函数也称为unhexlify(*string*)。

```
b2a_hex(data
)
```

将二进制数据字符串转换为十六进制编码。该函数也称为hexlify(*data*)。

```
a2b_hqx(string
)
```

将BinHex 4编码的数据字符串转换为二进制，不执行RLE（Run-Length Encoding）解压缩。

```
rledecode_hqx(data
)
```

将 *data* 中的二进制数据执行RLE解压缩。返回解压缩后的数据，如果数据输入不完整，将出现Incomplete 异常。

```
rlecode_hqx(data
)
```

对data 执行BinHex 4 RLE压缩。

```
b2a_hqx(data
)
```

将二进制数据转换为BinHex 4编码的ASCII字符串。 *data* 应该已经进行了RLE编码。同样， *data* 的长度应该能够被3整除，除非 *data* 是最后一个数据片段。

```
crc_hqx(data
, crc
)
```

计算字节字符串 *data* 的BinHex 4 CRC校验和。 *crc* 是校验和的起始值。

```
crc32(data
 [, crc
])
```

计算节字符串 *data* 的CRC-32校验和。 *crc* 是一个可选的初始CRC值。如果忽略， *crc* 默认为0。

24.3 CSV

csv 模块用于读取和写入逗号分隔值（CSV）组成的文件。CSV文件由文本行组成，每一行都由分隔符分隔的值组成，分隔符通常是逗号（,）或制表符。示例如下：

```
Blues,Elwood,"1060 W Addison","Chicago, IL 60613","B263-1655-2187",116,56
```

处理数据库和电子表格时经常会用到这种格式的变体。例如，数据库可能使用CSV格式导出表格，以便其他程序能够读取这些表格。字段包含分隔符时会增加复杂度。例如，在上例中，包含逗号的字段必须使用引号。这就是使用基本的字符串运算符（如 split（','） ）不足以处理这种文件的原因。

```
reader(csvfile
 [, dialect
 [, **fmtparams
]]
```

返回reader 对象，生成输入文件 *csvfile* 中每一行输入的值。 *csvfile* 可以是任何可迭代对象，只要每次迭代生成一个完整的文本行即可。返回的reader 对象是一个迭代器，每次迭代生成一列字符串。 *dialect* 参数既可以是包含某种方言名称的字符串，也可以是Dialect 对象。dialect 参数的目的主要是处理不同CSV编码之间的差异。该模块仅支持两个内置方言——'excel'（默认值）和'excel-tab'，但用户可以定义其他方言，本节后面将详细讨论这一点。 *fmtparams* 是一个关键字参数集合，可

以自定义方言的各个方面。可以指定的关键字参数如下所示。

关键字参数	描　　述
delimiter	用来分隔字段的字符（默认值为','）
doublequote	布尔标志，确定字段中出现引号（quotechar）时如何处理。如果为True，则使用双引号。如果为False，则在引号前添加转义字符（escapechar）。默认值为True
escapechar	当字段中出现分隔符且quoting 为QUOTE_NONE 时用作转义字符的字符。默认值为None
lineterminator	行终止序列（'\r\n' 是默认值）
quotechar	用来引用包含分隔符的字段的字符（'"' 为默认值）
skipinitialspace	如果为True，则忽略紧跟分隔符后的空格（False 为默认值）

```
writer(csvfile
 [, dialect
 [, **fmtparam
]])
```

返回可以用来创建CSV文件的写入对象。 *csvfile* 可以是支持write() 方法的类文件对象。 *dialect* 的含义与reader() 中的同名参数相同，用于处理各种CSV编码之间的不同。 *fmtparam* 的含义也类似read() 方法中的同名参数。但是，还有另一个关键字参数可用。

关键字参数	描　　述
quoting	控制输出数据的引号添加行为。它有4种设置：QUOTE_ALL （给所有字段添加引号）、QUOTE_MINIMAL （只给包含分隔符或以引号开头的字段添加引号）、QUOTE_NONNUMERIC （给所有数字字段添加引号）、QUOTE_NONE （不给任何字段添加引号）。默认值为QUOTE_MINIMAL

writer 对象实例 *w* 支持以下方法。

```
w.writerow(row
)
```

将一个数据行写入文件。 *row* 必须是字符串序列或数字序列。

```
w.writerows(rows
)
```

写入多行数据。 *rows* 必须是传递给writerow() 方法的行序列。

```
DictReader(csvfile
 [, fieldnames
 [, restkey
 [, restval
 [, dialect
 [,**fmtparams
]]]]])
```

返回可以像普通 *reader* 对象一样操作的对象，不同之处在于：读取文件时返回的是字典对象而不是字符串列表。 *fieldnames* 指定在返回的字典中用作键的字段名称。如果忽略该参数，将从输入文件的第一行取得字典键的名称。 *restkey* 提供用来存储超额数据的字典键名称，例如，行的数据字段比字段名称多时可以指定该参数。 *restval* 指定输入中缺少值的字段的默认值，在行中没有足够的字段时可以指定该参数。 *restkey* 和 *restval* 的默认值为None 。 *dialect* 和 *fmtparams* 的含义与reader() 中的同名参数含义相同。

```
DictWriter(csvfile
, fieldnames
```

```
 [, restval

 [, extrasaction

 [, dialect

 [,**fmtparams

]]]])
```

返回像普通writer 对象一样操作的对象，不同之处在于：将字典写入输出行。 *fieldnames* 指定写入文件的顺序和属性名称。如果写入的字典中缺少 *fieldnames* 指定的某个字段名称，那么写入 *restval* 的值。 *extrasaction* 是一个字符串，指定写入字典中的键未在 *fieldnames* 中列出时该如何做。 *extrasaction* 中的默认值是'raise'，即引起ValueError 异常。可以使用值'ignore'，忽略字典中的其他值。 *dialect* 和 *fmtparams* 的含义与writer() 中的同名参数含义相同。

DictWriter 实例 *w* 支持以下方法。

```
w.writerow(row

)
```

将数据行写入文件。 *row* 必须是将字段名称映射到值的字典。

```
w.writerows(rows

)
```

写入多个数据行。 *rows* 必须是传递到writerow() 方法的行的序列。

```
Sniffer()
```

创建Sniffer 对象，用于尝试并自动检测CSV文件的格式。

Sniffer 实例s 具有以下方法。

```
s.sniff(sample
 [, delimiters
])
```

查看sample 中的数据，并返回表示数据格式的相应Dialect 对象。 *sample* 是至少包含一行数据的CSV文件的一部分。如果提供了 *delimiters* ，那么它应该是包含字段分隔符的字符串。

```
s.has_header(sample
)
```

查看 *sample* 中的CSV数据，如果第一行是列标题的集合，则返回True 。

24.3.1 方言

csv 模块中的许多函数和方法都涉及特殊的 *dialect* 参数。这个参数的目的是为了适应CSV文件不同的格式约定（CSV没有官方的“标准”格式），例如，逗号分隔值和制表符定界值之间的不同、引号约定等。

可以通过继承自Dialect 类定义方言，且需定义相同的属性集，作为传递给reader() 和writer() 函数的格式化参数（delimiter 、doublequote 、escapechar 、lineterminator 、quotechar 、quoting 、skipinitialspace ）。

以下是用来管理方言的实用函数。

```
register_dialect(name
, dialect
)
```

在名称 *name* 下注册新的Dialect 对象dialect 。

unregister_dislect(*name*

)

删除名为 *name* 的Dialect 对象。

get_dialect(*name*

)

返回名为 *name* 的Dialect 对象。

list_dialects()

返回由所有已注册方言名称组成的列表。目前只有两种内置方言：'excel' 和'excel-tab' 。

24.3.2 示例

```
import csv
# 读取基本的CSV文件
f = open("scmods.csv","r")
for r in csv.reader(f):
    lastname, firstname, street, city, zip = r
    print("{0} {1} {2} {3} {4}".format(*r))

# 使用DictReader
f = open("address.csv")
r = csv.DictReader(f,['lastname','firstname','street','city','zip'])
for a in r:
    print("{firstname} {lastname} {street} {city} {zip}".format(**a))

# 写入基本的CSV文件
data = [
 ['Blues','Elwood','1060 W Addison','Chicago','IL','60613' ],
 ['McGurn','Jack','4802 N Broadway','Chicago','IL','60640' ],
]
f = open("address.csv","w")
w = csv.writer(f)
w.writerows(data)
```

```
f.close()
```

24.4 email 包

email 包提供了许多函数和对象，用于根据MIME标准表示、解析和操作编码后的电子邮件消息。

本节不准备详细介绍email 包的所有内容，大部分读者对此也不感兴趣。因此，本节后半部分将主要关注两个常见的问题：解析电子邮件消息以提取有用信息；创建电子邮件消息以便使用smtplib 模块发送。

24.4.1 解析电子邮件

email 模块在顶层提供了两个函数用于解析消息。

```
message_from_file(f
)
```

解析从类文件对象 *f* 读取的电子邮件消息， *f* 必须以文本模式打开。输入消息应该是一个完整的MIME编码的电子邮件消息，包括所有的报头、文本和附件。返回Message 实例。

```
message_from_string(str
)
```

解析电子邮件消息——从文本字符串 *str* 读取电子邮件消息。返回Message 实例。

上一个函数返回的Message 实例 *m* 模拟字典并支持以下查询消息数据的操作。

操　作	描　述
m [*name*]	返回报头 *name* 的值
m .keys()	返回由所有消息报头名称组成的列表

m .values()	返回由所有消息报头值组成的列表
m .items()	返回由包含消息报头名称和值的元组组成的列表
m .get(name [,*def*])	返回名为 *name* 的报头值。 *def* 指定在未找到值时返回的默认值
len(m)	返回消息报头的数量
str(m)	将消息转换为字符串。与as_string() 方法相同
name in *m*	如果 *name* 是消息报头的名称，则返回True

除了这些操作函数之外， *m* 还可以使用以下方法提取信息。

```
m
.get_all(name
 [, default
])
```

返回由名为 *name* 的报头的所有值组成的列表。如果不存在这样的报头，则返回 *default* 。

```
m
.get_boundary([default
])
```

返回消息'Content-type' 报头中的边界参数。一般来说，边界是一个类似于'================ 0995017162==' 的字符串，用于分隔消息的不同部分。如果不存在任何边界参数，则返回 *default* 。

```
```

```
m
.get_charset()
```

返回与消息负载相关联的字符集（例如，'iso-8859-1'）。

```
m
.get_charsets([default
])
```

返回由消息中出现的所有字符集组成的列表。对于多部分消息，该列表包含每个子部分的字符集。每个部分的字符集都是从消息'Content-type'报头中获取的。如果没有指定字符集或者‘Content- type’报头丢失，则该部分的字符集将设置为 *default* 指定的值（默认情况下为None）。

```
m
.get_content_charset([default
])
```

返回从消息的第一个'Content-type'报头中得到的字符集。如果没有找到报头或者没有指定任何字符集，则返回default。

```
m
.get_content_maintype()
```

返回主要的内容类型（例如，'text'或'multipart'）。

```
m
.get_content_subtype()
```

返回次要内容类型（例如，'plain' 或'mixed' ）。

```
m
.get_content_type()
```

返回包含消息内容类型的字符串（例如，'multipart/mixed' 或 'text/plain' ）。

```
m
.get_default_type()
```

返回默认的内容类型（例如，简单消息的'text/plain' ）。

```
m
.get_filename([default
])
```

返回从'Content-Disposition' 报头得到的filename 参数（如果有）。如果报头丢失或者没有filename 参数，则返回 *default* 。

```
```

```
m
.get_param(param
 [, default
 [, header
 [, unquote
]]])
```

电子邮件报头常常带有附加参数，如报头'Content-Type: text/plain; charset="utf-8"; format=flowed' 的'charset' 和'format' 部分。该方法返回特定报头参数的值。 *param* 是参数名称， *default* 是未找到参数时返回的默认值， *header* 是报头的名称， *unquote* 指定是否取消参数的引用。如果没有提供 *header* 值，将从'Content-type' 报头获取参数。 *unquote* 的默认值为True 。返回值可能是字符串，如果参数按照RFC-2231约定进行编码，那么返回值是一个3元组——(*charset, language, value*)。在这种情况下， *charset* 是类似于'iso-8859-1' 的字符串， *language* 是包含语言代码（如'en' ）的字符串， *value* 是参数值。

```
m
.get_params([default
 [, header
 [, unquote
]]])
```

将报头的所有参数以列表形式返回。 *default* 指定未找到报头时返回的值。如果忽略 *header* ，则使用'Content-type' 报头。 *unquote* 是一个标志，指定是否取消值的引用（默认值为True ）。返回列表的组成内容是元组(*name, value*)，其中 *name* 是参数名， *value* 是get_param() 方法返回的值。

```
m
.get_payload([i
 [, decode
```

```
]])
```

返回消息的负载。如果是一个简单消息，则返回包含消息正文的字节字符串 。如果消息是多部分消息，则返回包含所有子部分的列表。对于多部分消息，i 指定该列表中的可选索引。如果提供该参数，则仅返回它所指定的消息组件。如果 *decode* 为True，则根据可能出现的 'Content-Transfer- Encoding' 报头设置（例如，'quoted-printable' 、'base64' 等）解码负载。要解码简单消息（没有多个部分）的负载，将 *i* 设置为None 并将decode 设置为True，或者使用关键字参数指定 *decode* 。必须强调的是：返回的负载是包含原始内容的字节字符串。如果负载以UTF-8或者其他编码编码的文本表示，则应该对结果使用decode() 方法进行转换。

```
m
.get_unixfrom()
```

返回UNIX风格的'From ... ' 行（如果有）。

```
m
.is_multipart()
```

如果 *m* 多部分消息，则返回True 。

```
m
.walk()
```

创建迭代消息所有子部分的生成器，每个部分都使用Message 实例表示。该迭代对消息进行深度优先遍历。一般情况下，该函数可以用来处理多部分消息的所有组件。

最后，Message 实例有几个与底层解析处理有关的属性。

m

.preamble

出现在多部分消息中的文本，位于表示报头结束的空白行与第一次出现的多部分边界字符串（意味着消息的第一个子部分）之间。

m

.epilogue

出现在消息中的文本，位于最后一个多部分边界字符串和消息尾部后。

m

.defects

解析消息时找到的所有消息缺陷组成的列表。更多详细信息，请参考email.errors模块的在线文档。

下例演示了解析电子邮件消息时如何使用Message 类。以下代码读取一个电子邮件消息，打印有用报头的概要，打印消息的纯文本部分，然后保存附件。

```
import email
import sys

f = open(sys.argv[1],"r")           # 打开消息文件
m = email.message_from_file(f)     # 解析消息
# 打印发件人/收件人概要
print("From    : %s" % m["from"])
print("To      : %s" % m["to"])
print("Subject : %s" % m["subject"])
print("")

if not m.is_multipart():
    # 简单消息。只打印负载
    payload = m.get_payload(decode=True)
```

```
    charset = m.get_content_charset('iso-8859-1')
    print(payload.decode(charset))
else:
    # 多部分消息。    遍历所有子部分，并且
    #     1. 打印纯文本部分
    #     2. 保存附件
    for s in m.walk():
        filename = s.get_filename()
        if filename:
            print("Saving attachment: %s" % filename)
            data = s.get_payload(decode=True)
            open(filename,"wb").write(data)
        else:
          if s.get_content_type() == 'text/plain':
             payload = s.get_payload(decode=True)
             charset = s.get_content_charset('iso-8859-1')
             print(payload.decode(charset))
```

在本例中必须强调的是：提取消息负载的操作总是返回字节字符串。如果负载表示文本，那么还需要根据某种字符集进行解码。示例中的`m.get_content_charset()` 和 `payload.decode()` 操作负责执行这种转化。

24.4.2 编写电子邮件

要编写电子邮件，可以创建一个空的`Message` 对象实例（在`email.message` 模块中定义），也可以使用通过解析电子邮件创建的`Message` 对象（见上节）。

```
Message()
```

新建一个初始为空的消息。

`Message` 的实例 *`m`* 以使用以下方法组成一个包含内容、报头和其他信息的消息。

```
m
.add_header(name
, value
, **params
)
```

添加一个新的消息报头。name 是报头的名称， *value* 是报头值， *params* 关键字参数集，用于提供其他可选参数。例如，add_header('Foo','Bar',spam='major') 向消息添加报头行“Foo: Bar; spam="major”。

```
m
.as_string([unixfrom
])
```

将整个消息转换为字符串。 *unixfrom* 是一个布尔标志。如果设置为True，则第一行出现UNIX风格的'From ...' 行。默认情况下， *unixfrom* 为False。

```
m
.attach(payload
)
```

向多部分消息添加一个附件。 *payload* 必须是另一个Message 对象（例如，email.mime.text. MIMEText）。在内部， *payload* 将被添加到跟踪消息不同部分的列表中。如果不是多部分消息，可以使用set_payload() 将消息正文设置为简单的字符串。

```
m
.del_param(param
 [, header
 [, requote
]])
```

从报头 *header* 中删除参数 *param* 。例如，如果消息带有报头'Foo: Bar; spam="major"'，那么del_param('spam','Foo') 将删除报头的'spam="major"' 部

分。如果 *requote* 为True （默认值），那么在重新写入报头时将引用所有剩下的值。如果忽略header，将对'Content-type' 报头应用该操作。

```
m
.replace_header(name
, value
)
```

使用值 *value* 替代第一次出现的报头name 的值。如果未找到该报头，则出现 KeyError 。

```
m
.set_boundary(boundary
)
```

将消息的边界参数设置为字符串 *boundary* 。该字符串作为边界参数添加到消息的'Content-type' 报头。如果消息没有'Content-type' 报头，则出现 HeaderParseError 。

```
m
.set_charset(charset
)
```

设置消息使用的默认字符集。 *charset* 是'iso-8859-1' 或'euc-jp' 之类的字符串。设置字符集后，通常会向消息的'Content-type' 报头添加一个参数（例如，'Content-type: text/html; charset="iso -8859-1"' ）。

```
```

```
m
.set_default_type(ctype
)
```

将默认消息内容类型设置为 *ctype* 。 *ctype* 是包含MIME类型（如'text/plain'或'message/ rfc822'）的字符串。该类型不会存储在消息的'Content-type'报头中。

```
m
.set_param(param
, value
 [, header
 [, requote
 [, charset
 [, language
]]]])
```

设置报头参数的值。 *param* 是参数名称， *value* 是参数值。 *header* 指定报头的名称并且默认为'Content-type' 。 *requote* 指定添加参数之后是否重新引用报头中的所有值。默认情况下，该参数为True 。 *charset* 和 *language* 指定可选参数集和语言信息。如果提供这些参数，将根据RFC-2231编码参数。该函数将得到param*="'iso-8859-1'en-us'some%20value"之类的参数文本。

```
m
.set_payload(payload
 [, charset
])
```

将整个消息负载设置为 *payload* 。对于简单消息， *payload* 可以是包含消息正文的字节字符串。对于多部分消息， *payload* 是 *Message* 对象组成的列表。 *charset* 是可选的，指定用来编码文本的字符集（请参见set_charset ）。

```
m
.set_type(type
 [, header
 [, requote
]])
```

设置'Content-type' 报头中使用的类型。 *type* 是指定类型的字符串，如'text/plain' 或 'multipart/mixed' 。header 指定可以用来替代默认'Content-type' 报头的报头。 *requote* 引用已经添加到报头的值。默认情况下，该参数为True 。

```
m
.set_unixfrom(unixfrom
)
```

设置UNIX风格的'From ...' 行的文本。 *unixfrom* 是一个包含完整文本（包括'From' 在内）的字符串。该文本只在 *m* .as_string() 的 *unixfrom* 参数设置为True 时才能输出。

为了避免每次都创建原始Message 对象并从头构建内容，Python提供了对应于各种内容类型的预置消息对象集。这些消息对象在创建多部分MIME消息时尤其有用。例如，可以新建一个消息，然后使用Message 的attach() 方法添加各种部件。所有这些对象都定义在不同的子模块中，下面逐个进行说明。

```
MIMEApplication(data
 [, subtype
 [, encoder
```

```
 [, **params

]]])
```

在email.mime.application 中定义。创建包含应用程序数据的消息。 *data* 是包含原始数据的字节字符串。 *subtype* 指定数据子类型，默认情况下是'octet-stream' 。 *encoder* 是email.encoders 子包中的一个可选编码函数。默认情况下，数据编码为 base 64。 *params* 表示添加到消息'Content-type' 报头的可选关键字参数和值。

```
MIMEAudio(data

 [, subtype

 [, encoder

 [, **params

]]])
```

在email.mime.audio 中定义。创建包含音频数据的消息。data 是包含原始二进制音频数据的字节字符串。 *subtype* 指定数据的类型，它可能是'mpeg' 或'wav' 之类的字符串。如果不提供 *subtype* ，将使用sndhdr 模块查看数据来猜测音频类型。 *encoder* 和 *params* 的含义与MIMEApplication 中的相同。

```
MIMEImage(data

 [, subtype

 [, encoder

 [, **params

]]])
```

在email.mime.image 中定义。创建包含图片数据的消息。 *data* 是包含原始图片数据的字节字符串。 *subtype* 指定图片类型，它可能是'jpg' 或'png' 之类的字符串。如果没有提供 *subtype* ，将使用 *imghdr* 模块中的函数来猜测类型。 *encoder* 和 *params* 的含义与MIMEApplication 中的相同。

```

```

```
MIMEMessage(msg
 [, subtype
])
```

在email.mime.message 中定义。新建非多部分MIME消息。 *msg* 是包含消息初始负载的消息对象。 *subtype* 是消息类型，默认值为'rfc822' 。

```
MIMEMultipart([subtype
 [, boundary
 [, subparts
 [, **params
]]])
```

在email.mime.multipart 中定义。新建多部分MIME消息。 *subtype* 指定要添加到'Content-type:multipart/subtype' 报头的可选子类型。默认情况下， *subtype* 为'mixed' 。 *boundary* 是一个字符串，指定用来组成每个消息子部分的边界分隔符。如果设置为None 或者忽略，则自动确定适用的边界。 *subparts* 是一系列组成消息内容的Message 对象。 *params* 表示添加到消息'Content-type' 报头的可选关键字参数和值。创建多部分消息后，即可使用Message.attach() 方法添加其他子部分了。

```
MIMEText(data
 [, subtype
 [, charset
]])
```

在email.mime.text 中定义。创建包含文本数据的消息。data 是包含消息负载的字符串。 *subtype* 指定文本类型，它是'plain' （默认值）或'html' 之类的字符串。 *charset* 是字符集，默认为'us-ascii' 。消息编码取决于消息的内容。

下例演示了如何使用本节所述的这些类组成并发送电子邮件消息。

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.audio import MIMEAudio

sender  = "jon@nogodiggydie.net"
receiver= "dave@dabeaz.com"
subject = "Faders up!"
body    = "I never should have moved out of Texas. -J.\n"
audio   = "TexasFuneral.mp3"

m = MIMEMultipart()
m["to"]      = receiver
m["from"]    = sender
m["subject"] = subject

m.attach(MIMEText(body))
apart = MIMEAudio(open(audio,"rb").read(),"mpeg")
apart.add_header("Content-Disposition","attachment",filename=audio)
m.attach(apart)

# 发送电子邮件消息
s = smtplib.SMTP()
s.connect()
s.sendmail(sender, [receiver],m.as_string())
s.close()
```

24.4.3 注意

- 还有许多高级自定义和配置选项没有讨论。读者可以参考在线文档了解该模块的高级用法。
- `email` 包已经经历了至少4个不同的版本，底层编程接口都有很多变化（即重命名子模块、改变类的位置等）。本节讨论的接口是在Python 2.6和Python 3.0中使用的4.0版。如果需要处理遗留代码，基本的概念还是适用的，但是要注意调整类和子模块的位置。

24.5 hashlib

`hashlib` 模块实现了各种安全的散列和消息摘要算法，如MD5和SHA1。要计算散列值，首先调用以下函数，其名称与所表示的算法是相同的。

函　数	描　述	函　数	描　述
`md5()`	MD5散列（128位）	`sha256()`	SHA256散列（256位）
`sha1()`	SHA1散列（160位）	`sha384()`	SHA384散列（384位）
`sha224()`	SHA224散列（224位）	`sha512()`	SHA512散列（512位）

这些函数返回的摘要对象实例 *d* 具有以下接口。

方法或属性	描　　述
d .update(data)	使用新数据更新散列。 *data* 必须是字节字符串。重复调用与使用连续数据进行单次调用的效果相同
d .digest()	将摘要值作为原始字节字符串返回
d .hexdigest()	返回文本字符串，摘要的值编码为一系列十六进制数字
d .copy()	返回摘要副本。副本保留原摘要的内部状态
d .digest_size	所得散列的字节大小
d .block_size	散列算法的内部块字节大小

该模块还提供了另一个可供选择的构造接口：

```
new(hashname

)
```

新建一个摘要对象。 *hashname* 是一个'md5'或'sha256'之类的字符串，指定要使用的散列算法的名称。散列的名称必须是上述散列算法或者OpenSSL库（取决于安装）中的散列算法。

24.6 hmac

hmac 模块为HMAC（Keyed-Hashing for Message Authentication，加密散列消息确认）提供支持，RFC-2104对HMAC有详细描述。HMAC是一个用于消息验证的机制，它以加密的散列函数（如MD5和SHA-1）为构建基础。

```
new(key

 [, msg

 [, digest
```

```
]])
```

新建HMAC对象。在这里， *key* 是包含散列初始键的字符串， *msg* 包含要处理的初始数据， *digest* 是应该用于加密散列的摘要构造函数。默认情况下， *digest* 为`hashlib.md5` 。

通常，初始密钥值是使用加密强随机数字生成器随机确定的。

HMAC对象 *h* 具有以下方法。

```
h
.update(msg
)
```

将字符串 *msg* 添加到HMAC对象。

```
h
.digest()
```

返回目前为止处理的所有数据的摘要，并返回包含摘要值的字节字符串。字符串的长度取决于底层散列函数。在MD5中是16个字符。在SHA-1中是20个字符。

```
h
.hexdigest()
```

以十六进制数字字符串形式返回摘要。

```
h
.copy()
```

复制HMAC对象。

示例

hmac 模块主要用在需要验证消息发件人的应用程序中。为此，new() 的 *key* 参数必须是一个字节字符串，表示消息发件人和收件人都知道的密钥。发送消息时，发件人将新建带有给定密钥的HMAC对象，使用要发送的消息数据更新对象，然后将消息数据与得到的HMAC摘要值一起发送给收件人。收件人可以计算消息的HMAC摘要值（使用相同的密钥和消息数据），并与收到的摘要值进行对比，以验证消息。示例如下：

```
import hmac
secret_key = b"peekaboo"          # 只有我知道的字节字符串。
                                    # 通常可以使用通过os.urandom()或类似方法
                                    # 计算得到的随机字节字符串

data = b"Hello World"              # 要发送的消息

# 发送消息。out表示套接字或者
# 其他发送数据的I/O通道。
h = hmac.new(secret_key)
h.update(data)
out.send(data)                      # 发送数据
out.send(h.digest())                # 发送摘要

# 接收消息
# in表示套接字或者我们接受数据的其他I/O通道。
h = hmac.new(secret_key)
data = in.receive()                 # 获取消息数据
h.update(data)
digest = in.receive()               # 获取发件人发送的摘要
if digest != h.digest():
     raise AuthenticationError('Message not authenticated')
```

24.7 HTMLParser

该模块在Python 3中称为html.parser 。HTMLParser 模块定义了可以用来解析HTML和XHTML文档的类——HTMLParser 。要使用该模块，必须自定义继承自 HTMLParser 的类，并重新定义相应的方法。

```
HTMLParser()
```

这是用于创建HTML解析器的基类。它没有使用任何参数初始化。

HTMLParser 的实例h 具有以下方法。

```
h
.close()
```

关闭解析器并强制处理剩下的未解析数据。在所有的HTML数据都提供给解析器之后调用该方法。

```
h
.feed(data
)
```

将新数据提供给解析器。该数据将被立即解析。但是，如果数据不完整（例如，结束处HTML元素不完整），那么将缓存不完整的部分，并在下一次使用更多数据调用feed() 时重新解析。

```
h
.getpos()
```

以元组(line, offset) 形式返回该行的当前行号和字符偏移量。

```
h
.get_starttag_text()
```

返回对应于上一次打开的起始标记的文本。

```
h
.handle_charref(name
)
```

遇到字符引用（如'&#ref;'）时就会调用该处理程序。 *name* 是包含引用名称的字符串。例如，当解析'å'时，name 应该设置为'229'。

```
h
.handle_comment(data
)
```

当遇到注释时就会调用该处理程序。 *data* 是包含注释文本的字符串。例如，解析注释'<!--comment-->'时，data 应该包含文本'comment'。

```
h
.handle_data(data
)
```

调用该处理程序可以处理出现在标签之间的数据。 *data* 是包含文本的字符串。

```
h
```

.handle_decl(*decl*

)

调用该处理程序处理'<!DOCTYPE HTML ...>'之类的声明。 *decl* 是包含声明文本的字符串，但不包含前面的'<!'和后面的'>'。

h

.handle_endtag(*tag*

)

遇到结束标记时就会调用该处理程序。 *tag* 是转换为小写形式的标记名称。例如，如果结束标记是'</BODY>'，那么 *tag* 就是字符串'body'。

h

.handle_entityref(*name*

)

调用该处理程序可以处理实体引用，如'&name;'。 *name* 是包含引用名称的字符串。例如，如果解析'<'，name 将设置为'lt'。

h

.handle_pi(*data*

)

调用该处理程序处理'<?processing instruction>'之类的指令。 *data* 是包含

要处理指令文本的字符串，但不包括前面的'<?' 和后面的'>' 。对XHTML样式的指令'<?...?>' 调用该处理程序时， *data* 中将包括最后一个'?' 。

```
h
.handle_startendtag(tag
, attrs
)
```

该处理程序处理XHTML样式的空标记，如'<tagname="value".../>' 。 *tag* 是一个包含标记名称的字符串。 *attrs* 包含属性信息，它是一个形式为(name, value) 的元组组成的列表，其中 *name* 是转换为小写形式的属性名称， *value* 是属性值。提取值时，将替换引号和字符实体。例如，如果解析'<a href="http://www.foo.com "/>' ，那么tag 为'a' ， *attrs* 为[('href','http://www.foo.com ')] 。如果没有在继承类中定义，该方法的默认实现就会调用handle_starttag() 和handle_endtag() 。

```
h
.handle_starttag(tag
, attrs
)
```

该处理程序处理'<tag name="value" ...>' 之类的起始标记。 *tag* 和 *attrs* 的含义与handle_ startendtag() 中的描述相同。

```
h
.reset()
```

重置解析器，丢弃所有未处理的数据。

该模块提供了以下异常。

```
HTMLParserError
```

解析出错时出现的异常。该异常有3个属性。 *msg* 属性包含描述错误的消息，*lineno* 属性是发生解析错误的行号， *offset* 属性是行中的字符偏移量。

示例

下例使用`urllib` 包获取一个HTML文档，然后打印所有使用`'<a href="...">'` 声明指定的链接：

```
# printlinks.py
try:
    from HTMLParser import HTMLParser
    from urllib2 import urlopen
except ImportError:
    from html.parser import HTMLParser
    from urllib.request import urlopen
import sys

class PrintLinks(HTMLParser):
    def handle_starttag(self,tag,attrs):
        if tag == 'a':
              for name,value in attrs:
                    if name == 'href': print(value)

p    = PrintLinks()
u    = urlopen(sys.argv[1])
data = u.read()
charset = u.info().getparam('charset')     # Python 2
#charset = u.info().get_content_charset()  # Python 3
p.feed(data.decode(charset))
p.close()
```

在本例中必须注意的是，使用`urllib` 包获取的任何HTML都以字节字符串的形式返回。要执行正确的解析，必须根据文档字符集编码将其解码为文本。该示例演示了如何在Python 2和Python 3中实现这一点。

> 注意
>
> `HTMLParser` 的解析能力非常有限。实际上，对于复杂和/或有缺陷的HTML，解析器可能会中断。用户还会发现该模块不仅用途有限，而且更加底层。如果编写必须从HTML页面爬取数据的程序，可以考虑使用Beautiful Soup包（http://pypi.python.org/pypi/BeautifulSoup ）。

24.8 json

`json` 模块用于使用JSON（JavaScript Object Notation，JavaScript对象符号）序列化和反序列化对象。有关JSON的更多信息，请参考http://json.org ，其实它的格式只是JavaScript语法的子集。顺便提一下，JSON表示列表和字典的语法与Python几无区别。例如，JSON数组的编写形式为`[value1 , value2, ... ]`，JSON对象的编写形式为`{name:value, name:value, ... }`。

下表说明JSON值和Python值之间的映射关系。括号中列出的Python类型表示它在编码时可以接受，但解码时不会返回（返回列出的第一个类型）。

JSON类型	**Python**类型	**JSON**类型	**Python**类型
object	dict	true	True
array	list（tuple）	false	False
string	unicode（str，bytes）	null	None
number	int，float		

对于字符串数据，应该默认使用Unicode。如果在编码时遇到字节字符串，则默认使用`'unf-8'`将其解码为Unicode字符串（不过这是可以控制的）。解码时，JSON字符串总是以Unicode的形式返回。

以下函数用于编码/解码JSON文档：

```
dump(obj
, f
, **opts
)
```

将 *obj* 序列化为类文件对象 *f* 。 *opts* 表示关键字参数集合，可以通过指定它来控制序列化过程。

关键字参数	描　　述
skipkeys	布尔标志，控制当字典键（不是值）不是基本类型（字符串或数字等）时应该执行的操作。如果为`True`，则跳过这些键。如果为`False`（默认值），则出现`TypeError`

ensure_ascii	布尔标志，确定是否可以将Unicode字符串写入文件 *f* 。默认情况下，该参数为False 。只有在*f* 是可以正确处理Unicode的文件（如codecs 模块创建的文件，或者使用特定编码集打开的文件）时才设置为True
check_circular	布尔标志，确定检查了容器的循环引用。默认情况下，该参数为True 。如果设置为False 且遇到了循环引用，则出现OverflowError 异常。
allow_nan	布尔标志，确定是否序列化范围外的浮点值（如NaN、inf、-inf）。默认情况下，该参数为True
cls	要使用的JSONEncoder 的子类。如果通过继承JSONEncoder 创建自己的解码程序，那么会指定该参数。如果向dump() 提供了其他关键字参数，那么它们将作为参数传递到此类的构造函数
indent	非负整数，设置打印数组和对象成员时使用的缩进量。设置该参数将进行整齐打印。默认值情况下，该参数为None ，使结果以最紧凑的表示形式显示
separators	形式为(*item _separator* , *dict_separator*) 的元组，其中 *item_separator* 是包含数组项之间所用分隔符的字符串，dict_separator 是包含字典键和值之间所用分隔符的字符串。默认情况下，该值为(*', ', ': '*)
encoding	Unicode字符串使用的编码，默认情况下，该参数为 *'utf-8'*
default	用于序列化非基本支持类型的函数。它应该返回可以序列化的值（即字符串）或者引发 *TypeError* 。默认情况下，不受支持的类型将引发 *TypeError*

```
dumps(obj
, **opts
)
```

与dump() 类似，不同之处在于返回包含结果的字符串。

```
load(f
, **opts
)
```

反序列化类文件对象 *f* 的JSON对象并返回。 *opts* 表示关键字参数集合，指定该参数可以控制解码流程，这将在下一节介绍。注意，该函数调用 *f* .read() 使用f 的所有内容。因此，不应该对任何流文件（如套接字的JSON数据）使用该函数，因为它可能是一个更大的或持续的数据流的一部分。

关键字参数	描　　述
encoding	用来解释任何解码字符串值的编码。默认情况下，该参数为'utf-8' 。
strict	布尔标志，确定是否允许JSON字符串中出现原义（未转义）换行符。默认情况下，该参数为True ， 表示会对这种字符串生成异常
cls	用于解码的JSONDecoder 子类。只有在通过继承JSONDecoder 创建自定义解码程序时才指定该参数。load() 的任何其他关键字参数都提供给类构造函数
object_hook	解码每个JSON对象的结果而调用的函数。默认情况下，这是内置的dict() 函数
parse_float	解码JSON浮点值时调用的函数。默认情况下，这是内置的float() 函数
parse_int	解码JSON整数值时调用的函数。默认情况下，这是内置的int() 函数
parse_constant	解码JSON常数（如'NaN' 、'true' 、'false' 等）时调用的函数

```
loads
(
s
, **
opts
)
```

与load() 类似，不同之处在于它从字符串s 反序列化对象。

尽管这些函数与pickle 和marshal 模块的函数具有相同的名称并且都用来序列化数据，但是它们的使用方式不同。具体来说，不能使用dump() 向同一个文件写入多个JSON编码的对象。同样，不能使用load() 从同一个文件读取多个JSON编码的对象（如果输入文件中有多个对象，将提示错误）。JSON编码对象应该视同为HTML或XML。例如，你

通常不会将两个完全不相关的XML文档合并到同一个文件中。

如果需要自定义编码或解码过程，可以考虑继承以下基类：

```
JSONDecoder(**opts
)
```

解码JSON数据的类。 *opts* 表示关键字参数集合，与`load()`函数使用的参数相同。`JSONDecoder`的实例 *d* 具有以下两个方法。

```
d
.decode(s
)
```

返回 *s* 中JSON对象的Python表示形式。 *s* 是一个字符串。

```
d
.raw_decode(s
)
```

返回一个元组(*pyobj, index*)，其中 *pyobj* 是 *s* 中JSON对象的Python表示形式， *index* 是JSON对象在 *s* 中结束的位置。希望从输入流（尾部有其他数据）解析对象时可以使用该方法。

```
JSONEncoder(**opts
)
```

将Python对象编码为JSON的类。 *opts* 表示关键字参数集合，与dump() 函数使用的参数相同。JSONEncoder 的实例 *e* 具有以下方法。

```
e
.default(obj
)
```

Python对象 *obj* 无法根据任何普通的编码规则编码时调用此方法。该方法返回可以编码的类型（如字符串、列表或字典）。

```
e
.encode(obj
)
```

创建Python对象 *obj* 的JSON表示形式时调用的方法。

```
e
.iterencode(obj
)
```

创建迭代器，在计算Python对象 *obj* 时生成组成其JSON表示形式的字符串。创建JSON字符串的过程本身是一个高度递归的过程。例如，它需要迭代字典关键字并在此过程中遍历找到的其他字典和列表。如果使用该方法，可以逐个处理输出而不是将所有内容收集到一个占用很大内存的字符串中。

定义从JSONDecoder 或JSONEncoder 继承的子类时，如果类还定义了__init__() ，那么务必要小心。要处理好所有关键字参数，则应该按如下方式定义：

```
class MyJSONDecoder(JSONDecoder):
```

```
    def __init__(self, **kwargs):
        # 获取自己的参数
        foo = kwargs.pop('foo',None)
        bar = kwargs.pop('bar',None)
        # 使用剩下的内容初始化父类
        JSONDecoder.__init__(self,**kwargs)
```

24.9 mimetypes

mimetypes 模块用于根据文件扩展名猜测与文件关联的MIME类型。它还将MIME类型转换为标准的文件扩展名。MIME类型由类型/子类型对组成，如'text/html'、'image/png' 或'audio/mpeg' 。

```
guess_type(filename
 [, strict
])
```

根据文件名或URL猜测文件的MIME类型。返回一个元组(*type*, *encoding*)，其中 *type* 是"type/subtype" 形式的字符串， *encoding* 是用来编码传输数据的程序（如compress 或gzip ）。如果无法猜测类型，则返回(None, None) 。如果 *strict* 为 *True* （默认值），则只能识别向IANA注册的MIME类型（参见 http://www.iana.org/assignments/media-types ）。否则，一些常见但非官方的MIME类型将无法被识别。

```
guess_extension(type
 [, strict
])
```

根据文件的MIME类型猜测其标准文件扩展名。返回带有文件扩展名的字符串，包括前导点号（.）。未知类型返回None 。如果 *strict* 为True （默认值），那么只能识别官方MIME类型。

```
guess_all_extensions(type [, strict
])
```

与guess_extension() 类似，但返回的是由所有可能的文件扩展名组成的列表。

```
init([files

])
```

初始化该模块。 *files* 是一个文件名序列，用于提取类型信息。这些文件由多行数据组成，每一行都将一个MIME类型映射到一个可接受文件后缀的列表，如下：

```
image/jpeg:   jpe jpeg jpg
text/html:    htm html
...

read_mime_types(filename

)
```

从给定文件名加载类型映射。返回将文件扩展名映射到MIME类型字符串的字典。如果 *filename* 不存在或者不可读，则返回None 。

```
add_type(type

, ext

 [, strict

])
```

向映射添加新的MIME类型。type 是MIME类型，如'text/plain' ； *ext* 是文件扩展名，如'.txt' ； *strict* 是布尔值，表示是否为官方注册的MIME类型。默认情况下， *strict* 为True 。

24.10 quopri

quopri 模块对字节字符串执行Quoted Printable传输编码和解码。该格式主要用于编

码最可能作为ASCII读取、但是包含少量不可打印或特殊字符的8位文本文件（例如，128～255范围内的控制字符或非ASCII字符）。以下规则描述Quoted Printable编码的工作方式。

- 任何可打印的非空白ASCII字符（'=' 除外）都按原样表示。
- '=' 字符用作转义字符。如果后跟两个十六进制数字，则表示具有该值的字符（如'=0c'）。等号使用'=3D' 表示。如果'=' 出现在一行末尾，则表示软换行。只有需要将很长的输入文本分割为多个输出行时才会出现这种情况。
- 空格和制表符都按原样表示，但是不能出现在行末尾。

当文档使用扩展ASCII字符集中的特殊字符时经常可以看到这种格式。例如，假设文档包含文本“Copyright © 2009”，使用Python字节字符串表示则是b'Copyright \xa9 2009'，使用Quoted Printable字符串表示则是b'Copyright=A9 2009'，其中使用转义序列'=A9' 替换了其中的特殊字符'\xa9'。

```
decode(input
, output
 [, header
])
```

将字节解码为quopri 格式。 *input* 和 *output* 是以二进制模式打开的文件对象。如果 *header* 为True，那么下划线（_）将被解释为空格。否则，它将保留原样。该参数在解码已经编码的MIME报头 时使用。默认情况下， *header* 为False 。

```
decodestring(s
 [, header
])
```

解码字符串 *s* 。 *s* 可以是Unicode或字节字符串，但是结果始终是字节字符串。*header* 的含义与decode() 中相同。

```
encode(input
, output
, quotetabs
```

```
 [, header

])
```

将字节编码为quopri格式。 *input* 和 *output* 是以二进制模式打开的文件对象。 *quotetabs* 如果设置为True，则除了遵守一般的引用规则外，还会强制引用制表符。否则制表符保留原样。默认情况下， *quotetabs* 为False。 *header* 的含义与decode()中的含义相同。

```
encodestring(s

 [, quotetabs

 [, header

]])
```

编码字节字符串 *s* 。结果也是一个字节字符串。 *quotetabs* 和 *header* 的含义与decode()中的相同。

> 注意
>
> Quoted Printable数据编码早于Unicode，只适用于8位数据。尽管它最常用于文本，但它实际上只适用于以单字节表示的ASCII字符和ASCII扩展字符。使用该模块时，要确保所有文件都是二进制模式，并且处理的是字节字符串。

24.11 xml包

Python包含各种处理XML数据的模块。关于XML处理的主题涉及很广，本书不准备详细介绍相关内容。本节假定读者已经熟悉了某些基本的XML概念。Steve Holzner所著的*Inside XML* ① 、Elliotte Harold和W. Scott Means合著的*XML in a Nutshell* ② 等书籍都介绍了很多有用的XML基本概念。讨论使用Python处理XML的书籍也有很多，包括Christopher Jones的“*Python & XML* ”（O’Reilly and Associates）以及Sean McGrath的“*XML Processing with Python* ”（Prentice Hall）。

Python提供了两种XML支持。第一种是支持两种XML解析的行业标准方法——SAX和DOM。SAX（Simple API for XML，用于XML的简单API）以事件处理为基础，按照遇到XML元素的顺序读取XML文档，触发处理函数来执行处理。DOM（Document Object Model，文档对象模型）构建表示整个XML文档的树结构。树结构构建完成后，DOM将提供一个遍历树和提取数据的接口。SAX和DOM API都不源自Python，Python只是复制了为Java和JavaScript开发的接口。

第二种是支持特定于Python的XML解析方法。尽管可以使用SAX和DOM接口处理XML，但是标准库中最方便的编程接口还是ElementTree接口。这种方法充分利用了Python的语言特点，大部分用户认为它比SAX或DOM要更加简单快捷。本节后面部分将讨论这3种XML解析方法，其中重点讨论ElementTree方法。

这里需要提醒读者，本节内容只涉及基本的XML数据解析。Python中还有可以实现新解析器、从头构建XML文档的XML模块。此外，各种第三方扩展添加了其他XML功能，如支持XSLT和XPATH，从而扩展了Python的功能。更多参考信息链接详见 http://wiki.python.org/moin/PythonXml 。

24.11.1 XML示例文档

下例展示了一个典型的XML文档，是一道菜的菜谱。

```
<?xml version="1.0" encoding="iso-8859-1"?>
<recipe>
   <title>
   Famous Guacamole
   </title>
   <description>
   A southwest favorite!
   </description>
   <ingredients>
        <item num="4"> Large avocados, chopped </item>
        <item num="1"> Tomato, chopped </item>
        <item num="1/2" units="C"> White onion, chopped </item>
        <item num="2" units="tbl"> Fresh squeezed lemon juice </item>
        <item num="1"> Jalapeno pepper, diced </item>
        <item num="1" units="tbl"> Fresh cilantro, minced </item>
        <item num="1" units="tbl"> Garlic, minced </item>
        <item num="3" units="tsp"> Salt </item>
        <item num="12" units="bottles"> Ice-cold beer </item>
   </ingredients>
   <directions>
   Combine all ingredients and hand whisk to desired consistency.
   Serve and enjoy with ice-cold beers.
   </directions>
</recipe>
```

该文档由具有开始标记和结束标记（如`<title>...</title>`）的元素组成。元素通常是嵌入式的，并且分层次组织起来，如出现在`<ingredients>`下方的`<item>`。在每个文档中，只有一个元素是文档根。在本例中，文档根是`<receipe>`元素。元素可以有多个属性，如item元素`<item num="4"> Large avocados, chopped</item>`。

处理XML文档通常会涉及所有这些基本特性。例如，你可能想从特定元素类型中提取文本和属性。要查找元素位置，必须从根元素开始导航遍历整个文档层次结构。

24.11.2 xml.dom.minidom

xml.dom.minidom模块支持较基础的XML文档解析，并根据DOM约定在内存中以树结构形式存储该文档。它包含两个解析函数：

```
parse(file
 [, parser
])
```

解析 *file* 的内容并返回表示文档树顶部的节点。 *file* 是文件名或者已经打开的文件对象。parser 是可选的SAX2兼容解析对象，用来构造树结构。如果忽略该参数，则使用默认解析器。

```
parseString(string
 [, parser
])
```

与parse() 类似，不同之处在于它通过字符串而不是文件提供输入数据。

1. 节点

解析函数返回的文档树由节点集合链接而成。每个节点n 都具备以下属性，可以用来提取信息和导航遍历树结构。

节点属性	描　　述
n .attributes	保存属性值的映射对象（如果有）
n .childNodes	*n* 的所有子节点组成的列表
n .firstChild	节点 *n* 的第一个子节点
n .lastChild	节点 *n* 的最后一个子节点
n .localName	元素的本地标记名称。如果标记中出现冒号（如'<foo:bar ...>'），那么该属性只包含冒号后面的部分
n .namespaceURI	与 *n* 关联的命名空间（如果有）

n .nextSibling	树结构中出现在 *n* 后面且具有相同父节点的节点。如果 *n* 是最后一个节点，则为None
n .nodeName	节点的名称。含义取决于节点的类型
n .nodeType	描述节点类型的整数。它可以设置为以下值（均为Node类的类变量）： ATTRIBUTE_NODE 、 CDATA_SECTION_NODE 、 COMMENT_NODE 、 DOCUMENT_FRAGMENT_NODE 、 DOCUMENT_NODE 、 DOCUMENT_TYPE_NODE 、 ELEMENT_NODE 、 ENTITY_NODE 、 ENTITY_REFERENCE_NODE 、 NOTATION_NODE 、 PROCESSING_INSTRUCTION_NODE 、 TEXT_NODE
n .nodeValue	节点的值。含义取决于节点类型
n .parentNode	对父节点的引用
n .prefix	出现在冒号前面的标记名称部分。例如，元素'<foo:bar ...>' 应该具有前缀'foo'
n .previousSibling	树结构中出现在*n* 前面且与其具有相同父节点的节点

除了这些属性之外，所有节点都具有以下方法。这些方法通常用来操作树结构。

```
n
.appendChild(child
)
```

向 *n* 添加新的子节点 *child* 。新的子节点可以添加到任何子节点后面。

```
n
.cloneNode(deep
)
```

复制节点 *n* 。如果 *deep* 为True ，则还可以复制所有子节点。

```
n
.hasAttributes()
```

如果节点有任何属性，则返回True 。

```
n
.hasChildNodes()
```

如果节点有任何子节点，则返回True 。

```
n
.insertBefore(newchild
, ichild
)
```

在另一个子节点 *ichild* 之前插入新的子节点 *newchild* 。 *ichild* 必须是 *n* 的子节点。

```
n
.isSameNode(other
)
```

如果节点 *other* 引用的DOM节点与 *n* 相同，则返回True 。

```
n
.normalize()
```

将相邻的文本节点结合为一个单独的文本节点。

```
n
.removeChild(child
)
```

从 *n* 删除子节点 *child* 。

```
n
.replaceChild(newchild
,oldchild
)
```

使用 *newchild* 替换子节点 *oldchild* 。 *oldchild* 必须是 *n* 的子节点。

尽管树结构中可能出现许多种节点，但最常见的还是Document 、Element 和Text 节点。下面将对此逐一介绍。

2．Document 节点

Document 节点 *d* 出现在整个文档树的顶部，代表作为一个整体的文档。它具有以下方法和属性。

```
d
```

```
.documentElement
```

包含整个文档的根元素。

```
d
.getElementsByTagName(tagname
)
```

搜索所有子节点，并返回带有给定标记名称 *tagname* 的元素组成的列表。

```
d
.getElementsByTagNameNS(namespaceuri
, localname
)
```

搜索所有子节点，并返回带有给定标记命名空间URI和本地名的元素组成的列表。返回列表是NodeList 类型的对象。

3. Element 节点

Element 节点表示一个XML元素，如'<foo>...</foo>' 。要从元素中获取文本，需要查找其Text 子节点。定义以下属性和方法来获取其他信息。

```
e
.tagName
```

元素的标记名称。例如，如果使用'<foo ...>'定义元素，则标记名称是'foo'。

```
e
.getElementsByTagName(tagname
)
```

返回带有给定标记名称的所有子元素组成的列表。

```
e
.getElementsByTagNameNS(namespaceuri
, localname
)
```

返回命名空间中给定标记名称的所有子节点组成的列表。 *namespaceuri* 和 *localname* 是指定命名空间和标记名称的字符串。如果使用'<foo xmlns:foo="http://www.spam.com/foo ">'之类的声明来声明命名空间，那么 *namespaceuri* 将设置为'http://www.spam.com/foo '。如果搜索之后的元素'<foo:bar>'，那么 *localname* 应设置为'bar'。返回的对象类型为NodeList。

```
e
.hasAttribute(name
)
```

如果元素具有名为 *name* 的属性，则返回True。

```
e
```

```
.hasAttributeNS(namespaceuri
, localname
)
```

如果元素具有 *namespaceuri* 和 *localname* 指定的属性，则返回True 。参数的含义与get ElementsByTagNameNS() 中的描述相同。

```
e
.getAttribute(name
)
```

返回属性 *name* 的值。返回值是一个字符串。如果不存在这样的属性，则返回一个空字符串。

```
e
.getAttributeNS(namespaceuri
, localname
)
```

返回 *namespaceuri* 和 *localname* 指定的属性的值。返回值是一个字符串。如果不存在这样的属性，则返回一个空字符串。参数的含义与getElementsByTagNameNS() 中的描述相同。

4. **Text** 节点

Text 节点用于表示文本数据。文本数据存储在Text 对象 *t* 的 *t* .data 属性中。与给定文档元素关联的文本总是存储在元素子节点的Text 节点中。

5. 实用函数

Python对节点定义了以下实用方法。它们不是DOM标准的一部分，而是为了方便和

调试而提供的。

```
n
.toprettyxml([indent
 [, newl
]])
```

创建格式良好的字符串，该字符串包含节点n 及其子节点表示的XML。 *indent* 指定缩进字符串，默认为制表符（'\t'）。 *newl* 指定换行符，默认为'\n'。

```
n
.toxml([encoding
])
```

创建一个字符串，该字符串包含节点 *n* 及其子节点表示的XML。 *encoding* 指定编码（如'utf-8'）。如果没有指定任何编码，则在输出文本中也不指定。

```
n
.writexml(writer
 [, indent
 [, addindent
 [, newl
]])
```

将XML写入 *writer* 。 *writer* 可以是提供兼容文件接口的write() 方法的任何对象。 *indent* 指定 *n* 的缩进。它是一个字符串，添加到输出中n 节点的前面。 *addindent* 是一个字符串，指定应用于子节点 *n* 的增量缩进。 *newl* 指定换行符。

6. DOM 示例

以下示例展示了如何使用xml.dom.minidom 模块从XML文件中解析和提取信息：

```
from xml.dom import minidom
doc = minidom.parse("recipe.xml")

ingredients  = doc.getElementsByTagName("ingredients")[0]
items        = ingredients.getElementsByTagName("item")

for item in items:
    num      = item.getAttribute("num")
    units    = item.getAttribute("units")
    text     = item.firstChild.data.strip()
    quantity = "%s %s" % (num,units)
    print("%-10s %s" % (quantity,text))
```

> 注意
>
> xml.dom.minidom 模块有许多更改解析树和处理各种XML节点类型的功能。请参考在线文档了解更多信息。

24.11.3 xml.etree.ElementTree

xml.etree.ElementTree 模块定义了一个灵活的容器对象ElementTree，用于存储和操作分层数据。尽管该对象通常与XML处理结合使用，但其实它的用途非常广泛——它是能够横跨列表和字典的角色。

1. ElementTree 对象

以下类用来定义新的ElementTree 对象并表示层次结构的最上层。

```
ElementTree([element

 [, file

]])
```

新建ElementTree 对象。 *element* 是表示树结构根节点的实例。该实例支持下面描述的元素接口。file 要么是文件名，要么是类文件对象，将从中读取XML数据来组成树结构。

ElementTree 的实例 *tree* 具有以下方法。

```
tree._setroot(element

)
```

将根元素设为 *element* 。

```
tree.find(path

)
```

在树结构中查找并返回类型匹配给定路径的最上层元素。 *path* 是一个字符串，描述元素类型及其相对于其他元素的位置。下面描述了路径语法。

路　　径	描　　述
'*tag* '	只匹配带有给定标记（如'<*tag* >...</*tag* >' ）的最上层元素。不匹配低层次上定义的元素。不匹配嵌入另一个元素的元素类型 *tag* （如'<foo><*tag* >...</*tag* ></foo>' ）
'*parent/tag* '	如果标记'tag' 的元素子节点有标记'parent' ，则匹配该元素。可以根据需要指定任何数量的路径名称组件
'*'	选择所有子元素。例如，'*/tag' 应该匹配所有标记名为'tag' 的子元素
'.'	从当前节点开始搜索
'//'	选择元素下方所有层次的子元素。例如，'.//*tag* ' 表示匹配所有下层标记为'tag' 的元素

如果处理涉及XML命名空间的文档，那么路径中的 *tag* 字符串应该为'{uri}tag' 形式，其中 *uri* 是一个字符串，如'http://www.w3.org/TR/html4/ ' 。

```
tree

.findall(path

)
```

查找树结构中匹配给定路径的所有顶层元素，并按照文档顺序返回一个列表或者迭代器。

```
tree
.findtext(path
 [, default
])
```

返回树结构中匹配给定路径的最顶层元素的元素文本。 *default* 是在没有找到匹配元素时返回的字符串。

```
tree
.getiterator([tag
])
```

创建一个迭代器，按照小节顺序生成树结构中标签匹配 *tag* 的所有元素。如果忽略 *tag* ，则按顺序返回树结构中的所有元素。

```
tree
.getroot()
```

返回树结构的根元素。

```
tree
.parse(source
 [, parser
```

```
])
```

解析外部XML数据并使用结果替换根元素。 *source* 是文件名或者表示XML数据的类文件对象。 *parser* 是可选的TreeBuilder 实例，这将在后文介绍。

```
tree
.write(file
 [, encoding
])
```

将树结构的整个内容写入文件。 *file* 是为写入而打开的文件名或者类文件对象。 *encoding* 是要使用的输出编码，如果没有指定，则默认为解释器的默认编码（大部分情况下是'utf-8' 或'ascii' ）。

2. 创建元素

保存在ElementTree 中的元素类型可以使用各种类型的实例表示，这可以是通过解析文件在内部创建的，也可以是使用以下构造函数创建的。

```
Comment([text
])
```

新建注释元素。 *text* 是包含元素文本的字符串或字节字符串。该元素在解析或写入输出时映射到XML注释。

```
Element(tag
 [, attrib
 [, **extra
]])
```

新建元素。 *tag* 是元素的名称。例如，如果创建元素'<foo>....</foo>'，那么 *tag* 为'foo'。 *attrib* 是以字符串或字节字符串形式指定的元素属性字典。 *extra* 中提供的其他关键字参数也都可以用来设置元素属性。

```
fromstring(text

)
```

根据 *text* 中XML文本的片段创建元素，与下面描述的XML() 相同。

```
ProcessingInstruction(target

 [, text

])
```

新建对应于处理指令的元素。 *target* 和 *text* 都是字符串或字节字符串。当映射到XML时，该元素对应于'<?targettext?>'。

```
SubElement(parent

, tag

 [, attrib

 [, **extra

]])
```

与Element() 类似，但是它自动将新元素添加为 *parent* 的子元素。

```
XML(text

)
```

通过解析 *text* 中的XML节点片段创建元素。例如，如果将text 设置为'<foo>....</foo>'，那么该方法将使用标记'foo' 创建标准元素。

```
XMLID(text
)
```

与XML(*text*) 类似，不同之处在于此函数集中了'id' 属性并用来构建一个将ID值映射到元素的字典。返回一个元组(*elem, idmap*)，其中elem 是新元素，idmap 是ID映射字典。例如，XMLID('<foo id="123"><bar id="456">Hello</bar></foo>') 返回(<Element foo>, {'123': <Element foo>, '456': <Element bar>}) 。

3. 元素接口

尽管ElementTree 中存储的元素类型可能不同，但是它们都支持相同的接口。如果 *elem* 是某个元素，那么将定义以下Python运算符。

运 算 符	描 述
elem [*n*]	返回 *elem* 的第 *n* 个子元素
elem [*n*] = *newelem*	将 *elem* 的第 *n* 个子元素更改为不同的元素 *newelem*
del *elem* [*n*]	删除 *elem* 的第 *n* 个子元素
len(*elem*)	*elem* 的子元素数量

所有元素都具有以下基本数据属性。

属 性	描 述
elem. tag	标识元素类型的字符串。例如，<foo>...</foo> 的标记为'foo'
elem.	

text	与元素关联的数据。一般是包含XML元素起始标记和结束标记之间文本的字符串
elem.tail	与属性一起存储的其他数据。在XML中，这通常是一个字符串，而且该字符串包含元素结束标记之后但在下一个起始标记之前的空白
elem.attrib	包含元素属性的字典

元素支持以下方法，有些模拟的是字典上的方法。

```
elem
.append(subelement
)
```

将元素 *subelement* 添加到子元素列表。

```
elem
.clear()
```

清除元素中的所有数据，包括属性、文本和子元素。

```
elem
.find(path
)
```

查找类型匹配 *path* 的第一个子元素。

```
elem
.findall(path
)
```

查找类型匹配 *path* 的所有子元素。按照匹配元素的文档顺序返回列表或迭代。

```
elem
.findtext(path
 [, default
])
```

查找类型匹配 *path* 的第一个子元素的文本。 *default* 是一个字符串，提供没有匹配时要返回的值。

```
elem
.get(key
 [, default
])
```

获取属性key 的值。 *default* 是不存在属性时要返回的默认值。如果涉及XML命名空间，那么 *key* 将是形式为'{uri}key}' 的字符串，其中uri 是'http://www.w3.org/TR/html4/ ' 之类的字符串。

```
elem
.getchildren()
```

按文档顺序返回所有子元素。

```
elem
.getiterator([tag
])
```

返回一个迭代器，生成类型匹配 *tag* 的所有子元素。

```
elem
.insert(index
, subelement
)
```

在子元素列表的 *index* 位置处插入子元素。

```
elem
.items()
```

将所有元素属性以(*name, value*)对列表的形式返回。

```
elem
.keys()
```

返回所有属性名称组成的列表。

```
elem
.remove(subelement
)
```

从子元素列表中删除元素 *subelement* 。

```
elem
.set(key
, value
)
```

将属性 *key* 设为值value 。

4. 树的构建

使用以下对象可以轻松地根据其他树类结构创建ElementTree 对象。

```
TreeBuilder([element_factory
])
```

该类使用一系列start() 、end() 和data() 调用（在解析文件或者遍历另一个树结构时会触发这些调用）构建ElementTree 结构。 *element_factory* 是一个操作函数，将调用该函数新建元素实例。

TreeBuilder 的实例t 具有以下方法。

```
t
.close()
```

关闭树构建函数并返回创建的顶层ElementTree 对象。

```
t
.data(data
)
```

将文本数据添加到正在处理的当前元素。

```
t
.end(tag
)
```

关闭正在处理的当前元素并返回最后一个元素对象。

```
t
.start(tag
, attrs
)
```

新建元素。 *tag* 是元素名称， *attrs* 是带有属性值的字典。

5. 实用函数

Element Tree 定义了以下实用函数。

```
dump(elem
)
```

将elem 的元素结构转储到sys.stdout 以便调试。输出通常是XML。

```
iselement(elem
)
```

检查 *elem* 是否是有效的元素对象。

```
iterparse(source
 [, events
])
```

递增式解析 *source* 中的XML。 *source* 是引用XML数据的文件名或类文件对象。 *events* 是要生成的事件类型组成的列表。可能的事件类型有'start' 、'end' 、'start-ns' 和'end-ns' 。如果忽略该参数，那么只能生成'end' 事件。该函数返回的值是一个迭代器，生成元组(*event, elem*)，其中event 是'start' 或'end' 之类的字符串， *elem* 是正在处理的元素。对于'start' 事件，该元素是新创建的，并且初始为空（但有属性）。对于'end' 事件，该元素是完整的，包含所有子元素。

```
parse(source
)
```

将XML源完整地解析为一个ElementTree 对象。 *source* 是带有XML数据的文件名或类文件对象。

```
tostring(elem

)
```

创建表示 *elem* 及其所有子元素的XML字符串。

6. XML示例

下例使用ElementTree 解析示例菜谱文件并打印原料列表。类似于DOM中展示的示例。

```
from xml.etree.ElementTree import ElementTree

doc = ElementTree(file="recipe.xml")
ingredients = doc.find('ingredients')

for item in ingredients.findall('item'):
     num   = item.get('num')
     units = item.get('units','')
     text  = item.text.strip()
     quantity = "%s %s" % (num, units)
     print("%-10s %s" % (quantity, text))
```

ElementTree 的路径语法可以让我们更加轻松地简化某些任务，并根据需要使用快捷键。例如，下面是以上代码的另一种版本，它使用路径语法提取所有<item>... </item> 元素。

```
from xml.etree.ElementTree import ElementTree

doc = ElementTree(file="recipe.xml")
for item in doc.findall(".//item"):
     num   = item.get('num')
     units = item.get('units','')
     text  = item.text.strip()
     quantity = "%s %s" % (num, units)
     print("%-10s %s" % (quantity, text))
```

考虑利用命名空间的XML文件'recipens.xml'：

```
<?xml version="1.0" encoding="iso-8859-1"?>
<recipe xmlns:r="http://www.dabeaz.com/namespaces/recipe

">
   <r:title>
```

```
   Famous Guacamole
   </r:title>
   <r:description>
   A southwest favorite!
   </r:description>
   <r:ingredients>
        <r:item num="4"> Large avocados, chopped </r:item>
        ...
   </r:ingredients>
   <r:directions>
   Combine all ingredients and hand whisk to desired consistency.
   Serve and enjoy with ice-cold beers.
   </r:directions>
</recipe>
```

要处理命名空间，使用字典将命名空间前缀映射到相关命名空间URI通常可以简化操作。然后便可以使用字符串格式化运算符填充URI，如下所示：

```
from xml.etree.ElementTree import ElementTree
doc = ElementTree(file="recipens.xml")
ns = {
    'r' : 'http://www.dabeaz.com/namespaces/recipe

'
}
ingredients = doc.find('{%(r)s}ingredients' % ns)
for item in ingredients.findall('{%(r)s}item' % ns):
    num   = item.get('num')
    units = item.get('units','')
    text  = item.text.strip()
    quantity = "%s %s" % (num, units)
    print("%-10s %s" % (quantity, text))
```

对于小型XML文件，可以使用`ElementTree`模块快速将其加载到内存以便处理。但是，假如你处理的是一个很大的XML文件，结构如下：

```
<?xml version="1.0" encoding="utf-8"?>
<music>
   <album>
        <title>A Texas Funeral</title>
        <artist>Jon Wayne</artist>
        ...
   </album>
   <album>
        <title>Metaphysical Graffiti</title>
        <artist>The Dead Milkmen</artist>
        ...
   </album>
   .. . continues for 100000 more albums ...
</music>
```

将超大的XML文件读取到内存可能会占用太多内存。例如，读取10MB XML文件可能生成100MB以上的内存数据结构。如果要从此类文件中提取信息，最简单的方法是使用`ElementTree.iterparse()`函数。下面是迭代处理上一个文件中`<album>`节点的例子：

```
from xml.etree.ElementTree import iterparse

iparse = iterparse("music.xml", ['start','end'])
# 查找顶层音乐元素
for event, elem in iparse:
    if event == 'start' and elem.tag == 'music':
         musicNode = elem
         break

# 获取所有专辑
albums = (elem for event, elem in iparse
              if event == 'end' and elem.tag == 'album')

for album in albums:
    # 进行某些处理
    ...
    musicNode.remove(album)             # 完成后删除专辑
```

有效使用`iterparse()`的关键在于丢弃不再使用的数据。最后一条语句`musicNode.remove (album)`就是在完成处理后丢弃`<album>`元素（将其从父元素中移除）。如果监控上一个程序的内存占用，会发现即使在输入文件很大的情况下它的占用率也很低。

7．注意

- 在处理简单XML文档方面，`ElementTree`模块是目前为止Python中最容易、最灵活的方式。但是，它提供的功能有限。例如，它不支持验证，也没有很好的办法处理复杂的XML文档问题（如DTD）。这些功能需要安装第三方包。`lxml.etree`（见 http://codespeak.net/lxml/ ）就是这样一个包，它为常见的libxml2和libxslt提供ElementTree API，并且完全支持XPATH、XSLT和其他功能。
- `ElementTree`模块本身是Fredrik Lundh维护的一个第三方包，参见 http://effbot.org/zone/element-index.htm 。在该站点可以找到比标准库中更新的版本，新版本提供了更多的功能。

24.11.4 `xml.sax`

`xml.sax`模块支持使用SAX2 API解析XML文档。

```
parse(file

, handler

 [, error_handler

])
```

解析XML文档 *file* 。 *file* 是文件名或者打开的文件对象。 *handler* 是内容处理程序对象。error*_handler* 是可选的SAX错误处理程序对象，在线文档中对其有更加详细的描述。

```
parseString(string
, handler
 [, error_handler
])
```

与parse() 类似，但是解析的是字符串中包含的XML数据。

1．处理程序对象

要执行处理，必须为parse() 或parseString() 函数提供一个内容处理程序对象。可以定义从ContentHandler 继承的类，从而创建处理程序。ContentHandler 的实例c具有以下方法，所有方法都可以根据需要在自定义的处理程序类中重写。

```
c
.characters(content
)
```

解析器调用该方法以提供原始字符数据。 *content* 是包含字符的字符串。

```
c
.endDocument()
```

解析器在到达文档末尾时调用该方法。

```
c
.endElement(name
)
```

在到达元素 *name* 的末尾时调用该方法。例如，如果解析'</foo>'，那么调用该方法时 *name* 将设置为'foo'。

```
c
.endElementNS(name
, qname
)
```

到达包含XML命名空间的元素末尾时调用。name 是字符串元组(*uri, localname*)， *qname* 是完全限定名。通常 *qname* 为None，除非启用了SAX namespace-prefixes 功能。例如，如果将元素定义为'<foo:bar xmlns:foo="http://spam.com">'，那么 *name* 元组是(u'http://spam.com ', u' bar')。

```
c
.endPrefixMapping(prefix
)
```

在到达XML命名空间的结尾时调用该方法。 *prefix* 是命名空间的名称。

```
c
```

```
.ignorableWhitespace(whitespace
)
```

在文档中遇到可忽略的空白时调用该方法。 *whitespace* 是包含空白的字符串。

```
c
.processingInstruction(target
, data
)
```

遇到封装在<? ... ?> 内的XML处理指令时调用该方法。 *target* 是指令的类型，*data* 是指令数据。例如，如果指令为'<?xml-stylesheet href="mystyle.css" type="text/css"?>，则 *target* 设置为'xml-stylesheet'，data 是指令文本的其余部分'href="mystyle.css" type="text/css"' 。

```
c
.setDocumentLocator(locator
)
```

解析器调用该方法提供可用来跟踪行号、列和其他信息的定位器对象。该方法的主要目的是在某处存储定位器以便稍后使用。例如，在需要打印错误消息时使用。 *Locator* 中提供的定位器有4个方法：getColumnNumber()、getLineNumber()、getPublicId() 和getSystemId()，这些方法都可以用来获取位置信息。

```
c
.skippedEntity(name
)
```

解析器跳过实体时调用该方法。 *name* 是跳过的实体的名称。

```
c
.startDocument()
```

在文档起始处调用该方法。

```
c
.startElement(name
, attrs
)
```

遇到新的XML元素时调用该方法。 *name* 是元素的名称， *attrs* 是包含属性信息的对象。例如，如果XML元素是'<foo bar="whatever" spam="yes">'，则name 设置为'foo'， *attrs* 包含有关bar 和spam 属性的信息。 *attrs* 对象提供了许多获取属性信息的方法。

方　法	描　述	方　法	描　述
attrs .getLength()	返回属性的数量	*attrs*. getType(*name*)	获取属性 *name* 的类型
attrs .getNames()	返回属性名称列表	*attrs*. getValue(*name*)	获取属性 *name* 的值

```
c
.startElementNS(name
, qname
```

```
, attrs
)
```

遇到新的XML且将要使用XML命名空间时调用该方法。 *name* 是一个元组(*uri, localname*)，其中qname是完全限定元素名称（通常设为None，除非启用了SAX2 namespace-prefixes功能）。 *attrs* 是包含属性信息的对象。例如，如果XML元素为'<foo:bar xmlns:foo="http://spam. com "blah="whatever">'，那么name为(u'http://spam.com ', u'bar')， *qname* 为None， *attrs* 包含有关属性blah的信息。 *attrs* 对象访问属性时使用的方法与上文提到的startElement()方法相同。此外，还添加了以下方法来处理命名空间。

方　　法	描　　述
attrs .getValueByQName(*qname*)	返回限定名称的值
attrs .getNameByQName(*qname*)	返回名称的(*namespace, localname*)元组
attrs .getQNameByName(*name*)	返回元组(*namespace, localname*)指定的 *name* 的限定名称
attrs .getQNames()	返回所有属性的限定名称

```
c
.startPrefixMapping(prefix
, uri
)
```

在开始XML命名空间声明时调用该方法。例如，如果元素定义为'<foo:bar xmlns:foo="http: //spam.com ">'，那么 *prefix* 设置为'foo'， *uri* 设置为'http://spam.com '。

2．示例

下例演示了基于SAX的解析器，打印了上文中讨论的菜谱文件中的配料列表。可以对比24.11.2节中的示例。

```
from xml.sax import ContentHandler, parse

class RecipeHandler(ContentHandler):
    def startDocument(self):
        self.initem = False
    def startElement(self,name,attrs):
        if name == 'item':
           self.num = attrs.get('num','1')
           self.units = attrs.get('units','none')
           self.text = []
           self.initem = True
    def endElement(self,name):
        if name == 'item':
            text = "".join(self.text)
            if self.units == 'none': self.units = ""
            unitstr = "%s %s" % (self.num, self.units)
            print("%-10s %s" % (unitstr,text.strip()))
            self.initem = False
    def characters(self,data):
        if self.initem:
            self.text.append(data)

parse("recipe.xml",RecipeHandler())
```

3. 注意

xml.sax 模块还有许多功能，可以处理各种XML数据并创建自定义解析器。例如，可以定义一些处理程序对象来解析DTD数据和其他文档部分。请参见在线文档了解更多信息。

24.11.5 xml.sax.saxutils

xml.sax.saxutils 模块定义了一些经常与SAX解析器一起使用的实用函数和对象，这些函数和对象在其他方面也能发挥一定的作用。

```
escape(data

 [, entities

])
```

该函数将给定字符串 *data* 中的某些字符替换为转义序列。例如，'<' 被替换为'<'。 *entities* 是将字符映射到转义字符的可选字典。例如，将entities 设为 { u'\xf1' :'ñ' } 可以将ñ替换为'ñ'。

```
unescape(data
 [, entities
])
```

取消 *data* 中出现的特殊转义序列的转义。例如，将'<' 替换为'<' 。 *entities* 是将实体映射到非转义字符值的可选字典。 *entities* 是escape() 所用字典的反向字典，如{'ñ':u' \xf1' } 。

```
quoteattr(data
 [, entities
])
```

转义字符串 *data* ，但是还执行其他处理，以便结果值可以用作XML属性值。返回值可以直接作为属性值打印，例如，print "<element attr=%s>" % quoteattr(somevalue) 。 *entities* 是一个可用于escape() 函数的字典。

```
XMLGenerator([out
 [, encoding
]])
```

ContentHandler 对象，将解析的XML数据以XML文档的形式回显到输出流。这将重新创建原来的XML文档。 *out* 是输出文档，默认为sys.stdout 。 *encoding* 是要使用的字符编码，默认为'iso-8859-1' 。如果你需要调试解析代码，找到可用的处理程序，那么该函数会很有用。

---

① 中译版《XML完全探索》已由中国青年出版社出版。——编者注

② 中译版《XML技术手册》已由中国电力出版社出版。——编者注

第25章　其他库模块

本节主要介绍本书没有详细说明但同样属于标准库的模块。在之前的章节中，基本没有对这些模块进行过讨论，因为它们可能太底层、使用局限于某些特定的平台、已经过时，也可能太过复杂，需要一整本书才能介绍完。虽然本书没有对这些模块进行讨论，但是http://docs.python.org/library/modname 的在线文档对每个模块都有详细介绍。所有模块的索引可参见http://docs.python.org/library/modindex.html 。

这里列出的模块为在Python 2和Python 3共有的功能子集。如果你使用的模块未在本章列出，那么很可能该模块已经被官方正式废弃了。有些模块的名称在Python 3中发生了变化。新名称将使用括号标出（如果适用）。

25.1　Python服务

以下模块提供与Python语言和执行Python解释器有关的其他服务。这些模块大部分都与解析和编译Python源代码有关。

模　　块	描　　述	模　　块	描　　述
bdb	访问调试程序框架	pickletools	pickle 开发人员工具
code	解释器基类	pkgutil	包扩展实用工具
codeop	编译Python代码	pprint	对象的Prettyprinter
compileall	对目录中的Python文件进行字节编译	pyclbr	提取类浏览器的信息
copy_reg(copyreg)	注册用于pickle 模块的内置类型	py_compile	将Python源代码编译为字节代码文件
dis	反汇编程序	repr(reprlib)	可替代repr() 函数的实现
distutils	分发Python模块	symbol	用来表示解析树内部节点的常量
fpectl	浮点异常控制	tabnanny	检测不够明确的缩进
imp	访问import 语句的实现	test	回归测试包

keyword	测试字符串是否是Python关键字	token	解析树结构的终点
linecache	从源文件获取行	tokenize	Python源代码的扫描程序
modulefinder	查找脚本使用的模块	user	用户配置文件解析
parser	访问Python源代码的解析树	zipimport	从zip压缩文件导入模块

25.2 字符串处理

以下模块原来用于处理字符串，现在已经过时。

模　块	描　述	模　块	描　述
difflib	计算字符串之间的差异	stringprep	网络字符串准备
fpformat	浮点数字格式化	textwrap	文字环绕

25.3 操作系统模块

这些模块提供更多操作系统服务。这里列出的有些模块已经合并到了第19章中介绍的其他模块中。

模　块	描　述	模　块	描　述
crypt	访问UNIX crypt函数	rlcompleter	GNU readline的补全函数
curses	Curses库接口	resource	资源使用信息
grp	访问组数据库	sched	事件调度程序
pty	伪终端处理	spwd	访问隐藏式密码数据库
pipes	shell管道的接口	stat	支持解释os.stat()的结果
nis	Sun NIS的接口	syslog	UNIX syslog后台程序的接口
platform	访问特定于平台的信息	termios	UNIX TTY控制

pwd	访问密码数据库	tty	终端控制函数
readline	访问GNU readline库		

25.4 网络

以下模块支持较少使用的网络协议。

模　块	描　述	模　块	描　述
imaplib	IMAP协议	smtpd	SMTP服务器
nntplib	NNTP协议	telnetlib	Telnet协议
poplib	POP3协议		

25.5 网络数据处理

以下模块提供第24章中没有涉及的其他网络数据处理支持。

模　块	描　述	模　块	描　述
binhex	BinHex4文件格式支持	netrc	Netrc文件处理
formatter	一般输出格式化	plistlib	Macintosh plist文件处理
mailcap	Mailcap文件处理	uu	UUencode文件支持
mailbox	读取各种邮箱格式	xdrlib	编码和解码Sun XDR数据

25.6 国际化

以下模块用来编写国际化应用程序。

模　块	描　述	模　块	描　述

gettext	多国语言文本处理服务	locale	系统提供的国际化函数

25.7 多媒体服务

以下模块用来处理各种多媒体文件。

模　　块	描　　述	模　　块	描　　述
audioop	操作原始音频数据	colorsys	转换颜色系统
aifc	读取和写入AIFF和AIFC文件	imghdr	确定图片类型
sunau	读取和写入Sun AU文件	sndhdr	确定声音文件类型
wave	读取和写入WAV文件	ossaudiodev	访问兼容OSS的音频设备
chunk	读取IFF块文件		

25.8 其他

以下模块不属于任何上述分类，加上这些模块，整个列表得以补充完整。

模　　块	描　　述	模　　块	描　　述
cmd	面向行的命令解释器	sched	事件调度程序
calendar	日历生成函数	Tkinter (tkinter)	Tcl/Tk的Python接口
shlex	简单的词法分析模块	winsound	在Windows中播放声音

第三部分　扩展与嵌入

本部分内容

- 第26章　扩展与嵌入Python
- 附录　Python 3

第26章　扩展与嵌入Python

Python最强大的特性之一是能够与用C语言编写的软件进行交互。将Python与其他语言的代码相集成有两种常用的策略。首先，可以将使用其他语言编写的函数打包到Python库模块中，然后通过`import`语句使用。这类模块称为扩展模块，因为它们借助非Python语言编写的额外功能扩展了解释器。这也是迄今为止最常见的Python-C集成形式，因为这样可以让Python应用程序访问高性能的编程库。Python-C集成的另一种形式是嵌入，即从C语言中以库的形式访问Python程序和解释器。有些程序员出于某些理由（通常是用作脚本引擎），想要把Python解释器嵌入到现有的C应用程序框架中，便可采用嵌入这种方法。

本章介绍的是Python-C编程接口的基础知识。首先讲述的是构建扩展模块和嵌入Python解释器的C API基础部分。这部分内容并不能充当指南，因此不熟悉该主题的读者应该参考http://docs.python.org/extending 上的“Python解释器嵌入及扩展”（Embedding and Extending the Python Interpreter）文档，以及http://docs.python.org/c-api 上的参考手册（Reference Manual）。接下来讲述的是`ctypes`库模块。该模块十分有用，可以在不编写任何额外的C代码或者使用C编译器的情况下访问C语言库中的函数。

应该注意的是，为了实现高级扩展与嵌入应用程序，大多数程序员往往会转而求助于高级的代码生成器和编程库。例如，SWIG项目（http://www.swig.org ）就是通过解析C头文件的内容来创建Python扩展模块的编译器。
在http://wiki.pyhton.org/moin/IntegratingPythonWithOtherLanguages 上可以找到关于该项目和其他扩展构建工具的参考资料。

26.1　扩展模块

本节概述了为Python创建手写C扩展模块的基本过程。创建扩展模块就是在Python和使用C语言编写的现有功能之间构建一个接口。C语言库通常以头文件开始，如下所示：

```
/* 文件：example.h */
#include <stdio.h>
#include <string.h>
#include <math.h>

typedef struct Point {
  double x;
  double y;
} Point;

/* 计算两个整数x和y的最大公约数 */
extern int     gcd(int x, int y);

/* 将s中的och替换为nch，并返回替换的数量 */
extern int    replace(char *s, char och, char nch);

/* 计算两点之间的距离 */
extern double distance(Point *a, Point *b);
```

```
/* 一个预处理器常量 */
#define MAGIC 0x31337
```

这些函数原型都在单独的文件中实现，例如：

```
/* example.c */
#include "example.h"
/* 计算两个正整数x和y的最大公约数*/
int gcd(int x, int y) {
    int g;
    g = y;
    while (x > 0) {
         g = x;
         x = y % x;
         y = g;
    }
    return g;
}

/* 替换字符串中的一个字符 */
int replace(char *s, char oldch, char newch) {
  int nrep = 0;
  while (s = strchr(s,oldch)) {
    *(s++) = newch;
    nrep++;
  }
  return nrep;
}

/* 两点之间的距离 */
double distance(Point *a, Point *b) {
  double dx,dy;
  dx = a->x - b->x;
  dy = a->y - b->y;
  return sqrt(dx*dx + dy*dy);
}
```

下面这个C语言的main() 程序说明了这些函数的用法：

```
/* main.c */
#include "example.h"
int main() {
  /* 测试gcd()函数 */
  {
    printf("%d\n", gcd(128,72));
    printf("%d\n", gcd(37,42));
  }
  /* 测试replace()函数 */
  {
    char s[] = "Skipping along unaware of the unspeakable peril.";
    int nrep;
    nrep = replace(s,' ','-');
    printf("%d\n", nrep);
    printf("%s\n",s);
  }
```

```
  /* 测试distance()函数*/
  {
    Point a = { 10.0, 15.0 };
    Point b = { 13.0, 11.0 };
    printf("%0.2f\n", distance(&a,&b));
  }
}
```

下面是前面程序的输出：

```
% a.out


8
1
6
Skipping-along-unaware-of-the-unspeakable-peril.
5.00
```

26.1.1 扩展模块原型

构建扩展模块的方法是编写一个单独的C源文件，其中包含一组包装器函数，用于在Python解释器和底层的C代码之间提供接口。下例是一个叫_example 的基本扩展模块：

```
/* pyexample.c */

#include "Python.h"
#include "example.h"

static char py_gcd_doc[] = "Computes the GCD of two integers";
static PyObject *
py_gcd(PyObject *self, PyObject *args) {
  int x,y,r;
  if (!PyArg_ParseTuple(args,"ii:gcd",&x,&y)) {
      return NULL;
  }
  r = gcd(x,y);
  return Py_BuildValue("i",r);
}

static char py_replace_doc[] = "Replaces all characters in a string";
static PyObject *
py_replace(PyObject *self, PyObject *args, PyObject *kwargs) {
  static char *argnames[] = {"s","och","nch",NULL};
  char     *s,*sdup;
  char     och, nch;
  int      nrep;
  PyObject *result;
  if (!PyArg_ParseTupleAndKeywords(args,kwargs, "scc:replace",
                                   argnames, &s, &och, &nch)) {
      return NULL;
  }
  sdup = (char *) malloc(strlen(s)+1);
```

```
  strcpy(sdup,s);
  nrep = replace(sdup,och,nch);
  result = Py_BuildValue("(is)",nrep,sdup);
  free(sdup);
  return result;
}

static char py_distance_doc[] = "Computes the distance between two points";
static PyObject *
py_distance(PyObject *self, PyObject *args) {
  PyErr_SetString(PyExc_NotImplementedError,"distance() not implemented.");
  return NULL;
}

static PyMethodDef _examplemethods[] = {
  {"gcd", py_gcd, METH_VARARGS, py_gcd_doc},
  {"replace", py_replace, METH_VARARGS | METH_KEYWORDS, py_replace_doc},
  {"distance",py_distance,METH_VARARGS, py_distance_doc},
  {NULL, NULL, 0, NULL}
};

#if PY_MAJOR_VERSION < 3
/* Python 2 module initialization */
void init_example(void) {
  PyObject *mod;
  mod = Py_InitModule("_example", _examplemethods);
  PyModule_AddIntMacro(mod,MAGIC);
}
#else
/* Python 3 module initialization */
static struct PyModuleDef _examplemodule = {
   PyModuleDef_HEAD_INIT,
   "_example", /* name of module */
   NULL, /* module documentation, may be NULL */
   -1,
   _examplemethods
};
PyMODINIT_FUNC
PyInit__example(void) {
  PyObject *mod;
  mod = PyModule_Create(&_examplemodule);
  PyModule_AddIntMacro(mod, MAGIC);
  return mod;
}
#endif
```

扩展模块必须包含Python.h 。针对想访问的每个C函数，都需要编写一个包装器函数。这些包装器函数接受两个（self 和args ，类型均为PyObject * ）或三个参数（self 、args 和kwargs ，类型均为PyObject* ）。包装器函数实现要应用于某些对象实例的内置方法时，就要使用self 参数。在这种情况下，该实例放在self 参数中。否则，self 将被设置为NULL 。args 是一个元组，包含由解释器传递的函数参数。kwargs 是包含关键字参数的字典。

使用PyArg_ParseTuple() 或PyArg_ParseTupleAndKeywords() 函数可将参数从Python转换为C语言。类似地，使用Py_BuildValue() 函数可以构造一个可接受的返回

值。稍后的内容中将会介绍这些函数。

用于扩展函数的文档字符串应该放在单独的字符串变量中，如前面例子中的py_gcd_doc 和py_replace_doc 。模块初始化期间需要引用这些变量（后面介绍）。

包装器函数绝不能修改通过引用从解释器接收到的数据，否则将导致严重的后果。这正是py_replace() 包装器在将接收到的字符串传递给C函数（它将就地修改此字符串）之前，为其制作副本的原因。如果遗漏该步骤，包装器函数就可能违反Python的字符串不可变性规定。

如果要引发异常，可以使用前面所示py_distance() 包装器中的PyExc_SetString() 函数。返回NULL 表明出现错误。

方法表_examplemethods 用于将Python名称关联到C包装器函数。这些名称用于从解释器调用函数。METH_VARARGS 标志用于指示包装器的调用约定。在这种情况下，只接受元组形式的位置参数。也可以把它设置为METH_VARARGS | METH_KEYWORDS ，表示包装器函数接受关键字参数。该方法表还将为每个包装器函数设置文档字符串。

扩展模块的最后部分执行初始化过程，该过程在Python 2和Python 3中有所不同。在Python 2中，模块初始化函数init_example 可初始化模块的内容。在这个例子中，Py_InitModule("_example", _examplemethods) 函数创建模块_example ，并使用与方法表中列出的函数相对应的内置函数对象来填充它。在Python 3中，必须创建描述该模块的PyModuleDef 对象_examplemodules 。然后编写函数PyInit__example() 来初始化模块。如果有必要，模块初始化函数也是声明常量和模块其他部分的地方。例如，PyModule_AddIntMacro() 给模块增加了预处理器的值。

需要注意的是，命名对于模块初始化非常重要。如果创建名为 *modname* 的模块，在Python 2中模块初始化函数必须命名为initmodname() ，而在Python 3中必须为PyInit_*modname* () 。如果不这么做，解释器就不能正确加载模块。

26.1.2 命名扩展模块

在C扩展模块的名称前面加上下划线是一种标准做法，如_example 。Python标准库本身也遵循这种约定。例如，名为_socket 、_thread 、_sre 和_fileio 的模块分别对应于socket 、threading 、re 和io 模块等C编程组件。一般不直接使用这些C扩展模块，而是创建高级的Python模块，如：

```
# example.py
from _example import *
# 在下面添加其他支持代码
...
```

这种Python包装器的用途是为模块提供额外的支持代码或者提供高级接口。很多情况下，在Python语言中实现部分扩展模块比在C语言中更容易。这种设计让我们可以轻松地实现扩展模块。如果仔细查看标准库模块，你会发现很多都是以这种方式混合使用C和Python语言来实现的。

26.1.3 编译与打包扩展

编译和打包扩展模块的首选方式是用distutiles 。为此需要创建一个setup.py 文件，如：

```
# setup.py
from distutils.core import setup, Extension

setup(name="example",
      version="1.0",
      py_modules = ['example.py'],
      ext_modules = [
        Extension("_example",
                  ["pyexample.c","example.c"])
        ]
      )
```

在这个文件中，需要包含高级Python文件（example.py ）和构成扩展模块的源文件（pyexample.c 和example.c ）。构建测试模块时，输入以下内容：

```
% python setup.py build_ext --inplace
```

这会将扩展代码编译为一个共享库，并把它放在当前的工作目录中。这个库的名称将会是_examplemodule.so 、so_examplemodule.pyd 或者某种类似的名称。

编译成功之后，使用模块很简单，例如：

```
% python3.0


Python 3.0 (r30:67503, Dec  4  2008, 09:40:15)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import example


>>> example.gcd(78,120)


6
>>> example.replace(

"Hello World

",' ','-')


(1, 'Hello-World')
>>> example.distance()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NotImplementedError: distance() not implemented.
>>>
```

更为复杂的扩展模块可能还需要提供其他的构造信息，如包含目录、库和预处理器宏。它们也可以包含在setup.py 文件中，例如：

```
# setup.py
from distutils.core import setup, Extension

setup(name="example",
      version="1.0",
      py_modules = ['example.py'],
      ext_modules = [
        Extension("_example",
                  ["pyexample.c","example.c"],
                  include_dirs = ["/usr/include/X11","/opt/include"],
                  define_macros = [('DEBUG',1'),
                                   ('MONDO_FLAG',1)],
                  undef_macros = ['HAVE_FOO','HAVE_NOT'],
                  library_dirs= ["/usr/lib/X11", "/opt/lib"],
                  libraries = [ "X11", "Xt", "blah" ])
      ]
     )
```

如果需要安装一般用途的扩展模块，只要输入python setup.py install 即可。第8章详细介绍了该信息。

在某些情况下，可能需要手动构建扩展模块。这需要了解关于各种编译器和链接器选项的高级知识。下面给出了一个在Linux上手动构建模块的例子：

```
linux % gcc -c -fpic -I/usr/local/include/python2.6 example.c pyexample.c


linux % gcc -shared example.o pyexample.o -o _examplemodule.so
```

26.1.4 从Python到C语言的类型转换

扩展模块使用以下函数来转换从Python传递给C语言的参数。它们的原型定义包含在Python.h 头文件中。

```
int PyArg_ParseTuple(PyObject *args

, char *format
```

```
, ...);
```

该函数用于将 *args* 中包含的位置参数元组解析为一系列C语言变量。 *format* 是一个格式字符串，其中包含0个或多个表26-1～表26-3中提供的说明符字符串，它描述 *args* 中应有的内容。所有余下的参数包含可保存结果的C语言变量地址。这些参数的顺序和类型必须与 *format* 中使用的说明符匹配。如果参数无法解析，则返回0。

```
int PyArg_ParseTupleAndKeywords(PyObject *args

, PyObject *kwargs

,

                                char
*format



, char
**kwlist



, ...);
```

该函数用于解析 *kwargs* 中包含的位置参数元组和关键字参数字典。 *format* 的意义与PyArg_ParseTuple() 中相同。唯一的区别在于，kwlist 是包含所有参数名称的、以 null 值结尾的字符串列表。成功返回1，出现错误则返回0。

表26-1列出了可以赋给 *format* 参数以转换数字的格式代码。C参数类型列中列出了应该传递给PyArg_Parse*() 函数的C语言数据类型。对于数字，它是指向结果保存位置的指针。

表**26-1** **PyArg_Parse*** 函数的数字转换与相关的**C**数据类型

格　　式	**Python**类型	**C**参数类型
"b"	整数	signed char *r
"B"	整数	unsigned char *r
"h"	整数	short *r

"H"	整数	unsigned short *r
"i"	整数	int *r
"I"	整数	unsigned int *r
"l"	整数	long int *r
"k"	整数	unsigned long *r
"L"	整数	long long *r
"K"	整数	unsigned long long *r
"n"	整数	Py_ssize_t *r
"f"	浮点数	float *r
"d"	浮点数	double *r
"D"	复数	Py_complex *r

转换有符号整数值时，如果Python整数对于相应的C数据类型而言过大，就会引发OverflowError异常。但接受无符号值（如I、H、K等）的转换不会检查溢出问题，如果值超出支持的范围，将在后台对它进行截取。在浮点数转换中，可以提供Python的int或float类型的值作为输入，在这种情况下，整数将被转换成浮点数。如果用户定义的类提供正确的转换方法，如__int__()或__float__()，就可以把它们当作数字接受。例如，前面提到的任意一种整数转换都可以接受实现了__int__()方法的用户自定义类作为输入，而且转换时将自动调用__int__()方法。

表26-2列出了适用于字符串和字节的转换类型。很多字符串转换都返回一个指针和长度作为结果。

表**26-2** **PyArg_Parse*** 函数的字符串转换与相关的**C**数据类型

格　　式	**Python**类型	**C**参数类型
"c"	长度为1的字符串或字节字符串	char *r

"s"	字符串	char **r
"s#"	字符串、字节或缓存	char ** r, int *len
"s*"	字符串、字节或缓存	Py_buffer *r
"z"	字符串或None	char **r
"z#"	字符串、字节或None	char **r, int *len
"z*"	字符串、字节、缓存或None	Py_buffer *r
"y"	字节（null 结尾）	char **r
"y#"	字节	char **r, int *len
"y*"	字节或缓存	Py_buffer *r
"u"	字符串（Unicode）	Py_UNICODE **r
"u#"	字符串（Unicode）	Py_UNICODE **r, int *len
"es"	字符串	const char *enc, char **r
"es#"	字符串或字节	const char *enc, char **r, int *len
"et"	字符串或null 结尾的字节	const char *enc, char **r, int *len
"et#"	字符串或字节	const char *enc, char **r, int *len
"t#"	只读缓存	char **r, int *len
"w"	读写缓存	char **r
"w#"	读写缓存	char **r, int *len
"w*"	读写缓存	Py_buffer *r

字符串处理在C扩展中是一个特殊问题，原因在于char * 数据类型的用途很多。例如，它可能引用的是文本、一个字符或者原始二进制数据的一个缓存。还有一个问题是，如何处理C语言中为了表示文本字符串结束而嵌入的NULL 字符（'\x00'）。

在表26-2中，传递文本时应该使用"s"、"z"、"u"、"es" 和"et" 转换代码。对于这些代码，Python假定输入文本不包含任何嵌入的NULL 字符，否则就会引发TypeError 异常。但可以放心地假定得到的C字符串以NULL 字符结尾。在Python 2中，8位字符串和Unicode字符串均可传递，但在Python 3中，除"et" 之外的所有转换都需要Python的str 类型，不能使用bytes 。将Unicode字符串传递给C语言时，解释器会使用默认的Unicode编码（通常为UTF-8）方式来编码它们。一个例外是"u" 转换代码，它返回的字符串使用的是Python的内部Unicode表示。这是一个Py_UNICODE 值的数组，其中的Unicode字符在C语言中一般使用wchar_t 类型表示。

使用"es" 和"et" 代码可以为文本指定其他编码方式。使用它们时，需要提供编码名称，如'utf-8' 或'iso-8859-1' ，而且文本将被编码到一块缓存中并以该格式返回。"et" 代码与"es" 的区别在于：如果提供Python字节字符串，将假定它已进行了编码并在传递时不加修改。关于"es" 和"et" 转换有一点需要注意，即它们动态地为结果分配内存，并需要用户使用PyMem_Free() 显式地释放它。因此，使用这些转换的代码应该与下面所示的类似：

```
PyObject *py_wrapper(PyObject *self, PyObject *args) {
    char *buffer;
    if (!PyArg_ParseTuple(args,"es","utf-8",&buffer)) {
        return NULL;
    }
    /* 执行操作 */
    ...
    /* 释放内存，并返回结果 */
    PyMem_Free(buffer);
    return result;
}
```

处理文本或二进制数据时需要使用"s#" 、"z#" 、"u#" 、"es#" 或"et#" 代码。这些转换与前面讲到的转换完全相同，但它们还返回一个长度。因此，可以取消嵌入NULL字符的限制。另外，这些转换增加了对字节字符串和支持缓存接口的其他对象的支持。通过缓存接口 这种手段，Python对象可以公开表示其内容的原始二进制缓存。它一般用在字符串、字节和数组上（例如，array 模块中创建的数组支持它）。在这种情况下，如果对象提供可读的缓存接口，就会返回指向该缓存的指针及其大小。最后，如果给"es#" 和"et#" 转换提供非NULL指针和长度，就会假定它们表示一块预先分配好的缓存，可在其中放置编码的结果。在这种情况下，解释器不会为结果分配新的内存，也不必调用PyMem_Free() 方法。

"s*" 和"z*" 转换代码类似于"s#" 和"z#" ，但它们会使用所接收数据的相关信息填充Py_buffer 结构。PEP-3118中介绍了这方面的更多内容，但这种结构最少也有char *buf 、int len 和int itemsize 三个属性，分别代表缓存、缓存长度（单位为字节）和缓存中项的大小。另外，解释器还给缓存上了锁，防止在扩展代码占用它时其他线程对其进行修改。这可让扩展独立处理缓存内容，很可能是在解释器进程之外的进程中进行。所有处理执行完毕之后，由用户决定是否对缓存调用PyBuffer_Release() 方法。

转换代码"t#"、"w"、"w#"和"w*"类似于"s"系列代码，但它们只接受实现了缓存接口的对象。"t#"转换代码要求缓存是可读的。"w"代码要求缓存既是可读的也是可写的。支持可写缓存的Python对象是可变的。因此，C扩展可以合法地改写或修改缓存内容。

"y"、"y#"和"y*"转换代码类似于"s"系列代码，但它们只接受字节字符串。这些代码用于编写只接受字节、不接受Unicode字符串的函数。"y"代码只接受不包含嵌入NULL字符的字节字符串。

表26-3列出的转换代码可以接受任意Python对象作为输入，结果的类型为PyObject*。如果C扩展需要处理比简单的数字或字符串更复杂的Python对象，有时会要用到这些代码—例如想让一个C扩展函数接受Python类的实例或字典。

表**26-3** **PyArg_Parse*** 函数的**Python**对象转换与相关的**C**数据类型

格　　式	**Python**类型	**C**类型
"O"	任意	PyObject **r
"O!"	任意	PyTypeObject *type，PyObject **r
"O&"	任意	int (*converter)(PyObject *, void *)，void *r
"S"	字符串	PyObject **r
"U"	Unicode	PyObject **r

"O"、"S"和"U"说明符返回类型为PyObject *的原始Python对象。"S"和"U"分别限制该对象为字符串或Unicode字符串。

"O!"转换需要两个C参数：一个指向Python类型对象的指针，以及一个指向PyObject *的指针，后者中放置了指向该对象的指针。如果对象的类型与类型对象不匹配，则会引发TypeError。例如：

```
/* Parse a List Argument */
PyObject *listobj;
PyArg_ParseTuple(args,"O!", &PyList_Type, &listobj);
```

下面列出了C类型名称对应于这种转换的一些常用Python容器类型。

C名称	**Python**类型	**C**名称	**Python**类型

PyList_Type	list	PyTuple_Type	tuple
PyDict_Type	dict	PySlice_Type	slice
PySet_Type	set	PyByteArray_Type	bytearray
PyFrozenSet_Type	frozen_set		

"O&" 转换接受两个参数(converter, addr)，并使用函数将PyObject * 转换为C数据类型。converter 是指向原型为int converter(PyObject *obj , void *addr) 的函数的指针，其中 *obj* 是被传递的Python对象，而addr 是作为PyArg_ParseTuple() 函数第二个参数所提供的地址。converter() 执行成功时返回1，否则返回0。出现错误时，转换器也会引发异常。这类转换可用于将Python对象（如列表或元组）映射为C数据结构。例如，下面为我们前面代码中的distance() 包装器给出了一种可能的实现：

```
/* Convert a tuple into a Point structure. */
int convert_point(PyObject *obj, void *addr) {
    Point *p = (Point *) addr;
    return PyArg_ParseTuple(obj,"ii", &p->x, &p->y);
}
PyObject *py_distance(PyObject *self, PyObject *args) {
    Point p1, p2;
    double result;
    if (!PyArg_ParseTuple(args,"O&O&",
                          convert_point, &p1, convert_point, &p2)) {
        return NULL;
    }
    result = distance(&p1,&p2);
    return Py_BuildValue("d",result);
}
```

最后，参数格式字符串可以包含一些与元组拆装、文档、错误消息和默认参数相关的额外修饰符。下面列出了这些修饰符。

格式字符串	描　　述
"(items)"	拆装一个对象元组。items 由格式转换组成
"|"	可选参数的开始
":"	参数的结束。余下文本是函数名称
";"	参数的结束。余下文本是错误消息

"(items)" 用于拆装Python元组中的值。这是一种将元组映射为简单C结构的有用方式。例如，下面给出了py_distance() 包装器函数的另一种可能实现：

```
PyObject *py_distance(PyObject *self, PyObject *args) {
    Point p1, p2;
    double result;
    if (!PyArg_ParseTuple(args,"(dd)(dd)",
                          &p1.x, &p1.y, &p2.x, &p2.y)) {
        return NULL;
    }
    result = distance(&p1,&p2);
    return Py_BuildValue("d",result);
}
```

修饰符"|" 指定所有余下参数都是可选的。此修饰符在格式说明符中只能出现一次，而且不能嵌套使用。修饰符":" 指示参数的结束，位于它后面的任何文本都用作错误消息中的函数名称。修饰符";" 指示参数的结束，位于它后面的任何文本都用做错误消息。注意，只能使用: 和; 之中的一个。下面给出了一些例子：

```
PyArg_ParseTuple(args,"ii:gcd", &x, &y);
PyArg_ParseTuple(args,"ii; gcd requires 2 integers", &x, &y);

/* Parse with optional arguments */
PyArg_ParseTuple(args,"s|s", &buffer, &delimiter);
```

26.1.5 从C到Python的类型转换

下面的C函数用于将C变量中包含的值转换为Python对象：

```
PyObject *Py_BuildValue(char

 *format

, ...)
```

该函数根据一系列C变量构造一个Python对象。 *format* 是用于描述所需转换的字符串。余下参数是要转换的C变量的值。

format 说明符与PyArg_ParseTyple* 函数中所使用的类似，如表26-4所示。

表**26-4** **Py_BuildValue()** 函数的格式说明符

格式	**Python**类型	**C**类型	描　　述

""	None	void	无描述
"s"	字符串	char *	以Null 结束的字符串。如果C字符串指针是NULL ， 则返回None
"s#"	字符串	char *, int	字符串和长度。可能包含null 字节。如果C字符串指针是NULL ， 则返回None
"y"	字节	char *	同"s" ， 但返回的是字节字符串
"y#"	字节	char *, int	同"s#" ， 但返回的是字节字符串
"z"	字符串或None	char *	同"s"
"z#"	字符串或None	char *, int	同"s#"
"u"	Unicode	Py_UNICODE *	以Null 结束的Unicode字符串。如果字符串指针是NULL ， 则返回None
"u#"	Unicode	Py_UNICODE *	Unicode字符串和长度
"U"	Unicode	char *	将一个以null 结束的C字符串转换为Unicode字符串
"U#"	Unicode	char *, int	将C字符串转换为Unicode字符串
"b"	整数	char	8位整数
"B"	整数	unsigned char	8位无符号整数
"h"	整数	short	16位短整数
"H"	整数	unsigned short	无符号16位短整数
"i"	整数	int	整数
"I"	整数	unsigned int	无符号整数

"l"	整数	long	长整数
"L"	整数	unsigned long	无符号长整数
"k"	整数	long long	长长整数
"K"	整数	unsigned long long	无符号长长整数
"n"	整数	Py_ssize_t	Python大小类型
"c"	字符串	char	单个字符。创建长度为1的Python字符串
"f"	浮点数	float	单精度浮点数
"d"	浮点数	double	双精度浮点数
"D"	复数	Py_complex	复数
"O"	任意	PyObject *	任意Python对象。对象本身未改变，但它的引用计数增加1。如果指定一个NULL 指针，就会返回一个NULL 指针。如果需要传递在别处报出的错误，可以使用此说明符
"O&"	任意	converter, any	通过一个转换器函数处理的C数据
"S"	字符串	PyObject *	同"O"
"N"	任意	PyObject *	同"O"，但引用计数不会增加
"(*items*)"	元组	vars	创建一个元组。 *items* 是一个字符串，为本表中的格式说明符。 *vars* 是对应于 *items* 中元素的C变量列表
"[*items*]"	列表	vars	创建一个列表。 *items* 是一个字符串，为本表中的格式说明符。 *vars* 是对应于 *items* 中元素的C变量列表
"{*items*}"	字典	vars	创建一个字典

下面给出了构造不同类别值的一些例子：

```
Py_BuildValue("")                        None
Py_BuildValue("i",37)                    37
Py_BuildValue("ids",37,3.4,"hello")      (37, 3.5, "hello")
Py_BuildValue("s#","hello",4)            "hell"
Py_BuildValue("()")                      ()
Py_BuildValue("(i)",37)                  (37,)
Py_BuildValue("[ii]",1,2)                [1,2]
Py_BuildValue("[i,i]",1,2)               [1,2]
Py_BuildValue("{s:i,s:i}","x",1,"y",2)   {'x':1, 'y':2}
```

对于使用char * 的Unicode字符串转换，假定数据由一系列已经使用默认的Unicode编码格式（通常为UTF-8）编码的字节组成。传递给Python时，数据将被自动解码为Unicode字符串。唯一的例外是"y" 和"y#" 转换，它们返回的是原始字节字符串。

26.1.6 给模块添加值

在扩展模块的模块初始化函数中，经常会加入常量和其他支持值。使用下面的函数可以做到这一点：

```
int PyModule_AddObject(PyObject *module
, const char *name
, PyObject *value
)
```

为模块增加一个新值。 *name* 是值的名称，而value 是包含该值的Python对象。可以使用Py_BuildValue() 函数来构造这个值。

```
int PyModule_AddIntConstant(PyObject *module
, const char *name
, long value
)
```

为模块添加一个整数值。

```
void PyModule_AddStringConstant(PyObject *module
```

```
, const char *name
, const char *value
)
```

为模块添加一个字符串值。value 必须是以null 结尾的字符串。

```
void PyModule_AddIntMacro(PyObject *module
, macro
)
```

将宏值当作整数添加给模块。 *macro* 必须是预处理器宏的名称。

```
void PyModule_AddStringMacro(PyObject *module
, macro
)
```

将宏值当作字符串添加给模块。

26.1.7 错误处理

扩展模块指示错误的方式是给解释器返回NULL 。在返回NULL 之前，应该使用以下函数之一设置异常。

```
void PyErr_NoMemory()
```

引发MemoryError 异常。

```
void PyErr_SetFromErrno(PyObject *exc
```

```
)
```

引发一个异常。 *exc* 是异常对象。异常的值取自C库中的 *errno* 变量。

```
void PyErr_SetFromErrnoWithFilename(PyObject *exc
, char *filename
)
```

该函数很像PyErr_SetFromErrno() 函数，但在异常值中包含了文件名。

```
void PyErr_SetObject(PyObject *exc
, PyObject *val
)
```

引发一个异常。 *exc* 是一个异常对象，而 *val* 是包含异常值的对象。

```
void PyErr_SetString(PyObject *exc
, char *msg
)
```

引发一个异常。 *exc* 是一个异常对象，而msg 是描述错误的消息。

这些函数中， *exc* 参数的值可以是以下之一。

C名称	**Python**异常
PyExc_ArithmeticError	ArithmeticError

PyExc_AssertionError	AssertionError
PyExc_AttributeError	AttributeError
PyExc_EnvironmentError	EnvironmentError
PyExc_EOFError	EOFError
PyExc_Exception	Exception
PyExc_FloatingPointError	FloatingPointError
PyExc_ImportError	ImportError
PyExc_IndexError	IndexError
PyExc_IOError	IOError
PyExc_KeyError	KeyError
PyExc_KeyboardInterrupt	KeyboardInterrupt
PyExc_LookupError	LookupError
PyExc_MemoryError	MemoryError
PyExc_NameError	NameError
PyExc_NotImplementedError	NotImplementedError
PyExc_OSError	OSError
PyExc_OverflowError	OverflowError
PyExc_ReferenceError	ReferenceError
PyExc_RuntimeError	RuntimeError
PyExc_StandardError	StandardError

PyExc_StopIteration	StopIteration
PyExc_SyntaxError	SyntaxError
PyExc_SystemError	SystemError
PyExc_SystemExit	SystemExit
PyExc_TypeError	TypeError
PyExc_UnicodeError	UnicodeError
PyExc_UnicodeEncodeError	UnicodeEncodeError
PyExc_UnicodeDecodeError	UnicodeDecodeError
PyExc_UnicodeTranslateError	UnicodeTranslateError
PyExc_ValueError	ValueError
PyExc_WindowsError	WindowsError
PyExc_ZeroDivisionError	ZeroDivisionError

以下函数用于查询或清除解释器的异常状态。

```
void PyErr_Clear()
```

该函数用于清除所有以前引发的异常。

```
PyObject *PyErr_Occurred()
```

检查是否引发了异常。如果是，则返回当前的异常值，否则返回NULL 。

```
```

```
int PyErr_ExceptionMatches(PyObject *exc

)
```

检查当前异常是否匹配异常 *exc* 。如果匹配则返回1，否则返回0。该函数的异常匹配规则与Python代码中的相同。因此， *exc* 可以是当前异常的超类或异常类的元组。

以下原型说明了如何在C语言中实现try-except 代码块：

```
/* 执行涉及Python对象的某个操作 */
if (PyErr_Occurred()) {
    if (PyErr_ExceptionMatches(PyExc_ValueError)) {
        /* 采取某种恢复措施 */
        ...
        PyErr_Clear();
        return result; /* 有效的PyObject * */
    } else {
        return NULL; /* 将异常传递给解释器 */
    }
}
```

26.1.8 引用计数

与使用Python语言编写的程序不同，C语言扩展可能必须操作Python对象的引用计数。可以使用下面的宏来实现，它们都适用于PyObject * 类型的对象。

宏	描　　述
Py_INCREF(obj)	增加 *obj* 的引用计数，不能为null
Py_DECREF(obj)	减少 *obj* 的引用计数，不能为null
Py_XINCREF(obj)	增加 *obj* 的引用计数，可以为null
Py_XDECREF(obj)	减少 *obj* 的引用计数，可以为null

在C语言中，操作Pyton对象的引用计数是一个棘手的话题，我们强烈建议读者在继续深入了解之前参考“扩展与嵌入Python解释器”文档，地址是http://docs.python.org/extending 。一般而言，除了以下情况之外，不必担心C扩展函数中的引用计数。

- 如果保存对Python对象的引用以便稍后在C结构中使用，必须增加其引用计数。
- 类似地，要销毁以前保存的对象，必须减少其引用计数。

- 如果在C语言中操作Python容器（列表、字典等），必须手动操作容器中每个项的引用计数。例如，获取或设置容器中项的高级操作一般会增加引用计数。

如果扩展代码导致解释器崩溃（忘记增加引用计数）或者在使用扩展函数时解释器出现内存泄露（忘记减少引用计数），就意味着存在引用计数问题。

26.1.9 线程

全局解释器锁定用于防止多个线程在解释器中同时执行。如果在扩展模块中编写的某个函数执行了很长时间，那么在它完成之前会一直阻塞其他线程的执行。这是因为只要调用扩展函数，它就会保持锁定。如果扩展模块是线程安全的，可以使用下面的宏来释放和重新获取全局解释器锁。

```
Py_BEGIN_ALLOW_THREADS
```

释放全局解释器锁，同时允许其他线程在解释器中运行。释放锁时，禁止C扩展调用Python C API中的任何函数。

```
Py_END_ALLOW_THREADS
```

重新获取全局解释器锁。使用该函数时，扩展将会阻塞，直到成功获取锁。

下面的例子说明了这些宏的用法：

```
PyObject *py_wrapper(PyObject *self, PyObject *args) {
         ...
         PyArg_ParseTuple(args, ...)
         Py_BEGIN_ALLOW_THREADS
         result = run_long_calculation(args);
         Py_END_ALLOW_THREADS
         ...
         return Py_BuildValue(fmt,result);
}
```

26.2 嵌入Python解释器

也可以将Python解释器嵌入到C应用程序中。嵌入之后，Python解释器的运行类似于一个编程库。C程序可以初始化解释器，让解释器运行脚本和代码片段，载入库模块，以及操作Python中实现的函数和对象。

26.2.1 嵌入模板

C程序通过嵌入可以操作解释器。下面这个简单的C程序说明了最基本的嵌入形式：

```
#include <Python.h>

int main(int argc, char **argv) {
    Py_Initialize();
    PyRun_SimpleString("print('Hello World')");
    Py_Finalize();
    return 0;
}
```

这个例子初始化了解释器，以字符串的形式执行短脚本，然后关闭解释器。在了解进一步的内容之前，最好先学会这个例子。

26.2.2 编译与链接

要在UNIX上编译嵌入的解释器，代码必须包含`Python.h`头文件并链接到像`libpython2.6.a`这样的解释器库。头文件所在的位置一般为/usr/local/include/python2.6，而库所在的位置一般为/usr/local/lib/python2.6/config。在Windows上，这些文件位于Python安装目录中。注意，解释器可能需要其他的库，因此链接时需要包含这些库。不幸的是，这往往特定于平台，而且与计算机上对Python的配置有关——用户可能必须在这些方面花些功夫。

26.2.3 基本的解释器操作与设置

下面的函数用于启动解释器并运行脚本。

```
int PyRun_AnyFile(FILE *fp

, char *filename

)
```

如果 *fp* 是一个交互式设备，如UNIX中的tty，该函数将调用`PyRun_InteractiveLoop()`函数。否则将调用`PyRun_SimpleFile()`函数。*filename* 是用于对输入流命名的字符串。解释器报告错误时将出现该名称。如果 *filename* 为`null`，就会使用默认字符串`???`作为文件名。

```
int PyRun_SimpleFile(FILE *fp

, char *filename
```

```
)
```

该函数类似于PyRun_SimpleString() 函数，但会从文件 *fp* 读取程序。

```
int PyRun_SimpleString(char *command
)
```

执行解释器__main__ 模块中的command 。执行成功返回0，出现异常则返回-1。

```
int PyRun_InteractiveOne(FILE *fp
, char *filename
)
```

执行单条交互式命令。

```
int PyRun_InterativeLoop(FILE *fp
, char *filename
)
```

以交互模式运行解释器。

```
void Py_Initialize()
```

初始化Python解释器。应该在使用C API中的其他函数之前调用该函数，否则会引发Py_Set ProgramName()、PyEval_InitThreads()、PyEval_ReleaseLock() 和

PyEval_AcquireLock() 异常。

```
int Py_IsInitialized()
```

如果解释器已经初始化，则返回1，否则返回0。

```
void Py_Finalize()
```

通过将调用Py_Initialize() 函数后创建的所有子解释器和对象进行销毁来清理解释器。通常，该函数将释放解释器分配的所有内存。但循环引用和扩展模块引起的内存泄露不能通过该函数恢复。

```
void Py_SetProgramName(char *name

)
```

设置程序名称，它通常是sys 模块的argv[0] 参数。应该在Py_Initialize() 函数之前调用该函数。

```
char *Py_GetPrefix()
```

返回已安装的平台独立文件的前缀。其返回值与sys.prefix 中的相同。

```
char *Py_GetExecPrefix()
```

返回已安装的平台独立文件的exec-prefix 。其返回值与sys.exec-prefix 中的相同。

```
char *Py_GetProgramFullPath()
```

返回Python可执行文件的完整路径名称。

```
char *Py_GetPath()
```

返回默认的模块搜索路径。返回的路径是一个字符串，其中包含由独立于平台的分隔符（UNIX上是:，DOS/Windows上是;）分隔的目录名称。

```
int PySys_SetArgv(int argc
, char **argv
)
```

设置用于填充`sys.argv` 值的命令行选项。应该在`Py_Initialize()` 函数之前调用该函数。

26.2.4 在C语言中访问Python

从C语言访问解释器存在多种方式，但嵌入时一般都需要完成4个基本任务。

- 导入Python库模块（模拟`import` 语句）。
- 获得对模块中定义的函数的引用。
- 调用Python函数、类和方法。
- 访问对象的属性（数据、方法等）。

所有这些操作都可以使用Python C API中定义的这些基本操作来完成。

```
PyObject *PyImport_ImportModule(const char *modname
)
```

导入模块 *modname* ，并返回对相关模块对象的引用。

```
PyObject *PyObject_GetAttrString(PyObject *obj
, const char *name
)
```

获取对象的属性，同 *obj.name* 。

```
int PyObject_SetAttrString(PyObject *obj
, const char *name
, PyObject *value
)
```

设置对象的属性，同 *obj.name = value* 。

```
PyObject *PyEval_CallObject(PyObject *func
, PyObject *args
)
```

使用参数 *args* 调用 *func* 。 *func* 是Python可调用对象（函数、方法、类等），*args* 是参数元组。

```
PyObject *

PyEval_CallObjectWithKeywords(PyObject *func, PyObject *args, PyObject *kwargs)
```

使用位置参数 *args* 和关键字参数 *kwargs* 调用 *func* 。 *func* 是可调用对象， *args* 是元组，而 *kwargs* 是字典。

下面的例子通过在C语言中调用和使用re 模块的各个部分，说明了这些函数的用法。在这个例子中，stdin 包含用户提供的Python正则表达式，而该程序打印出从stdin中读出的所有行。

```
#include "Python.h"

int main(int argc, char **argv) {
    PyObject *re;
    PyObject *re_compile;
    PyObject *pat;
    PyObject *pat_search;
    PyObject *args;
    char     buffer[256];

    if  (argc != 2) {
        fprintf(stderr,"Usage: %s pattern\n",argv[0]);
        exit(1);
    }

    Py_Initialize();
    /* import re */
    re   = PyImport_ImportModule("re");

    /* pat = re.compile(pat,flags) */
    re_compile = PyObject_GetAttrString(re,"compile");
    args = Py_BuildValue("(s)", argv[1]);
    pat = PyEval_CallObject(re_compile, args);
    Py_DECREF(args);

    /* pat_search = pat.search - bound method*/
    pat_search = PyObject_GetAttrString(pat,"search");

    /* Read lines and perform matches */
    while (fgets(buffer,255,stdin)) {
       PyObject *match;
       args = Py_BuildValue("(s)", buffer);

       /* match = pat.search(buffer) */
       match = PyEval_CallObject(pat_search,args);
       Py_DECREF(args);
       if (match != Py_None) {
           printf("%s",buffer);
       }
       Py_XDECREF(match);
    }
    Py_DECREF(pat);
    Py_DECREF(re_compile);
    Py_DECREF(re);
    Py_Finalize();
    return 0;
}
```

在嵌入代码中，正确管理引用计数是非常重要的工作。特别是在执行完函数后，要减少从C语言创建或返回给C语言的所有对象的引用计数。

26.2.5 将Python对象转换为C对象

嵌入使用解释器的一个主要问题是，需要将Python函数或方法调用的结果转换为合适的C语言表示。一般来说，用户需要提前知道操作返回数据的确切类型。然而，不存在像`PyArg_ParseTyple()`这样方便的高级函数可用于转换单个对象的值。但下面列出了一些底层转换函数，可将一些主要的Python数据类型转换为相应的C语言表示，前提是知道要处理的Python对象的确切类型。

Python-C转换函数

```
long          PyInt_AsLong(PyObject *)

long          PyLong_AsLong(PyObject *)

double        PyFloat_AsDouble(PyObject *)

char          *PyString_AsString(PyObject *)（仅在Python 2中使用）

char          *PyBytes_AsString(PyObject *)（仅在Python 3中使用）
```

对于很多比这些复杂得多的类型，需要参考C API文档（http://docs.python.org/c-api）。

26.3 ctypes

通过`ctypes`模块，Python能够访问在DLL和共享库中定义的C函数。使用`ctypes`模块访问C代码不必编写C扩展包装器代码或者使用C编译器编译任何内容，但需要了解关于底层C库的一些细节（名称、调用参数、类型等）。`ctypes`是一个很大的库模块，其中包含大量高级功能。此处我们会介绍使用该库所必需的基本部分。

26.3.1 加载共享库

下面的类用于加载C共享库并返回表示其内容的实例。

```
CDLL(name
 [, mode
 [, handle
 [, use_errno
 [, use_last_error
]]]])
```

此类代表一个标准的C共享库。 *name* 是库名称，如`'libc.so.6'`或`'msvcrt.dll'`

。 *mode* 是一个标志，用于确定如何加载库并把它传递给UNIX上的底层函数dlopen() 。可以把这个标志置为RTLD_LOCAL 、RTLD_GLOBAL 或RTLD-DEFAULT （默认值）的按位OR。在Windows上， *mode* 将被忽略。handle 指定指向已经加载的库（如果可用的话）的句柄，其默认值为None 。 *use_errno* 是布尔型标志，用于给处理已加载库中的C *errno* 变量增加一个额外的安全层。在调用任何外来函数然后恢复值之前，这一层保存了 *errno* 的一份线程局部副本。默认情况下， *use_errno* 的值为False 。 *use_last_error* 是布尔型标志，支持使用一对函数get_last_error() 和 set_last_error() 操纵系统错误代码。这些函数在Windows上更加常用。默认情况下， *use_last_error* 的值是False 。

```
WinDLL(name
 [, mode
 [, handle
 [, use_errno
 [, use_last_error
]]]])
```

该函数与CDLL() 函数相同，但它假定库中的函数遵循Windows stdcall 调用约定（Windows）。

下面这个实用工具函数可找到系统上的共享库，并构造一个适合用作前面类中 *name* 参数的名称。该函数定义在ctypes.util 子模块中。

```
find_library(name
)
```

该函数定义在ctypes.util 中，返回库 *name* 的路径名称。 *name* 是没有任何文件后缀的库名称，如'libc' 、'libm' 等。该函数返回的字符串是一个完整的路径名称，如'/usr/lib/libc.so.6' 。该函数的行为是高度依赖于系统的，与共享库和环境的底层配置都有紧密联系（例如，LD_LIBRARY_PATH 和其他参数的设置）。如果无法找到库，则返回None。

26.3.2 外来函数

CDLL() 类创建的共享库实例好比底层C语言库的代理。访问库的内容时，只要使用

属性查找（运算符）即可，例如：

```
>>> import ctypes


>>> libc = ctypes.CDLL(

"/usr/lib/libc.dylib

")


>>> libc.rand()


16807
>>> libc.atoi(

"12345

")


12345
>>>
```

在这个例子中，libc.rand() 和libc.atoi() 这样的操作直接调用已加载的C语言库中的函数。

ctypes 假定所有函数均接受类型为int 或char* 的参数，并返回int 类型的结果。因此，尽管前面的函数调用“成功”，但调用其他C库函数的结果也会出乎意料。例如：

```
>>> libc.atof(

"34.5

")


-1073746168
>>>
```

要解决这个问题，可以修改以下属性来设置类型标记和外来函数func 的处理方式：

```


func

.argtypes
```

这是一个ctypes数据类型的元组，用于描述 *func* 的输入参数。

```
func
.restype
```

这是一个ctypes数据类型，用于描述 *func* 的返回值。如果函数返回`void`，它的值为`None`。

```
func
.errcheck
```

这是一个Python可调用对象，带有3个参数（*result*、*func*、*args*），其中 *result* 是外来函数的返回值，`func` 是对外来函数自身的引用， *args* 是一个输入参数的元组。该函数是在外来函数调用之后进行调用，可执行错误检查和其他操作。

下面这个例子修复了前面例子中的`atof()` 函数接口：

```
>>> libc.atof.restype=ctypes.c_double


>>> libc.atof(

"34.5

")


34.5
>>>
```

`ctypes.d_double` 是对预定义数据类型的引用。下一节将会介绍这些数据类型。

26.3.3 数据类型

表26-5列出了可以用在外来函数的`argtypes` 和`restype` 属性中的ctypes数据类型。

Python值一栏描述的是指定数据类型接受的Python数据类型。

表**26-5　ctypes**数据类型

ctypes类型名称	**C**数据类型	**Python**值
c_bool	bool	Ture 或False
c_bytes	signed char	小整数
c_char	char	单个字符
c_char_p	char *	以Null结尾的字符串或字节
c_double	double	浮点数
c_longdouble	long double	浮点数
c_float	float	浮点数
c_int	int	整数
c_int8	signed char	8位整数
c_int16	short	16位整数
c_int32	int	32位整数
c_int64	long long	64位整数
c_long	long	整数
c_longlong	long long	整数
c_short	short	整数
c_size_t	size_t	整数
c_ubyte	unsigned char	无符号整数

c_uint	unsigned int	无符号整数
c_uint8	unsigned char	8位无符号整数
c_uint16	unsigned short	16位无符号整数
c_uint32	unsigned int	32位无符号整数
c_uint64	unsigned long long	64位无符号整数
c_ulong	unsigned long	无符号整数
c_ulonglong	unsigned long long	无符号整数
c_ushort	unsigned short	无符号整数
c_void_p	void *	整数
c_wchar	wchar_t	单个Unicode字符
c_wchar_p	wchar_t *	以Null结尾的Unicode

对其中的类型使用以下函数，可以创建表示C指针的类型。

```
POINTER(type
)
```

定义了一种类型，它是指向类型 *type* 的指针。例如，POINTER(c_int) 表示C类型 int* 。

如果要定义表示固定大小C数组的类型，将现有类型乘以代表数组维度的数字即可。例如，c_int*4 表示C数据类型int[4] 。

如果要定义一种表示C语言中结构或联合的类型，需要从基类Structure 或Union 进行继承。在每个派生类中，定义类变量_fields_ 来描述内容。_fields_ 是2个或3个元组的列表，元组的形式为(*name, ctype*) 或(*name, ctype, width*)，其中name 是

结构字段的标识符， *ctype* 是描述类型的ctype类，而width 是整数位字段宽度。例如，考虑下面的C结构：

```
struct Point {
   double x, y;
};
```

这种结构的ctypes描述是：

```
class Point(Structure):
   _fields_ = [ ("x", c_double),
                ("y", c_double) ]
```

26.3.4 调用外来函数

如果要调用一个库中的函数，只要使用符合其类型标记的一组参数调用正确的函数即可。对于像c_int 、c_double 这样的简单数据类型，只要传递兼容的Python类型作为输入即可，如整数、浮点数等。还可以传递c_int 、c_double 及类似类型的实例作为输入。对于数组，只要传递兼容类型的Python序列即可。

如果要将一个指针传递给外来函数，必须首先创建一个可代表被指向值的一个ctype实例，然后使用以下函数之一创建一个指针对象：

```
byref(cvalue

 [, offset

])
```

该函数表示指向 *cvalue* 的轻量级指针。 *cvalue* 必须是ctypes数据类型的一个实例。 *offset* 是要添加给指针值的字节位移。该函数的返回值只能在函数调用中使用。

```
pointer(cvalue

)
```

该函数创建指向 *cvalue* 的指针实例。 *cvalue* 必须是ctypes数据类型的一个实例。该函数创建的实例属于前面讲过的POINTER 类型。

下面这个例子说明如何将一个double * 类型的参数传递给C函数：

```
dval = c_double(0.0)        # 创建一个double类型的实例
r = foo(byref(dval))        # 调用foo(&dval)

p_dval = pointer(dval)      # 创建一个指针变量
r = foo(p_dval)            # 调用foo(p_dval)

# 随后检查dval的值
print (dval.value)
```

应该注意，不能创建指向内置类型（如int 或float ）的指针。如果底层的C函数修改了值，传递指向这些类型的指针将违反其可变性。

ctypes实例 *cobj* 的属性 *cobj*. value 包含内部数据。例如，引用前面代码中的dval.value 将返回保存在ctypes c_double 实例dval 中的浮点值。

如果要将一个结构传递给C函数，必须创建结构或联合的一个实例。为此需要调用前面定义的结构或联合类型StructureType ，如下所示：

```
StructureType(*args

, **kwargs

)
```

创建StructureType 的一个实例，其中StructureType 是从Structure 或Union 派生而来的类。 **args* 中的位置参数用于按照_fields_ 中列出的先后顺序初始化结构成员。 ***kwargs* 中的关键字参数只会初始化已命名的结构成员。

26.3.5 其他类型构造方法

ctypes类型（如c_int 、POINTER 等）的所有实例都有一些类方法，用于从内存位置和其他对象创建ctypes类型的实例。

```
ty.from_buffer(source

 [,offset

])
```

该函数创建ctypes类型 *ty* 的一个实例， *ty* 与 *source* 共享同一块缓存。 *source* 必须是支持可写缓存接口（如array 模块中的bytearray 、array、mmap 等）的任意其他对象。 *offset* 是距离要使用的缓存开始处的字节数。

```
ty
.from_buffer_copy(source
 [, offset
])
```

该函数与ty.from_buffer() 相同，但会制作一份内存的副本，而且 *source* 可以是只读的。

```
ty
.from_address(address
)
```

根据原始内存地址 *address* 创建ctypes类型 *ty* 的实例， *address* 被指定为一个整数。

```
ty
.from_param(obj
)
```

根据Python对象 *obj* 创建ctypes类型 *ty* 的实例。只有所传递的对象 *obj* 能够被转换为适当类型时，该函数才有效。例如，可以将Python整数转换为c_int 实例。

```
ty
```

```
.in_dll(library
, name
)
```

根据共享库中的符号创建ctypes类型 *ty* 的一个实例。`library` 是已加载库的实例，如`CDLL` 类创建的对象。 *name* 是符号的名称。此方法可用于给定义在库中的全局变量加上ctypes包装器。

下面的例子说明如何给创建对`libexample.so` 库中定义的全局变量`int status` 的引用。

```
libexample = ctypes.CDLL("libexample.so")
status = ctypes.c_int.in_dll(libexample,"status")
```

26.3.6 实用工具函数

ctypes定义了以下实用工具函数。

```
addressof(cobj
)
```

以整数形式返回 *cobj* 的内存地址。 *cobj* 必须是`ctypes` 类型的实例。

```
alignment(ctype_or_obj
)
```

返回一个ctypes类型或对象的整数对齐要求。 *ctype_or_obj* 必须是ctypes类型或类型的实例。

```
cast(cobj
, ctype
```

```
)
```

将ctypes对象 *cobj* 强制转换为ctype中指定的新类型。这只对指针有效，因此 *cobj* 必须是指针或数组，而且 *ctype* 必须是指针类型。

```
create_string_buffer(init
 [, size
])
```

创建一块可变的字符缓存，作为一个c_char 类型的ctypes数组。 *init* 是一个整数大小或是一个字符串，表示初始内容。size 是可选参数，指定当 *init* 是字符串时要使用的大小。默认情况下， *size* 的值要比 *init* 中的字符数大1。使用默认编码将Unicode字符串编码为字节。

```
create_unicode_buffer(init
 [, size
])
```

该函数与create_string_buffer() 相同，但会创建一个c_wchar 类型的ctypes数组。

```
get_errno()
```

返回 *errno* 的ctypes私有副本的当前值。

```
get_last_error()
```

返回Windows上`LastError` 的ctypes私有副本的当前值。

```
memmove(dst
, src
, count
)
```

将 *`count`* 个字节从 *`src`* 复制到 *`dst`* 。 *`src`* 和 *`dst`* 可以是表示内存地址的整数，也可以是能转换为指针的ctypes类型实例。该函数同C语言的`memmove()` 库函数相同。

```
memset(dst
, c
, count
)
```

将从`dst` 开始的 *`count`* 个内存字节置为字节值 *`c`* 。 *`dst`* 是一个整数或是一个ctypes实例。 *`c`* 是一个整数，表示范围在0～255之间的一个字节。

```
resize(cobj
, size
)
```

调整用于表示ctypes对象 *`cobj`* 的内部内存大小。 *`size`* 表示新的大小值，单位是字节。

```
set_conversion_mode(encoding
, errors
```

```
)
```

设置从Unicode字符串转换为8位字符串时使用的Unicode编码。 *encoding* 是编码名称，如'utf-8'，而 *errors* 是错误处理策略，如'strict'或'ignore'。该函数使用前面的设置返回一个元组(*encoding* , *errors*)。

```
set_errno(value

)
```

设置系统变量 *errno* 的ctypes私有副本并返回上一个值。

```
set_last_error(value

)
```

设置Windows LastError 变量并返回上一个值。

```
sizeof(type_or_cobj

)
```

返回ctypes类型或对象的大小，单位是字节。

```
string_at(address

 [, size

])
```

该函数返回一个字节字符串，表示从地址 *address* 开始的 *size* 个内存字节。如果省略 *size* ，它将假定字节字符串是以null 结尾的。

```
wstring_at(address

 [, size

])
```

返回一个Unicode字符串，表示从地址 *address* 开始的 *size* 个字符。如果省略 *size* ，它将假定字节字符串是以NULL 结尾的。

26.3.7 示例

下面的例子通过为本章第一部分（讲述手动创建Python扩展的详细内容）中使用的C函数集构造一个接口，来说明ctypes 模块的用法。

```
# example.py

import ctypes
_example = ctypes.CDLL("./libexample.so")

# int gcd(int, int)
gcd = _example.gcd
gcd.argtypes = (ctypes.c_int,
                ctypes.c_int)
gcd.restype = ctypes.c_int

# int replace(char *s, char olcdh, char newch)
_example.replace.argtypes = (ctypes.c_char_p,
                             ctypes.c_char,
                             ctypes.c_char)
_example.replace.restype = ctypes.c_int

def replace(s, oldch, newch):
    sbuffer = ctypes.create_string_buffer(s)
    nrep = _example.replace(sbuffer,oldch,newch)
    return (nrep,sbuffer.value)

# double distance(Point *p1, Point *p2)
class Point(ctypes.Structure):
    _fields_ = [ ("x", ctypes.c_double),
                 ("y", ctypes.c_double) ]

_example.distance.argtypes = (ctypes.POINTER(Point),
                              ctypes.POINTER(Point))
_example.distance.restype = ctypes.c_double

def distance(a,b):
    p1 = Point(*a)
    p2 = Point(*b)
```

```
    return _example.distance(byref(p1),byref(p2))
```

一般而言，使用ctypes 总是要涉及复杂程度不尽相同的Python包装器层。例如，你或许可以直接调用某个C函数，但是还必须实现一个小的包装层用于处理底层C代码的某些问题。在这个例子中，replace() 函数进行了额外的处理，应对C库改变了输入缓存的问题。distance() 函数也进行了额外的处理，先根据元组创建Point 实例然后再传递指针。

> 注意
>
> 在ctypes模块中，还有很多高级特性在这里没有讲到。例如，库可以访问Windows上众多不同的库，还支持回调函数和不完整类型，以及其他细节。在线文档中给出了大量的示例，可以作为进一步使用该模块的起点。

26.4 高级扩展与嵌入

使用简单的C代码扩展Python时，创建手写的扩展模块或者使用ctypes通常很直观。但是，对于更加复杂的扩展，情况很快就会变得很糟糕。因此你将需要一种更方便的扩展构建工具。这些工具要么能够自动化很多扩展构建过程，要么能够提供可以在更高层次上操作的编程接口。http://wiki.python.org/moin/IntegratingPythonWithOtherLanguages 上提供了很多此类工具的链接。下面给出SWIG（http://www.swig.org ）的一个简短例子，仅作参考。为了信息全面公开，应该注意SWIG原来是由本书作者开发的。

借助自动化工具，通常只要在高层次上描述扩展模块的内容即可。例如，使用SWIG可以编写下面这样简洁的接口规范：

```
/* example.i : Sample Swig specification */
%module example
%{
/* Preamble. Include all required header files here */
#include "example.h"
%}

/* Module contents.  List all C declarations here */
typedef struct Point {
  double x;
  double y;
} Point;
extern int    gcd(int, int);
extern int    replace(char *s, char oldch, char newch);
extern double distance(Point *a, Point *b);
```

使用此规范，SWIG将自动生成在构建一个Python扩展模块时所需的一切内容。像调用编译器一样调用SWIG，即可运行：

```
% swig -python example.i


%
```

它生成的输出是一组.c 和.py 文件，但用户通常不必担心这个问题。如果使用distutiles 并在设置规范中包含一个.i 文件，构建扩展时将自动运行SWIG。例如，这个setup.py 文件将在列出的example.i 文件上自动运行SWIG。

```
# setup.py
from distutils.core import setup, Extension
setup(name="example",
      version="1.0",
      py_modules = ['example.py'],
      ext_modules = [
        Extension("_example",
                  ["example.i","example.c"])
        ]
      )
```

原来，这个example.i 文件和setup.py 文件是这个例子中创建一个有效扩展模块所需的全部内容。如果输入python setup.py build_ext --inplace，就会发现目录中出现了一个完整有效的扩展。

26.5 Jython和IronPython

扩展和嵌入并不限于C程序。如果使用Java，可以考虑使用Jython（http://www.jython.org ），它是Python解释器在Java中的完整实现。借助jython，仅使用import 语句就可导入Java库，例如：

```
bash-3.2$ jython
Jython 2.2.1 on java1.5.0_16
Type "copyright", "credits" or "license" for more information.
>>> from java.lang import System


>>> System.out.println(

"Hello World

")


Hello World
>>>
```

如果在Windows上使用.NET框架，可以考虑使用IronPython（http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython ），它是Python解释器在C#中的完整实现。借助IronPython，可以轻松地用类似方式从Python访问所有.NET库：

```
%
```

```
 ipy


IronPython 1.1.2 (1.1.2) on .NET 2.0.50727.42
Copyright (c) Microsoft Corporation. All rights reserved.
>>> import System.Math


>>> dir(System.Math)


['Abs', 'Acos', 'Asin', 'Atan', 'Atan2', 'BigMul', 'Ceiling', 'Cos', 'Cosh', ...]
>>> System.Math.Cos(3)


-0.9899924966
>>>
```

详细介绍jython 和IronPython 已经超出了本书的范围。不过，只要记住它们都是Python就行了，它们与标准实现之间最主要的区别在于它们的内置库。

附录　Python 3

2008年12月，Python 3发布了。Python 3是Python语言的一次大幅提升，它在很多关键领域与Python 2不具备向后兼容性。在http://docs.python.org/3.0/whatsnew/3.0.html 上可以找到*What's New in Python 3.0* 文档，其中完整地阐述了Python 3中的变化。从某种意义上说，本书的前26章与该文档是南辕北辙的。也就是说，迄今为止讲述的所有内容都是Python 2和Python 3所共有的。这包括所有的标准库模块、主要的语言特性和示例。除了一些不起眼的命名差异和`print()` 是函数之外，没有描述过其他Python 3所独有的特性。

本附录重点介绍只在Python 3中可用的语言特性，以及在迁移现有代码时要牢记的一些重要区别。本附录的结尾描述了一些移植策略和`2to3` 代码转换工具的使用方法。

A.1　谁应该使用Python 3

深入介绍之前，还有一个重要的问题有待解决，即谁应该使用Python 3.0版本。在Python社区中有一条共识，即向Python 3的转变不会在一夜之间完成，而且Python 2分支在未来的一段时间（数年）内将继续存在。因此到本书撰写之际，Python 2代码还很普遍。我预测，几年之后本书的第5版出版时，人们依然还在开发大量的Python 2代码。

使用Python 3.0的一个主要问题是与第三方库的兼容性。Python的强大多来自它的各种框架和库。但除非将这些库显式地迁移到Python 3中，否则几乎肯定无法使用。很多库都依赖于其他的库，而这些其他的库又依赖于更多其他的库，问题由此变得越来越复杂。在本书撰写之际（2009年），还有很多重要的Python库和框架没有迁移到Python 2.4中，更不用说是2.6或3.0了。因此，如果使用Python的目的是运行第三方代码，目前最好仍继续使用Python 2。如果您看到本书的时候已经到了2012年，估计这种情形已经得到显著地改变。

尽管Python 3去除了语言中的很多瑕疵，但对于只是学习基础内容的新用户来说是否为一个明智的选择，这个问题的答案目前尚不明朗。几乎所有现存的文档、教程、技术手册和示例都假定是Python 2，使用的编码约定与Python 3不兼容。毋庸置疑，如果一切学习材料都是脱节的，用户是不可能拥有良好学习体验的。即便是官方文档也没有完全更新到Python 3，编写本书时，作者提交了大量关于文档错误和疏漏的bug报告。

最后，即使Python 3.0声称是最新和最好的版本，它也存在大量性能和运行上的问题。例如在最初发行的版本中，I/O系统的运行时性能确实很差，以致让人无法接受。字节和Unicode的分离也不是没有问题。即使是一些内置的库模块，也由于I/O和字符串处理方面的一些变化而无法使用。显然，随着更多程序员对此版本进行压力测试，这些问题也会得到改进。但以本书作者的观点来看，Python 3.0只适合经验十分丰富的Python程序员使用。如果主要注重稳定性和代码的高质量，请继续使用Python 2，直到Python 3系列存在的问题得以解决。

A.2　新的语言特性

本节扼要介绍Python 2中尚不支持的一些Python 3特性。

A.2.1 源代码编码和标识符

Python 3假定源代码使用UTF-8进行编码。另外，关于标识符中哪些字符是合法的规则也放宽了。具体来说，标识符可以包含代码点为U+0080及以上的任意有效Unicode字符。例如：

```
π.= 3.141592654
r = 4.0
print(2*π*r)
```

可以在源代码中使用这种字符并不意味着这就是一种好的做法。并非所有的编辑器、终端或开发工具在处理Unicode方面都同样老到。此外，强迫程序员敲奇怪的键序列输入标准键盘上没有的字符，也是一件令人生厌的事情。（这可能会激发办公室里的一些老黑客的兴趣，给大家讲另一个有趣的APL故事。）因此，最好是保留在注释和字符串字面量中使用Unicode字符的做法。

A.2.2 集合字面量

在大括号`{`*`items`*` }`中放入一组值就可以定义一个集合，例如：

```
days = { 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun' }
```

这种语法的作用与使用`set()`函数相同：

```
days = set(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])
```

A.2.3 集合与字典推导

语法`{ `*`expr`*` for `*`x`*` in `*`s`*` if `*`condition`*` }`称为集合推导。它将一种操作应用给集合`s`的所有元素，使用方式与列表推导类似。例如：

```
>>> values = { 1, 2, 3, 4 }


>>> squares = {x*x for x in values}


>>> squares


{16, 1, 4, 9}
>>>
```

语法{ *kexpr:vexpr* for *k,v* ins if *condition* }称为字典推导。它将一种操作应用给(*key, value*)元组序列 *s* 中的所有键和值，然后返回一个字典。表达式 *kexpr* 用于描述新字典的键，而表达式 *vexpr* 用于描述它的值。这可以看作是dict()函数的一个增强版。

举个例子，假定有一个股票价格文件'prices.dat'，如：

```
GOOG 509.71
YHOO 28.34
IBM 106.11
MSFT 30.47
AAPL 122.13
```

下面这个程序将上面这个文件读取到一个字典中，然后使用字典推导将股票名称映射到价格：

```
fields = (line.split() for line in open("prices.dat"))
prices = {sym:float(val) for sym,val in fields}
```

下面这个例子将价格字典中的所有键转换为小写形式：

```
d = {sym.lower():price for sym,price in prices.items()}
```

下面这个例子为价格在100美元以上的股票创建价格字典：

```
d = {sym:price for sym,price in prices.items() if price >= 100.0}
```

A.2.4 扩展的可迭代对象解包

在Python 2中，使用如下语法可将一个可迭代对象中的项解包到一些变量中：

```
items = [1,2,3,4]
a,b,c,d = items         # 将各个项解包到变量中
```

只有变量个数与要解包的项数完全相等，解包才能进行。

在Python 3中，可以使用通配符变量仅仅解包一个序列中的某些项，而将其他值放到一个列表中，例如：

```
a,*rest   = items      # a = 1, rest = [2,3,4]
a,*rest,d = items      # a = 1, rest = [2,3], d = 4
*rest, d  = items      # rest = [1,2,3], d = 4
```

在这些例子中，使用*前缀的变量接收所有其他的值，并将它们放入一个列表中。如

果不存在其他的项，该列表将为空。这种特性的用途之一是在循环元组（或序列）列表时，元组的大小可以不等，例如：

```
points = [ (1,2), (3,4,"red"), (4,5,"blue"), (6,7) ]
for x,y, *opt in points:
    if opt:
        # 找到其他的字段
    statements
```

使用这种语法时，只能出现一个带有* 号的变量。

A.2.5 Nonlocal变量

使用nonlocal 声明后，内部函数可以修改外部函数中的变量，例如：

```
def countdown(n):
    def decrement():
        nonlocal n
        n -= 1
    while n > 0:
        print("T-minus", n)
        decrement()
```

在Python 2中，内部函数可以读取外部函数中的变量，但不能修改它们。使用nonlocal 声明后就能修改它们了。

A.2.6 函数注释

可以使用任意值来注释函数的参数和返回值，例如：

```
def foo(x:1,y:2) -> 3:
    pass
```

函数属性__annotations__ 是一个字典，作用是将参数名称映射到注释值。特殊的'return' 键被映射为返回值注释，例如：

```
>>> foo.__annotations


__
{'y': 4, 'x': 3, 'return': 5}
>>>
```

解释器不会赋予这些注释任何意义。实际上，它们可以是任何值，但类型信息可能是未来最有用的。例如可以这样写：

```
def foo(x:int, y:int) -> str:
```

```
    statements
```

注释并不限于单个值。注释可以是任意有效的Python表达式。同样的语法也适用于各种位置参数和关键字参数，例如：

```
def bar(x, *args:"additional", **kwargs:"options"):
    statements
```

再次强调，Python不会给注释赋予任何含义。第三方库和框架中可以使用它们来注释涉及元编程的各种应用程序，这包括但不限于静态分析工具、文档、测试、函数重载、列集、远程过程调用、IDE、契约等。下面这个装饰器函数的例子对函数参数进行断言并返回值：

```
def ensure(func):
    # 提取注释数据
    return_check = func.__annotations__.get('return',None)
    arg_checks   = [(name,func.__annotations__.get(name))
                    for name in func.__code__.co_varnames]

    # 创建一个封装器来检查参数值并使用注释中指定的函数返回结果

    def assert_call(*args,**kwargs):
        for (name,check),value in zip(arg_checks,args):
            if check: assert check(value), "%s %s" % (name, check.__doc__)
        for name,check in arg_checks[len(args):]:
            if check: assert check(kwargs[name]), "%s %s" % (name, check.__doc__)
        result = func(*args,**kwargs)
        assert return_check(result), "return %s" % return_check.__doc__
        return result

    return assert_call
```

下面这段代码使用了上面的装饰器：

```
def positive(x):
    "must be positive"
    return x > 0

def negative(x):
    "must be negative"
    return x < 0

@ensure
def foo(a:positive, b:negative) -> positive:
    return a - b
```

以下是使用该函数的一些示例输出：

```
>>> foo(3,-2)


5
>>> foo(-5,2)


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "meta.py", line 19, in call
    def assert_call(*args,**kwargs):
AssertionError: a must be positive
>>>
```

A.2.7 只能通过关键字引用的参数

函数可以指定只能通过关键字引用的参数，方法是在第一个带*号的参数后定义额外的参数，例如：

```
def foo(x, *args, strict=False):
    statements
```

调用此函数时，只能以关键字的形式指定strict 参数，例如：

```
a = foo(1, strict=True)
```

所有额外的位置参数都只能放在args 中，不能用于设置strict 的值。如果不想接受可变数量的参数，但又要使用只能通过关键字引用的参数，可以在参数列表中使用一个单独的*号，例如：

```
def foo(x, *, strict=False):
    statements
```

下面给出了一个用法示例：

```
foo(1,True)        # 失败.TypeError: foo() takes 1 positional argument
foo(1,strict=True) # Ok.
```

A.2.8 省略号表达式

Ellipsis 对象（...）可以用作表达式，因此可放在容器中或赋值给变量，例如：

```
>>> x =

 ...       # 赋值为省略号
>>> x


Ellipsis
>>> a = [1,2,...]


>>> a


[1, 2, Ellipsis]
>>> ... in a


True
>>> x is ...


True
>>>
```

对省略号的解释仍然取决于使用它的应用程序。这种特性可以将...作为库和框架中语法的一个有趣组成部分。（例如，用来表示一个通配符、续行符或一些类似符号。）

A.2.9 链接异常

现在可以将异常链接在一起。从本质上讲，这是让当前异常携带与前面异常相关信息的一种途径。from 限定符与raise 语句在一起使用便可显式地链接异常。例如：

```
try:
    statements


except ValueError as e:
    raise SyntaxError("Couldn't parse configuration") from e
```

引发SyntaxError 异常时，就会生成像下面这样一条跟踪消息，同时显示两个异常：

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'nine'
```

```
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
SyntaxError: Couldn't parse configuration
```

异常对象的__cause__属性被置为前一个异常。使用from限定符和raise语句可以设置该属性。

还有更加微妙的异常链接示例，涉及在另一个异常处理器中引发异常，例如：

```
def error(msg):
    print(m)              # 注意：这里的错误是有意的（m未定义）

try:
    statements


except ValueError as e:
    error("Couldn't parse configuration")
```

如果在Python 2中执行这段代码，在error()中只有一个与NameError相关的异常。而在Python 3中，前一个要处理的异常与结果链接在了一起。例如，用户将看到这条消息：

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'nine'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
  File "<stdin>", line 2, in error
NameError: global name 'm' is not defined
```

如果是隐式链接，异常实例e的__context__属性将包含对前一个异常的引用。

A.2.10 经过改进的super()函数

super()函数用于查找基类中的方法，在Python 3中它可以不带任何参数运行，例如：

```
class C(A,B):
    def bar(self):
        return super().bar()        # 调用基类中的bar()方法
```

而在Python 2中，必须使用super(C,self).bar()。现在仍然支持旧式语法，但它十分累赘。

A.2.11 高级元类

在Python 2中，可以定义元类来改变类的行为。关于元类的实现有一点需要注意，它进行的处理只会在类的主体执行完毕之后开始。也就是说，解释器执行类的整个主体并填充字典。一旦字典填充完毕，就会被传递给元类构造函数（在类主体执行完以后）。

在Python 3中，元类还可以在类主体执行之前完成一些额外的工作，方法是在元类中定义一个特殊的类方法`__prepare__(cls, name, bases, **kwargs)`。该方法的返回结果必须是一个字典，而类定义的主体执行时将填充这个字典。下面这个例子概要列出了基本的过程：

```
class MyMeta(type):
    @classmethod
    def  __prepare__(cls,name,bases,**kwargs):
        print("preparing",name,bases,kwargs)
        return {}
    def  __new__(cls,name,bases,classdict):
         print("creating",name,bases,classdict)
         return type.__new__(cls,name,bases,classdict)
```

Python 3使用另一种语法来指定元类。例如，使用以下代码可以定义一个使用`MyMeta`的类：

```
class Foo(metaclass=MyMeta):
    print("About to define methods")
    def __init__(self):
        pass
    def bar(self):
        pass
    print("Done defining methods")
```

如果运行上述代码，在输出中会看到控制流的走向：

```
preparing Foo () {}
About to define methods
Done defining methods
creating Foo (){'__module__': '__main__',
               'bar':<function bar at 0x3845d0>,
               '__init__': <function __init__ at 0x384588>}
```

元类的`__prepare__()`方法上的额外关键字参数是从`class`语句的基类列表中使用的关键字参数传递而来。例如，语句`class Foo(metaclass=MyMeta, spam=42, blah="Hello")`将关键字参数`spam`和`blah`传递给`MyMeta. __prepare__()`方法。这种约定可以将任意配置信息传递给元类。

如果要通过元类的新方法`__prepare__()`执行有用的处理，通常需要让方法返回一个自定义的字典对象。例如，如果要在定义类时执行特殊处理，定义一个从`dict`继承而来的类，然后重新实现`__setitem__()`方法以捕捉给类字典的赋值。下面的例子说明了

这一点，它定义了一个元类，让它在重复定义方法或类变量时报告错误。

```
class MultipleDef(dict):
    def __init__(self):
        self.multiple= set()
    def __setitem__(self,name,value):
        if name in self:
            self.multiple.add(name)
        dict.__setitem__(self,name,value)

class MultiMeta(type):
    @classmethod
    def __prepare__(cls,name,bases,**kwargs):
         return MultipleDef()
    def __new__(cls,name,bases,classdict):
         for name in classdict.multiple:
            print(name,"multiply defined")
        if classdict.multiple:
           raise TypeError("Multiple definitions exist")
        return type.__new__(cls,name,bases,classdict)
```

如果将这个元类应用给另一个类定义，它将在重复定义方法时报告错误，例如：

```
class Foo(metaclass=MultiMeta):
    def __init__(self):
        pass
    def __init__(self,x):              # Error. __init__ multiply defined.
        pass
```

A.3 常见陷阱

如果要从Python 2迁移到Python 3，必须清楚Python 3不仅仅只是拥有新的语法和语言特性。核心语言和库的主要部分已经发生了微妙的变化。对于Python 2程序员而言，Python 3的有些方面就如同bug一般。在其他方面，过去在Python 2中很“容易的”事情现在已经被禁止。

本节概述了Python 2程序员在进行迁移的过程中，可能会遇到的一些主要陷阱。

A.3.1 文本与字节

Python 3对文本字符串（字符）和二进制数据（字节）进行了严格区分。像`"hello"`这样的字面量表示一个以Unicode编码保存的文本字符串，而`b"hello"`则表示一个字节字符串（包含ASCII字符）。

在Python 3中，无论什么情况都不能混用`str`和`bytes`类型。例如，如果要连接字符串和字节，就会引发`TypeError`异常。而在Python 2中，会根据需要将字节字符串自动转换为Unicode。

要将文本字符串转换为字节，必须使用 *s* `.encode(encoding)`方法。例

如，`s.encode('utf-8')` 可将 *s* 转换为一个UTF-8编码的字节字符串。要将字节字符串`t` 转换回为文本，必须使用 *t* `.decode (encoding)` 方法。可将`encode()` 和`decode()` 方法视为字符串和字节之间的一种“类型强制转换”。

从根本上讲，严格区分文本和字节是一件好事，因为Python 2中混用字符串类型的规则不够清晰，难于理解。但是，Python 3这样做的一个后果是，字节字符串在实际应用中模仿“文本”的能力受到限制。尽管有像`split()` 和`replace()` 这样的标准字符串方法，但字节字符串的其他方面与Python 2中并不相同。例如，要打印某个字节字符串，只会获得带有引号的`repr()` 输出，如`b'`*contents* `'` 。类似地，所有的字符串格式化操作（`%`，`.format()`）也都不能用。例如：

```
x = b'Hello World'
print(x)                      # 生成 b'Hello World'
print(b"You said '%s'" % x)   # TypeError: % operator not supported
```

对于系统程序员而言，字节缺少类似文本的行为是一个潜在陷阱。抛开Unicode的影响不谈，在很多情况下，用户实际上想要处理和操作面向字节的数据，如ASCII。用户可能倾向于使用`bytes` 类型，以避免Unicode带来的开销和复杂性。但是，这实际上会让与面向字节的文本处理有关的一切变得更加困难。下面这个例子说明了这个潜在的问题：

```
>>> # 使用字符串创建响应消息（Unicode）

>>> status = 200

>>> msg = "OK"

>>> proto = "HTTP/1.0"

>>> response = "%s %d %s" % (proto, status, msg)

>>> print(response)

HTTP/1.0 200 OK
>>> # 只使用字节创建响应消息（ASCII）

>>> status = 200

>>> msg = b"OK"

>>> proto = b"HTTP/1.0"

>>> response = b"%s %d %s" % (proto, status, msg)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'

>>> response = proto + b" " + str(status) + b" " + msg


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str

>>> bytes(status)


b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00....'

>>> bytes(str(status))


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string argument without an encoding

>>> bytes(str(status),'ascii')


b'200'

>>> response = proto + b" " + bytes(str(status),'ascii') + b" " + msg


>>> print(response)


b'HTTP/1.0 200 OK'

>>> print(response.decode('ascii'))


HTTP/1.0 200 OK
>>>
```

在这个例子中，可以看到Python 3是如何严格区分文本/字节的。即便是看上去应该很简单的操作，例如将一个整数转换为ASCII字符，对字节来说也要复杂得多。

这里的要点是，如果要执行基于文本的处理或格式化，使用标准的文本字符串很可能会更好。如果需要在完成处理之后获得一个字节字符串，可以使用`s.encode('latin-1')`从Unicode进行转换。

处理各种库模块时，文本/字节的区分会显得更加微妙一些。有些库对于文本或字节同样有效，而有些库却完全禁止使用字节。在其他情况下，模块行为会根据接收到的输入类别的不同而有所不同。例如，如果`dirname`是一个字符串，`os.listdir(dirname)`函数只返回能够成功解码为Unicode的文件名。如果`dirname`是一个字节字符串，则会以字

节字符串的形式返回所有文件名。

A.3.2 新的I/O系统

Python 3实现了一个全新的I/O系统，第19章的`io`模块一节介绍了关于它的详细情况。新的I/O系统也反映了文本和字符串形式的二进制数据之间的巨大差异。

如果要对文本执行任何I/O操作，Python 3会强制用户使用“文本模式”打开文件，如果不想使用默认编码（通常为UTF-8），还需提供可选的编码方式。如果对二进制数据执行I/O操作，必须使用“二进制模式”打开文件，并且使用字节字符串。常见的错误源头是将输出数据传递给以错误模式打开的文件或I/O流。例如：

```
>>> f = open("foo.txt","wb")


>>> f.write("Hello World\n")


Traceback (most recent call last):


File "<stdin>", line 1, in <module>


File "/tmp/lib/python3.0/io.py", line 1035, in write


raise TypeError("can't write str to binary stream")
TypeError: can't write str to binary stream
>>>
```

套接字、管道和其他类别的I/O通道应该始终假定为使用二进制模式。特别是网络代码有一个潜在的问题，即很多网络协议都涉及基于文本的请求/响应处理（如HTTP、SMTP、FTP等）。如果套接字是二进制，这种混合使用二进制I/O和文本处理的做法可能导致混用文本和字节相关的问题，这一点在前面一节中提到过，应该小心对待。

A.3.3 `print()`和`exec()`函数

Python 3中的`print`和`exec`语句已经变成了函数。`print()`函数的用法与Python 2中的用法比较如下所示：

```
print(x,y,z)          # 同print x, y, z
print(x,y,z,end=' ')   # 同print x, y, z,
print(a,file=f)        # 同print >>f, a
```

因为`print()`现在是一个函数，所以需要时可以使用其他的定义来替换它。

`exec()`现在也是一个函数，但它在Python 3中的行为与Python 2中略有不同。例如，

考虑以下代码：

```
def foo():
    exec("a = 42")
    print(a)
```

在Python 2中，调用foo() 函数将打印结果数字'42' 。而在Python 3中将会引发一个NameError 异常，因为变量a 没有定义。原因在于exec() 作为函数，只操作globals() 和locals() 函数返回的字典。但locals() 函数返回的字典实际上是局部变量的一份副本 。exec() 函数中进行的赋值只修改了局部变量的这份副本，而非局部变量本身。下面给出了一种解决方法：

```
def foo():
    _locals = locals()
    exec("a = 42",globals(),_locals)
    a = _locals['a']      # 提取已设置的变量内容
    print(a)
```

一般而言，不要期望Python 3能够像Python 2中那样支持使用exec() 、eval() 和exefile() 函数。事实上，execfile() 函数已经彻底消失。（将一个表示已打开文件的对象传递给exec() 函数就可以模拟其功能。）

A.3.4 使用迭代器和视图

相对Python 2而言，Python 3更好地利用了迭代器和生成器。像zip() 、map() 和range() 这样过去返回列表的内置函数现在返回的是可迭代对象。如果需要将结果转换为列表，请使用list() 函数。

Python 3从字典提取键和值信息方面的做法略有不同。在Python 2中，可以分别使用诸如d.keys() 、d.values() 或d.items() 之类的方法获得键、值或键/值对的列表。而在Python 3中，这些方法返回的是所谓的视图对象，例如：

```
>>> s = { 'GOOG': 490.10, 'AAPL': 123.45, 'IBM': 91.10 }


>>> k = s.keys()


>>> k


<dict_keys object at 0x33d950>
>>> v = s.values()


>>> v


<dict_values object at 0x33d960>
>>>
```

这些对象支持迭代，因此如果要查看它们的内容，可以使用for 循环，例如：

```
>>> for x in k:

...     print(x)

...
GOOG
AAPL
IBM
>>>
```

视图对象始终关联到用于创建它们的字典。有趣的是，如果底层字典出现变化，视图生成的项也会随之变化，例如：

```
>>> s['ACME'] = 5612.25

>>> for x in k:

...     print(x)

...
GOOG
AAPL
IBM
ACME
>>>
```

如果需要构建字典键或值的列表，只要使用list() 函数即可，例如 list(s.keys()) 。

A.3.5 整数与整数除法

Python 3不再分别使用int 和long 类型表示32位整数和长整数。现在，int 类型表示任意精度的整数（其中的内部细节并未公开给用户）。

另外，整数除法现在生成的结果始终是浮点数。例如，3/5 的结果是0.6 ，而不是0 。即使结果正好是整数，也会被转换为浮点数。例如，8/2 的结果是4.0 ，而不是4 。

A.3.6 比较

Python 3对于值的比较要严格得多。在Python 2中，任意两个对象均可进行比较，即使这没有任何意义也可以，例如：

```
>>> 3 < "Hello"


True
>>>
```

在Python 3中，这种比较将导致TypeError 异常，例如：

```
>>> 3 < "Hello"


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
>>>
```

这种变化可谓微不足道，但这意味着在Python 3中，需要更加注意数据的类型是否正确。例如，如果使用列表的sort() 方法，列表中的所有项都必须与< 运算符兼容，否则就会出现错误。在Python 2中，这类操作将会悄悄执行，并返回一个通常是毫无意义的结果。

A.3.7 迭代器与生成器

Python 3对迭代器协议略作了修改。原来迭代时要调用__iter__() 和next() 方法，现在next() 方法已经更名为__next__() 。这种改动对大多数用户没有影响，除非已经写好手动迭代可迭代对象的代码或者已经自定义了迭代器对象。用户一定要修改类中next() 方法的名称。使用内置的next() 函数来调用迭代器的next() 或__next__() 方法这样代码的可移植性强。

A.3.8 文件名、参数与环境变量

在Python 3中，根据地区设置的不同，文件名、sys.argv 中的命令行参数和os.environ 中的环境变量不一定会被当作Unicode处理。唯一的问题在于，操作系统环境并非全部使用了Unicode。例如在很多系统上，使用与有效Unicdoe编码不对应的原始字节序列来指定文件名、命令行选项和环境变量在技术上讲是可行的。尽管这些情形在实际中很少见，在使用Python编程执行系统管理相关任务时还是要引起注意。正如前面提到的那样，以字节字符串的形式提供文件和目录名可以解决很多问题，例如os.listdir(b'/foo') 。

A.3.9 库的重新组织

Python 3重新组织并修改了标准库中某些部分的名称，其中最明显的是与网络和网络

数据格式相关的常用模块。另外，各种遗留模块已经从库中删除，如gopherlib、rfc822 等。

使用小写形式的模块名现在已经成为标准实践。ConfigParser 、Queue 和SocketServer 等模块已经各自更名为configparser 、queue 和socketserver 。用户应该尝试在自己的代码中遵循类似的约定。

Python 3创建了一些包，用于重新组织以前位于不同模块中的代码，如包含用于写HTTP服务器的所有模块的http 包，包含HTML解析模块的html 包，以及包含XML-RPC模块的xmlrpc 包，诸如此类。

对于已废弃的模块，本书经过仔细斟酌，只讲述了目前Python 2.6和Python 3中正在使用的模块。如果在现有的Python 2代码中发现这里没有提到的模块，它很有可能已经被更新的模块所替代。举个例子，Python 3没有Python 2中常用于启动子进程的popen2 模块，现在使用的是subprocess 模块。

A.3.10 绝对导入

由于库重新组织的关系，一个包的子模块中出现的所有import 语句都要使用绝对名称。第8章详细介绍了这一点，但前提是包的组织要像下面这样：

```
foo/
    __init__.py
    spam.py
    bar.py
```

如果文件spam.py 使用语句import bar ，就会引发ImportError 异常，即便bar.py 文件位于同一目录中也是如此。要加载该子模块，spam.py 需要使用import foo.bar 语句或者使用像from . import bar 这样的包相对导入。

这与Python 2有所区别，在继续检查sys.path 中的其他目录之前，Python 2中的import 始终会先在当前目录中寻求匹配。

A.4 代码迁移与2to3

将代码从Python 2转换到Python 3是一个较难处理的主题。一定要清楚，没有任何神奇的导入、标志、环境变量或工具能让Python 3运行所有Python 2程序。但可以通过一些专门的步骤来迁移代码，我们现在就开始介绍它们。

A.4.1 将代码移植到Python 2.6

建议任何要将代码移植到Python 3的用户首先将代码移植到Python 2.6。Python 2.6不仅与Python 2.5向后兼容，而且支持Python 3中的部分新特性，如高级字符串格式化、新的异常语法、字节字面量、I/O库和抽象基类。因此，即使还不能完全迁移，Python 2程序也可以开始利用有用的Python 3特性。

迁移到Python 2.6的另一个理由是，如果使用-3 命令行选项运行Python 2.6，它会对已废弃的特性发出警告消息，例如：

```
bash-3.2$ python -3
Python 2.6 (trunk:66714:66715M, Oct 1 2008, 18:36:04)
[GCC 4.0.1 (Apple Computer, Inc. build 5370)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = { }
>>> a.has_key('foo')
__main__:1: DeprecationWarning: dict.has_key() not supported in 3.x; use the in
operator
False
>>
```

用户可以参考这些警告消息，在迁移到Python 3之前，尽力保证程序在Python 2.6上运行时不会出现任何警告。

A.4.2 提供测试覆盖

Python有一些很有用的测试模块，包括doctest 和unittest 。确保在尝试移植到Python 3之前，对应用程序进行全面测试。如果程序此时还未经任何测试，现在正是开始的好时机。要确保测试的覆盖范围尽可能大，而且程序在Python 2.6上运行时，能通过测试并且没有出现任何警告信息。

A.4.3 使用2to3 工具

Python 3中包含一个名为2to3 的工具，可以协助从Python 2.6到Python 3的代码迁移工作。该工具通常位于Python源代码发布版的Tools/scripts 目录中，而且在大多数系统上也与python 3.0二进制文件安装在同一目录中。它是一个命令行工具，一般可以从UNIX或Windows命令行运行。

例如，考虑以下包含大量已废弃特性的程序：

```
# example.py
import ConfigParser

for i in xrange(10):
    print i, 2*i

def spam(d):
    if not d.has_key("spam"):
        d["spam"] = load_spam()
    return d["spam"]
```

要对这个程序运行2to3 工具，输入2to3 example.py 即可，例如：

```
%

 2to3 example.py
```

```
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- example.py (original)
+++ example.py (refactored)
@@ -1,10 +1,10 @@
#example.py
-import ConfigParser
+import configparser

-for i in xrange(10):
-    print i, 2*i
+for i in range(10):
+    print(i, 2*i)

  def spam(d):
-     if not d.has_key("spam"):
+     if "spam" not in d:
          d["spam"] = load_spam()
      return d["spam"]
RefactoringTool: Files that need to be modified:
RefactoringTool: example.py
```

2to3 工具将输出有问题的和可能需要修改的程序部分。这些内容在输出中显示为环境差异。我们已经对单个文件使用过2to3 工具，但如果给它提供一个目录名，它将循环查找该目录结构中包含的所有Python文件，然后生成统一的报告。

默认情况下，2to3 工具不能实际修复它扫描的任何源代码，它只是报告出可能需要修改的代码部分。2to3 工具面临的难题是，它提供的信息经常不完整。例如，考虑示例代码中的spam() 函数，它调用了一个方法d.has_key() 。在字典操作中，has_key() 方法已经被in 运算符所取代。2to3 工具报告了这个变化，但没有给出更多信息，用户无从知道spam() 函数是否在操作字典。情况可能是这样，d 是某种恰好提供了has_key() 方法的其他对象（或许是一个数据库），但如果使用in 运算符代替它就会失败。2to3 工具的另一个问题出现在对字节字符串和Unicode的处理上。由于Python 2将自动把字节字符串转换为Unicode，所以不小心混合使用这两种字符串类型是很常见的情况。不幸的是，2to3 工具不能将这类情况全部查找出来。这也是要尽可能扩大单元测试范围的原因。当然，所有这些问题都取决于应用程序本身。

用户可以指导2to3 工具来修复选中的不兼容性问题。首先，输入2to3 -l 就能看到一个“修复器”的列表，例如：

```
% 2to3 -l

Available transformations for the -f/--fix option:
apply
basestring
buffer
```

```
callable
...
...
xrange
xreadlines
zip
```

如果使用这个列表中的名称，只要输入2to3 –f fixname filename 就能看到实际的修复效果。如果要应用多个修复，只要使用单独的-f 选项来指定每个修复即可。如果要将修复应用给某个源文件，需要添加-w 选项，如2to3 –f fixname –w filename 。例如：

```
% 2to3 -f xrange -w example.py


--- example.py (original)
+++ example.py (refactored)
@@ -1,7 +1,7 @@
 # example.py
 import ConfigParser

-for i in xrange(10):
+for i in range(10):
     print i, 2*i

 def spam(d):
RefactoringTool: Files that were modified:
RefactoringTool: example.py
```

如果执行该操作后查看example.py 文件，就会发现xrange() 函数已经变为range()，除此之外再无变化。此外，还会给原始的example.py 文件生成一个备份文件example.py.bak 。

-f 选项对应的是-x 选项。如果使用2to3 –x fixname filename，它将运行除使用-x 选项列出的修复器之外的所有修复器。

尽管可以指导2to3 修复所有内容和改写所有文件，但在实际应用中应该避免这样做。要记住，代码转换并不能确保精确，2to3 并不是总能“做出正确的处理”。通过系统计算的方式处理代码迁移始终会有更好的效果，而不是祈求上帝，让它能够神奇地自行完成。

2to3 工具还有一些额外的有用选项。使用-v 选项可以启用Verbose 模式，从而打印出很多额外信息以便进行调试。使用-p 选项可以告诉2to3 工具代码中已经使用print 语句作为函数，不应再对它进行转换（使用from __futrue import print_statement 语句可启用）。

A.4.4 实用的移植策略

下面介绍了一种将Python 2代码移植到Python 3的实用策略。再次提醒，最好是通过系统性的方法处理迁移，而不是妄图一次性到位。

（1）确保代码经过了充分的单元测试，而且所有测试在Python 2下都已通过。

（2）将代码和测试组件移植到Python 2.6，并确保所有测试仍然能够通过。

（3）打开Python 2.6的`-3` 选项。处理掉所有警告信息，并确保程序在没有任何警告信息出现的情况下运行并通过测试。如果正确地完成了这一步，代码在Python 2.5甚至更高版本中仍然正常运行的机会将大增。这个步骤的目的是清理掉程序中所有不合适的内容。

（4）备份代码（这个步骤自不必说）。

（5）将单元测试组件移植到Python 3中，并确保测试环境本身的正常运转。因为尚未移植任何代码，单独运行单元测试本身将会失败。但是，编写正确的测试组件应该能够处理测试失败的情况，同时测试软件本身内部不会崩溃。

（6）使用`2to3` 工具将程序本身转换为Python 3。对得到的代码运行单元测试组件，并修复出现的所有问题。完成这个步骤的策略多种多样。如果你觉得自己运气足够好，可以让`2to3` 工具修复所有内容，然后看看会发生什么。如果你足够谨慎，可能会先让`2to3` 工具修复十分明显的地方（`print`，`except` 语句，`xrange()`，库模块名称等），然后再逐个处理剩下的问题。

以上处理过程结束时，代码应该通过了所有单元测试，并且能够和以前一样地运行。

从理论上说，可以构造出既能够运行在Python 2.6中，又能够在无需人工干预的情况下自动转换为Python 3的代码。但这需要非常严格地遵守现代Python编码约定——最起码绝对需要确保在Python 2.6中不出现任何警告。如果自动转换过程需要准确地使用`2to3` 工具（如只运行选中的修复器），应该编写一个shell脚本来自动执行所需的操作，而不是要求用户自己运行`2to3` 工具。

A.4.5 同时支持Python 2和Python 3

关于Python 3迁移的最后一个问题是，是否存在不用修改便可同时支持Python 2和Python 3的单个代码库。尽管这在某些情况下是可以的，但结果代码有可能变得很乱。例如，必须避免所有`print` 语句，并确保所有`except` 子句决不使用任何异常值（而是从`sys.exc_info()` 中提取它们）。有的Python特性则根本无法使用。例如，由于语法差异，Python 2和Python 3中对元类的使用不可能相互兼容。

因此，如果要求代码必须同时运行在Python 2和3上，最好的选择是确保代码尽可能干净，能够运行在Python 2.6下并通过一套单元测试，并开发一组`2to3` 修复来进行自动转换。

对于单元测试来说，只维持单一代码库是可行的。`2to3` 工具转换完毕后，如果要检查应用程序的行为是否正确，不经修改便可在Python 2.6和Python 3上运行的测试组件就能够派上用场。

A.4.6 参与

作为一个开源项目，Python的继续发展离不开用户的贡献。特别是Python 3，报告bug、性能问题和其他问题是至关重要的。如果您要报告bug，请访问http://bugs.python.org。不要不好意思——您的反馈将会使Python日臻完美。

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书

同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在  里填入可使用的积分数值，即可扣减相应金额。

> 特别优惠
>
> 购买本电子书的读者专享异步社区优惠券 。 使用方法：注册成为社区用户，在下单购书时输入**“57AWG ”**，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

异步社区

微信订阅号

微信服务号

官方微博

QQ群：368449889

社区网址： www.epubit.com.cn

官方微信： 异步社区

官方微博： @人邮异步社区，@人民邮电出版社-信息技术分社

投稿**&**咨询： contact@epubit.com.cn