Python 教程

Python 是一门简单易学又功能强大的编程语言。它具有高效的高级数据结构和简单而有效的面向对象编程的特性。Python 优雅的语法和动态类型、 以及其解释性的性质，使它在许多领域和大多数平台成为脚本编写和快速应用程序开发的理想语言。

从 Python 网站 https://www.python.org/可以免费获得所有主要平台的源代码或二进制形式的 Python 解释器和广泛的标准库，并且可以自由地分发。网站还包含许多免费的第三方 Python 模块、 程序、工具以及附加文档的发布包和链接。

Python 解释器可以用 C 或 C+ + （或从 C 中可以调用的其他语言）轻松的对新的函数和数据类型进行扩展 Python 也适合作为可定制应用程序的一种扩展语言。

本教程通俗地向读者介绍 Python 语言及其体系的基本概念和功能。 随手使用 Python 解释器来亲自动手实践是很有帮助的，并且由于所有示例都是自成体系的，所以本教程也可以离线阅读。

有关标准对象和模块的说明，请参阅 Python 标准库。 Python 语言参考 给出了 Python 语言的更正式的定义。要编写 C 或 C+ + 的扩展，请阅读扩展和嵌入 Python 解释器 与 Python/C API 参考手册。也有几本书深度地介绍了 Python 。

本教程不会尝试全面地涵盖每一个单独特性，甚至即使它是常用的特性。相反，它介绍了许多 Python 的值得注意的特性，从而能让你很好的把握这门语言的特性。经过学习，你将能够阅读和编写 Python 的模块和程序，并可以更好的学会 Python 标准库 中描述的各种 Python 库模块。

词汇表 也值得浏览一下。

1. 开胃菜

如果你要用计算机做很多工作，后来你会发现有些任务你希望它是自动完成的。例如，你可能希望对大量的文本的文件执行搜索和替换，或以复杂的方式重命名并重新排列一堆照片文件。也许你想写一个小的自定义数据库，或一个专门的 GUI 应用程序或一个简单的游戏。

如果你是一个专业的软件开发人员，您可能必须使用几个 C / C + + /Java 库，但发现通常的编写/编译/测试/重新编译周期太慢。也许你要写这样的库中的测试套件，然后发现编写测试代码是很乏味的工作。或也许您编写了一个程序，它可以使用一种扩展语言，但你不想为您的应用程序来设计与实现一个完整的新语言。

Python 正是这样为你准备的语言。

你可以为其中一些任务写一个 Unix shell 脚本或 Windows 批处理文件，但是 shell 脚本最适合处理文件移动和文本编辑，而不适用于 GUI 应用程序和游戏。你可以用 C / C + +/Java 写一个 程序，但是可能会花费大量的开发时间去完成一份初稿。Python 更简单易用， 可用于 Windows、 Mac OS X 和 Unix 操作系统，并将帮助您更快地完成工作。

Python 使用很简单，但它是一个真正的编程语言，对于编写大型程序，Python 比 shell 脚本或批处理文件提供更多的结构和支持。另一方面，Python 还提供了比 C 更多的错误检查，并且，作为一种高级语言，它有内置的高级数据类型，比如灵活的数组和字典。因为其丰富的更加通用的数据类型， Python 的适用领域比 Awk 甚至 Perl 要广泛得多，而且很多事情在 Python 中至少和那些语言一样容易。

Python 允许您将您的程序拆分成可以在其他 Python 程序中再次使用的模块。它拥有大量的标准模块，你可以将其用作你的程序的基础 — 或者作为学习 Python 编程的示例。这些模块提供诸如文件 I/O、 系统调用、 套接字，甚至还为像 Tk 这样的图形界面开发包提供了接口。

Python 是一门解释性的语言，因为没有编译和链接，它可以节省你程序开发过程中的大量时间。Python 解释器可以交互地使用，这使得试验 Python 语言的特性、编写用后即扔的程序或在自底向上的程序开发中测试功能非常容易。它也是一个方便的桌面计算器。

Python 使程序编写起来能够简洁易读。编写的 Python 程序通常比等价的 C、 C + + 或 Java 程序短很多，原因有几个：

高级数据类型允许您在单个语句中来表达复杂的操作；
语句分组是通过缩进，而不是开始和结束的括号 ；
变量和参数的声明不是必须的

Python 是可扩展的: 如果您知道如何用 C 编程，那么将很容易添加一个新的内置函数或模块到解释器中，这样做要么是为了以最快的速度执行关键的操作，要么是为了将 Python 程序与只有二进制形式的库（如特定供应商提供的图形库）链接起来。一旦你真的着迷，你可以把 Python 解释器链接到 C 编写的应用程序中，并把它当作那个程序的扩展或命令行语

言。

顺便说一句，Python 语言的名字来自于 BBC 的 “Monty Python’s Flying Circus” 节目，与爬行动物无关。我们允许甚至鼓励在文档中引用 Monty Python 短剧！

既然现在你们都为 Python 感到兴奋，你们一定会想更加详细地研究它。学习一门语言最好的方法就是使用它，本教程推荐你边读边使用 Python 解释器练习。

在下一章中，我们将解释 Python 解释器的用法。这是很简单的一件事情，但它有助于试验后面的例子

本教程的其余部分通过实例介绍了 Python 语言和体系的各种特性，以简单的表达式、 语句和数据类型开始，然后是 函数和模块，最后讲述高级概念，如异常和用户自定义的类。

2. Python 解释器
2.1 调用解释器

Python 解释器在机器上通常安装成/usr/local/bin/python3.4；将/usr/local/bin 放在您的 Unix shell 搜索路径，使得可以通过在 shell 中键入命令

python3.4

来启动它。 [1]由于解释器放置的目录是一个安装选项，其他地方也是可能的；请与您的 Python 专家或系统管理员联系。(例如， /usr/local/python 是一个常见的替代位置。

在 Windows 机器上，Python 的安装通常是放置在 C:\Python3，当然你可以在运行安装程序时进行更改。你可以在一个 DOS 窗口的命令提示符下键入以下命令来把这个目录添加到路径中：

set path=%path%;C:\python34

主提示符下键入文件结束字符 （ Unix 上是 Control-D 、 Windows 上是 Control-Z） 会导致该解释器以零退出状态退出。如果无法正常工作，您可以通过键入以下命令退出解释器：quit()。

编辑器的行编辑功能包括交互式编辑，历史记录和代码补全，其中代码补全功能需要系统支持 readline 库。也许检查是否支持命令行编辑的最快方法就是在您遇到的第一个 Python 提示符下敲击 Control-P。如果它发出蜂鸣声，则证明支持命令行编辑 ；请参阅附录交互式输入编辑和历史替代 的有关快捷键的介绍。如果什么都没发生，或者显示^ P，命令行编辑不可用 ；你就只能够使用退格键删除当前行中的字符。

解释器有些像 Unix shell :当使用 tty 设备作为标准输入调用时，它交互地读取并执行命令 ；当用文件名参数或文件作为标准输入调用，它将读取并执行该文件中的脚本。

第二种启动解释器的方式是 python -c command [arg] ...，这种方式执行在 command 中的语句，类似于 shell 的-c 选项中。因为 Python 语句经常包含空格或其他 shell 特殊字符，通常建议把 command 部分全部放在单引号里。

有些 Python 模块同时也是可执行的脚本。这些模块可以使用 python -m module [arg] ..... 直接调用，这和在命令行输入完整的路径名执行模块 的源文件是一样的。

当使用一个脚本文件时，运行该脚本之后进入交互模式，有时是很有用的。这可以通过在脚本前面加上-i 选项实现。

有关所有命令行选项的说明请参阅 命令行与环境.
2.1.1. 参数传递

调用解释器时，脚本名称和其他参数被转换成一个字符串列表并赋值给 sys 模块中的 argv

变量。你可以通过 import sys 访问此列表。列表的长度是至少是 1 ；如果没有给出脚本和参数， sys.argv[0]是一个空字符串。当使用-c 命令时， sys.argv[0]设置为'-c'。当使用-m 模块参数时， sys.argv[0]被设定为指定模块的全名。-c 选项或-m 选项后面的选项不会被 Python 解释器处理，但是会被保存在 sys.argv 中，供命令或模块使用。

2.1.2. 交互模式

当从 tty 读取命令时，我们说解释器在交互模式 下。这种模式下，解释器以主提示符 提示输入命令，主提示符通常标识为三个大于号(>>>) ；如果有续行，解释器以从提示符 提示输入，默认为三个点 (...)。在第一个提示符之前，解释器会打印出一条欢迎信息声明它的版本号和授权公告：

```
$ python3.4
Python 3.4 (default, Mar 16 2014, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

输入多行结构时需要续行。作为一个例子，看看这个 if 语句：

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...         print("Be careful not to fall off!")
...
Be careful not to fall off!
```

更多交互模式内容，请看 Interactive Mode.

2.2. 解释器及其环境

2.2.1. 源程序的编码

Python 源文件默认以 UTF-8 编码。在这种编码下，世界上大多数语言的字符可以在字符串，标识符和注释中同时使用 — 尽管标准库中的标识符只使用 ASCII 字符，它是可移植代码应该遵循的一个惯例。为了能够正确显示所有的这些字符，你的编辑器必须能够识别文件是 UTF-8 编码，且必须使用支持文件中所有字符的字体。

也可以给源文件指定一个不同的编码。方法是在 #! 行的后面再增加一行特殊的注释来定义源文件的编码：

```
# -*- coding: encoding -*-
```

通过此声明，源文件中的所有字符将被视为由 encoding 而不是 UTF-8 编码。在 Python 库参考手册的 codecs 小节中，可以找到所有可能的编码方式列表。

例如，如果你选择的编辑器不支持 UTF-8 编码的文件，而只能用其它编码比如 Windows-1252，你可以这样写：

```
# -*- coding: cp-1252 -*-

currency = u"€"
print ord(currency)
```

然后源文件中的所有字符仍然使用 Windows-1252 字符集。这个特殊的编码注释必须位于文件的第一或者第二 行。

脚注

[1] 在 Unix 系统中，为了避免与系统默认安装的 Python 2.x 可执行文件相冲突，Python 3.x 默认安装的文件中并不包含名为 python 的可执行文件。

3. Python 简介

以下的示例中，输入和输出通过提示符（>>>和...）是否出现加以区分：如果要重复该示例，你必须在提示符出现后，输入提示符后面的所有内容；没有以提示符开头的行是解释器的输出。注意示例中出现从提示符意味着你一定要在最后加上一个空行；这用于结束一个多行命令。

本手册中的很多示例，甚至在交互方式下输入的示例，都带有注释。Python 中的注释以“井号”， #，开始，直至实际的行尾。注释可以从行首开始，也可以跟在空白或代码之后，但不能包含在字符串字面量中。字符串字面量中的#字符仅仅表示#。因为注释只是为了解释代码并且不会被 Python 解释器解释，所以敲入示例的时候可以忽略它们。

例如：

```
# this is the first comment
spam = 1    # and this is the second comment
            # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1. 将 Python 当做计算器

让我们尝试一些简单的 Python 命令。启动解释器然后等待主提示符 >>>。（应该不需要很久。）

3.1.1. 数字

解释器可作为一个简单的计算器：你可以向它输入一个表达式，它将返回其结果。表达式语法很直白：运算符+、 -、 *和/的用法就和其它大部分语言一样（例如 Pascal 或 C）；括号(())可以用来分组。例如：

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5    # division always returns a floating point number
1.6
```

整数(例如 2, 4, 20)的类型是 int，带有小数部分数字(e.g. 5.0, 1.6)的类型是 float。在本教程的后面我们会看到更多关于数字类型的内容。

除法(/)永远返回一个浮点数。如要使用 floor 除法并且得到整数结果（丢掉任何小数部分），你可以使用//运算符；要计算余数你可以使用%：

```
>>> 17 / 3   # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3   # floor division discards the fractional part
5
>>> 17 % 3   # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2   # result * divisor + remainder
17
```

通过 Python，还可以使用**运算符符计算幂乘方[1]：

```
>>> 5 ** 2   # 5 squared
25
>>> 2 ** 7   # 2 to the power of 7
128
```

等号（=）用于给变量赋值。赋值之后，在下一个提示符之前不会有任何结果显示：

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

如果变量没有“定义”（赋值），使用的时候将会报错：

```
>>> n   # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

浮点数完全支持；整数和浮点数的混合计算中，整数会被转换为浮点数：

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

在交互模式下，最近一次表达式的值被赋给变量_。这意味着把 Python 当做桌面计算器使用的时候，可以方便的进行连续计算，例如：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
```

```
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

用户应该将这个变量视为只读的。不要试图去给它赋值 — — 你将会创建出一个独立的同名局部变量，并且屏蔽了内置变量的魔术效果。

除了 int 和 float，Python 还支持其它数字类型，例如小数 and 分数。Python 还内建支持复数，使用后缀 j 或 J 表示虚数部分（例如 3+5j）。

3.1.2. 字符串

除了数值，Python 还可以操作字符串，可以用几种方法来表示。它们可以用单引号('...')或双引号("...")括起来，效果是一样的[2]。\ 可以用来转义引号。

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

在交互式解释器中，输出的字符串会用引号引起来，特殊字符会用反斜杠转义。虽然可能和输入看上去不太一样（括起来的引号可能会变），但是两个字符串是相等的。如果字符串中只有单引号而没有双引号，就用双引号引用，否则用单引号引用。print()函数生成可读性更好的输出，它会省去引号并且打印出转义后的特殊字符：

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

如果你前面带有\的字符被当作特殊字符，你可以使用原始字符串，方法是在第一个引号前面加上一个 r:

```
>>> print('C:\some\name')   # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')   # note the r before the quote
C:\some\name
```

字符串可以跨多行。一种方法是使用三引号："""..."""或者'''...'''。行尾换行符会被自动包含到字符串中，但是可以在行尾加上 \ 来避免这个行为。下面的示例：

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

将生成以下输出（注意，没有开始的第一行）：

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

字符串可以用 +操作符联接，也可以用* 操作符重复多次：

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

相邻的两个或多个字符串字面量（用引号引起来的）会自动连接。

```
>>> 'Py' 'thon'
'Python'
```

然而这种方式只能用于两个字符串的连接，变量或者表达式是不行的。

```
>>> prefix = 'Py'
>>> prefix 'thon'   # can't concatenate a variable and a string literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
```

SyntaxError: invalid syntax

如果你想连接多个变量或者连接一个变量和一个常量，使用+：

```
>>> prefix + 'thon'
'Python'
```

这个功能在你想切分很长的字符串的时候特别有用：

```
>>> text = ('Put several strings within parentheses '
            'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

字符串可以索引，第一个字符的索引值为 0。Python 没有单独的字符类型；字符就是长度为 1 的字符串。

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

索引也可以是负值，此时从右侧开始计数：

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

注意，因为 -0 和 0 是一样的，负的索引从 -1 开始。

除了索引，还支持切片。索引用于获得单个字符，切片让你获得子字符串：

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

注意，包含起始的字符，不包含末尾的字符。这使得 s[:i] + s[i:] 永远等于 s ：

```
>>> word[:2] + word[2:]
```

```
'Python'
>>> word[:4] + word[4:]
'Python'
```

切片的索引有非常有用的默认值；省略的第一个索引默认为零，省略的第二个索引默认为切片的字符串的大小。

```
>>> word[:2]    # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]    # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

记住切片如何工作的一种方法是把索引当做字符之间的点，第一个字符的左边是 0。含有 n 个字符的字符串的最后一个字符的右边是索引 n，例如：

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

第一行给出了字符串中 0…6 各索引的位置；第二行给出了相应的负索引。从 i 到 j 的切片由 i 和 j 之间的所有字符组成。

对于非负索引，如果上下都在边界内，切片长度就是两个索引之差。例如，word[1:3] 的长度是 2。

试图使用太大的索引会导致错误：

```
>>> word[42]    # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

但是，当用于截断时，超出范围的截断索引会优雅地处理：

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python 字符串不可以改变—它们是不可变的。因此，赋值给字符串索引的位置会导致错误：

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

如果你需要一个不同的字符串，你应该创建一个新的：

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

内置函数 len()返回字符串的长度：

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

请参阅

Text Sequence Type — str
    Strings are examples of sequence types, and support the common operations supported by such types.
String Methods
    Strings support a large number of methods for basic transformations and searching.
String Formatting
    Information about string formatting with str.format() is described here.
printf-style String Formatting
    The old formatting operations invoked when strings and Unicode strings are the left operand of the % operator are described in more detail here.

3.1.3. 列表

Python 有几个 复合 数据类型，用来组合其他的值。最有用的是 列表，可以写成中括号中的一列用逗号分隔的值。列表可以包含不同类型的元素，但是通常一个列表中的所有元素都拥有相同的类型。

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

和字符串（以及其它所有内建的 序列类型）一样，列表可以索引和切片：

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

所有的切片操作都会返回一个包含请求的元素的新列表。这意味着下面的切片操作返回列表一个新的（浅）拷贝副本。

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

列表也支持连接这样的操作：

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

与不可变的字符串不同，列表是可变的 类型，例如可以改变它们的内容：

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

你还可以使用 append() 方法（后面我们会看到更多关于方法的内容）在列表的末尾添加新的元素：

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

给切片赋值也是可以的，此操作甚至可以改变列表的大小或者清空它:：

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
```

```
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

内置函数 len()也适用于列表：

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

列表可以嵌套 （创建包含其他列表的列表），例如：

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2. 编程第一步

当然，我们可以将 Python 用于比 2 加 2 更复杂的任务。例如，我们可以写一个生成斐波那契 初始子序列的程序，如下所示：

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...         print(b)
...         a, b = b, a+b
...
1
1
2
```

3
5
8

本示例介绍了几种新功能。

第一行包括了一个多重赋值：变量 a 和 b 同时获得新的值 0 和 1。最后一行又这样使用了一次，说明等号右边的表达式在赋值之前首先被完全解析。右侧表达式是从左到右计算的。

只要条件（这里是 b < 10）为 true，while 循环反复执行。在 Python 中，和 C 一样，任何非零整数值都为 true；零为 false。循环条件也可以是一个字符串或者列表，实际上可以是任何序列；长度不为零的序列为 true，空序列为 false。示例中使用的测试是一个简单的比较。标准比较运算符与 C 的写法一样：< (小于)， > （大于)， = = (等于)， < = （小于或等于)， > = （大于或等于） 和! = （不等于)。

循环体 是缩进 的：缩进是 Python 分组语句的方式。交互式输入时，你必须为每个缩进的行输入一个 tab 或（多个）空格。实践中你会用文本编辑器来编写复杂的 Python 程序；所有差不多的文本编辑器都有自动缩进的功能。交互式输入复合语句时，最后必须跟随一个空行来表示结束（因为解析器无法猜测你什么时候已经输入最后一行)。注意基本块内的每一行必须按相同的量缩进。

print()函数输出传给它的参数的值。与仅仅输出你想输出的表达式不同（就像我们在前面计算器的例子中所做的)，它可以输出多个参数、浮点数和字符串。打印出来的字符串不包含引号，项目之间会插入一个空格，所以你可以设置漂亮的格式，像这样：

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

关键字参数 end 可以避免在输出后面的空行，或者可以指定输出后面带有一个不同的字符串：

```
>>> a, b = 0, 1
>>> while b < 1000:
...         print(b, end=',')
...         a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

脚注

[1] 因乘幂运算 ** 比取负运算 -有更高的优先级，因此-3**2 将被解释为 -(3**2) 所以结果会得到-9. 为避免这种结果，而得到想要的结果 9，你可以使用 (-3)**2.

[2] 不同于其他语言的，特殊字符如 \n 在单引号 ('...') 和双引号 ("...") 之内是一样的. 两者之间的唯一区别是单引号内不需要对双引号 " 转义(但你要转义单引号 \') 反之亦然.

4. 控制流

除了前面介绍的 while 语句，Python 也有其它语言常见的流程控制语句，但是稍有不同。

4.1. if 语句

也许最为人知的语句类型是 if 语句。例如：

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...         x = 0
...         print('Negative changed to zero')
... elif x == 0:
...         print('Zero')
... elif x == 1:
...         print('Single')
... else:
...         print('More')
...
More
```

可以有零个或多个 elif 部分，else 部分是可选的。关键字 'elif' 是 'else if' 的简写，可以有效避免过深的缩进。if... elif ... elif ... 序列用于替代其它语言中 switch 或 case 语句。

4.2. for 语句

Python 中的 for 语句和你可能熟悉的 C 或 Pascal 中的有点不同。相比于总是遍历算术级数的数字 （如在 Pascal），或使用户能够定义迭代步长和停止条件 （如 C），Python 的 for 语句循环访问项目的任何序列 （列表或一个字符串），按照它们在序列中出现的顺序。例如（没有双关意）：

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...         print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

如果要在循环内修改正在迭代的序列（例如，复制所选的项目），建议首先制作副本。迭代序列不会隐式地创建副本。 使用切片就可以很容易地做到：

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...         if len(w) > 6:
```

```
...                 words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3. range() 函数

如果你确实需要遍历一个数字序列，内置函数 range()很方便。它会生成等差序列：

```
>>> for i in range(5):
...         print(i)
...
0
1
2
3
4
```

给定的终点永远不会在生成的序列中；若要依据索引迭代序列，你可以结合使用 range()和 len()，如下所示：也可以让 range 函数从另一个数值开始，或者可以指定一个不同的步进值（甚至是负数，有时这也被称为‘步长’）：

```
range(5, 10)
    5 through 9

range(0, 10, 3)
    0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

若要依据索引迭代序列，你可以结合使用 range () 和 len()，如下所示：

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...         print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

然而，在这种情况下，大部分时候使用 enumerate()函数会更加方便，请参见 Looping

Techniques。

如果你只打印 range，会出现奇怪的结果：

```
>>> print(range(10))
range(0, 10)
```

range()返回的对象的行为在很多方面很像一个列表，但实际上它并不是列表。当你迭代它的时候它会依次返回期望序列的元素，但是它不会真正产生一个列表，因此可以节省空间。

我们把这样的对象称为可迭代的，也就是说，它们适合作为期望连续获得元素直到穷尽的函数和构造器的目标。我们已经看到 for 语句是这样的一个迭代器。list()函数是另外一个；它从可迭代对象创建列表。

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

后面我们会看到更多返回可迭代对象和以可迭代对象作为参数的函数。

4.4. break 和 continue 语句，以及循环中 else 子句

Break 语句和 C 中的类似，用于跳出最近的 for 或 while 循环。

循环语句可以有一个 else 子句；当循环是因为迭代完整个列表(for 语句)或者循环条件不成立（while 语句）终止，而非由 break 语句终止时，else 子句将被执行。下面循环搜索质数的代码例示了这一点：

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

（是的，这是正确的代码。看仔细：else 子句属于 for 循环，不属于 if 语句。）

与循环一起使用的 else 子句更类似于 try 语句的 else 子句而不是 if 语句的 else 子句：try 语句的 else 子句在没有任何异常发生时运行，而循环的 else 子句在没有 break 发生时运行。更多关于 try 语句和异常的内容，请参见处理异常。

continue 语句，也是从 C 语言借来的，表示继续下一次迭代：

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5. pass 语句

pass 语句什么也不做。它用于语法上必须要有一条语句，但程序什么也不需要做的场合。例如：

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

它通常用于创建最小的类：

```
>>> class MyEmptyClass:
...     pass
...
```

另一个使用 pass 的地方是编写新代码时作为函数体或控制体的占位符 ， 这让你在更抽象层次上思考。pass 语句将被默默地忽略：

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

4.6.定义函数

我们可以创建一个生成任意上界菲波那契数列的函数：

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

关键字 def 引入函数的 定义。其后必须跟有函数名和以括号标明的形式参数列表。组成函数体的语句从下一行开始，且必须缩进。

函数体的第一行可以是一个可选的字符串文本；此字符串是该函数的文档字符串，或称为 docstring。（更多关于 docstrings 的内容可以在 文档字符串一节中找到。）有工具使用 docstrings 自动生成在线的或可打印的文档，或者让用户在代码中交互浏览；在您编写的代码中包含 docstrings 是很好的做法，所以让它成为习惯吧。

函数的 执行 会引入一个新的符号表，用于函数的局部变量。更确切地说，函数中的所有的赋值都是将值存储在局部符号表；而变量引用首先查找局部符号表，然后是上层函数的局部符号表，然后是全局符号表，最后是内置名字表。因此，在函数内部全局变量不能直接赋值（除非用 global 语句命名），虽然可以引用它们。

被调函数的实际参数是在被调用时从其局部符号表中引入的；因此，参数的传递使用传值调用（这里的 值 始终是对象的 引用，不是对象的值）。[1]一个函数调用另一个函数时，会为本次调用创建一个新的局部符号表。

函数定义会在当前符号表内引入函数名。函数名对应的值的类型是解释器可识别的用户自定义函数。此值可以分配给另一个名称，然后该名称也可作为函数。这是通用的重命名机制：

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

如果你使用过其他语言，你可能会反对说： fib 不是一个函数，而是一个过程，因为它并不返回任何值。事实上，没有 return 语句的函数也返回一个值，尽管是一个很无聊的值。此值

被称为 None（它是一个内置的名称）。如果一个变量只等于 None，那么在交互模式下输入变量的名字，解释器是不会打印出来的，如果你真的想看到这个值，可以使用 print()：

```
>>> fib(0)
>>> print(fib(0))
None
```

写一个函数返回菲波那契数列的列表，而不是打印出来，非常简单：

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

此示例中，像往常一样，演示了一些新的 Python 功能：

Return 语句从函数中返回一个值。不带表达式参数的 return 返回 None。函数直接结束后也返回 None。

语句 result.append(a) 调用列表对象 result 的一个 方法。方法是 ‘隶属于’某个对象的函数，被命名成 obj.methodname 的形式，其中 obj 是某个对象 （或是一个表达式），methodname 是由对象类型定义的方法的名称。不同类型定义了不同的方法。不同类型的方法可能具有相同的名称，而不会引起歧义。（也可以使用 class 定义你自己的对象类型和方法，请参见类）本例中所示的 append() 方法是 list 对象定义的。它在列表的末尾添加一个新的元素。在本例中它等同于 result = result +[a]，但效率更高。

4.7.更多关于定义函数

可以定义具有可变数目的参数的函数。有三种函数形式，可以结合使用。

4.7.1.默认参数值

最有用的形式是指定一个或多个参数的默认值。这种方法创建的函数被调用时，可以带有比定义的要少的参数。例如：

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = input(prompt)
```

```
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise OSError('uncooperative user')
        print(complaint)
```

这个函数可以通过几种方式调用：

只给出强制参数： ask_ok (' Do you really want to quit?')
给出一个可选的参数： ask_ok ('OK to overwrite the file?', 2)
或者给出所有的参数： ask_ok ('OK to overwrite the file?', 2, 'Come on, only yes or no!')

此示例还引入了 in 关键字。它测试一个序列是否包含特定的值。

默认值在定义时段中执行函数定义时计算，因此：

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

将打印 5。

重要的警告：默认值只计算一次。这在默认值是列表、字典或大部分类的实例等易变的对象的时候又有所不同。例如，下面的函数在后续调用过程中会累积传给它的参数：

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

这将会打印

```
[1]
[1, 2]
```

[1, 2, 3]

如果你不想默认值在随后的调用中共享，可以像这样编写函数：

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2. 关键字参数

函数也可以通过 kwarg = value 这种 关键字=参数的形式调用。例如，下面的函数：

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

接受一个必选参数 （voltage） 和三个可选参数 （state, action 和 type）。可以用下列任意一种方式调用这个函数：

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

但下面的所有调用将无效：

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

在函数调用中，关键字参数必须跟随在位置参数的后面。传递的所有关键字参数必须与函数接受的某个参数相匹配（例如 actor 不是 parrot 函数的有效参数），它们的顺序并不重要。这也包括非可选参数（例如 parrot(voltage=1000) 也是有效的）。任何参数都不可以多次赋值。下面的示例由于这种限制将失败：

```
>>> def function(a):
...     pass
```

...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'

当最后一个形参以 **name 形式出现，它接收一个字典（见映射类型 —— 字典），该字典包含了所有未出现在形式参数列表中的关键字参数。它还可能与 *name 形式的参数（在下一小节中所述）组合使用，*name 接收一个元组，该元组包含了所有未出现在形参列表中的位置参数。(*name 必须出现在 ** name 之前。)例如，如果我们定义这样的函数：

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    keys = sorted(keywords.keys())
    for kw in keys:
        print(kw, ":", keywords[kw])
```

它可以这样被调用：

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

并且当然它会打印：

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

注意在打印关键字参数之前，通过对关键字字典 keys() 方法的结果进行排序，生成了关键字参数名的列表；如果不这样做，打印出来的参数的顺序是未定义的。

4.7.3.任意参数列表

最后，最不常用的场景是指明某个函数可以被可变个数的参数调用。这些参数被放在一个元组（见元组和序列）中。在可变个数的参数之前，可以有零到多个普通的参数。

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

通常情况下，这些可变参数位于形式参数列表的最后，因为它们会搜集传递给函数的所有剩余输入参数。出现在*args 参数后面的任何形式参数都是 ‘keyword-only’ 参数，意味着它们只能作为关键字参数而不能作为位置参数。

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.7.4. 参数列表的分拆

当传递的参数已经是一个列表或元组时，情况与之前相反，你要分拆这些参数，因为函数调用要求独立的位置参数。例如，内置的 range() 函数期望单独的 start 和 stop 参数。如果它们不是独立的，函数调用时使用 *-操作符将参数从列表或元组中分拆开来：

```
>>> list(range(3, 6))                # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))               # call with arguments unpacked from a list
[3, 4, 5]
```

以同样的方式，可以用**-操作符让字典传递关键字参数：

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

4.7.5. lambda 表达式

可以使用 lambda 关键字创建小的匿名函数。此函数返回其两个参数的总和： lambda a, b:

a + b。Lambda 函数可以用于任何函数对象需要的地方。在语法上，它们被局限于只能有一个单独的表达式。在语义上，他们只是普通函数定义的语法糖。像嵌套的函数定义，lambda 函数可以从其被包含的范围中引用变量：

```
>>> def make_incrementor(n):
...         return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

上面的示例使用 lambda 表达式返回一个函数。另一种用法是将一个小函数作为参数传递：

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.6. 文档字符串

下面是一些关于文档字符串内容和格式的惯例。

第一行永远应该是对象用途的简短、精确的总述。为了简单起见，不应该明确的陈述对象的名字或类型，因为这些信息可以从别的途径了解到（除非这个名字碰巧就是描述这个函数操作的动词）。这一行应该以大写字母开头，并以句号结尾。

如果在文档字符串中有更多的行，第二行应该是空白，在视觉上把摘要与剩余的描述分离开来。以下各行应该是一段或多段描述对象的调用约定、 其副作用等。

Python 解释器不会从多行的文档字符串中去除缩进，所以必要的时候处理文档字符串的工具应当自己清除缩进。这通过使用以下约定可以达到。第一行 之后 的第一个非空行字符串确定整个文档字符串的缩进的量。(我们不用第一行是因为它通常紧靠着字符串起始的引号，其缩进格式不明晰。）所有行起始的等于缩进量的空格都将被过滤掉。不应该存在缩进更少的行，但如果确实存在，应去除所有其前导空白。应该在展开制表符之后（展开后通常为 8 个空格）再去测试留白的长度。

这里是一个多行文档字符串的示例：

```
>>> def my_function():
...         """Do nothing, but document it.
...
...         No, really, it doesn't do anything.
```

```
...         """
...         pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

        No, really, it doesn't do anything.
```

4.7.7. 函数注释

函数注释是可选的元数据信息，用于描述自定义函数中的类型 (see PEP 484 更多信息).

注释以字典的形式存储在函数的__annotations__属性中，对函数的其它任何部分都没有影响。参数注释用参数名后的冒号定义 ，冒号后面紧跟着一个用于计算注释的表达式.返回值注释使用 ->定义，紧跟着参数列表和 def 语句的末尾的冒号之间的表达式.下面的示例包含一个位置参数，一个关键字参数，和没有意义的返回值注释。
>>>

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...         print("Annotations:", f.__annotations__)
...         print("Arguments:", ham, eggs)
...         return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.8. 插曲：编码风格

若要编写更长更复杂的 Python 代码，是时候谈一谈 编码风格了 。大多数语言可以采用不同的风格编写（更准确地讲，是格式化）；其中有一些会比其他风格更具可读性。让你的代码更具可读性是个好主意，而良好的编码风格对此有很大的帮助。

对于 Python 而言， PEP 8 已成为大多数项目遵循的风格指南；它给出了一个高度可读，视觉友好的编码风格。每个 Python 开发者应该阅读一下；这里是为你提取出来的最重要的要点：

使用 4 个空格的缩进，不要使用制表符。

4 个空格是小缩进（允许更深的嵌套）和大缩进（易于阅读）之间很好的折衷。制表符会引起混乱，最好弃用。

折行以确保其不会超过 79 个字符。

这有助于小显示器用户阅读，也可以让大显示器能并排显示几个代码文件。

使用空行分隔函数和类，以及函数内的大块代码。

如果可能，注释独占一行。

使用 docstrings。

运算符周围和逗号后面使用空格，但是括号里侧不加空格： a = f(1, 2) + g(3, 4)。

一致地命名您的类和函数；常见的做法是命名类的时候使用驼峰法，命名函数和方法的时候使用小写字母+下划线法。始终使用 self 作为方法的第一个参数的名称 （请参见初识类更多的关于的类和方法）。

不要使用花哨的编码，如果您的代码只在国际环境中使用。Python 默认的 UTF-8，或者纯 ASCII 在任何情况下永远工作得最好。

同样的，如果讲其它语言的人很少有机会阅读或维护你的代码，不要使用非 ASCII 字符作为标识符。

脚注

[1] 事实上， 按对象引用传递 可能是更恰当的说法，因为如果传递了一个可变对象，调用函数将看到任何被调用函数对该可变对象做出的改变（比如添加到列表中的元素）

5. 数据结构

本章详细讲述你已经学过的一些知识，并增加一些新内容。

5.1. 详解列表

列表数据类型还有更多的方法。这里是列表对象方法的清单：

list.append(x)

添加一个元素到列表的末尾。相当于 a[len(a):] = [x]。

list.extend(L)

将给定列表 L 中的所有元素附加到原列表 a 的末尾。相当于 a[len(a):] = L。

list.insert(i, x)

在给定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，所以 a.insert(0, x) 在列表的最前面插入，a.insert(len(a), x) 相当于 a.append(x)。

list.remove(x)

删除列表中第一个值为 x 的元素。如果没有这样的元素将会报错。

list.pop([i])

删除列表中给定位置的元素并返回它。如果未指定索引，a.pop() 删除并返回列表中的最后一个元素。(i 两边的方括号表示这个参数是可选的，而不是要你输入方括号。你会在 Python 参考库中经常看到这种表示法)。

list.clear()

删除列表中所有的元素。相当于 del a[:]。

list.index(x)

返回列表中第一个值为 x 的元素的索引。如果没有这样的元素将会报错。

list.count(x)

返回列表中 x 出现的次数。

list.sort(cmp=None, key=None, reverse=False)

原地排序列表中的元素。

list.reverse()

反转列表中的元素。

list.copy()

返回列表的一个浅拷贝。等同于 a[:]。

使用了列表大多数方法的例子：

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

你可能已经注意到像 insert, remove 或者 sort 之类的方法只修改列表而没有返回值打印出来 -- 它们其实返回了默认值 None。[1]这是 Python 中所有可变数据结构的设计原则。

5.1.1. 将列表作为堆栈使用

列表方法使得将列表当作堆栈非常容易，最先进入的元素最后一个取出（后进先出）。使用 append() 将元素添加到堆栈的顶部。使用不带索引的 pop() 从堆栈的顶部取出元素。例如：

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
```

```
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. 将列表当作队列使用

也可以将列表当作队列使用，此时最先进入的元素第一个取出（先进先出）；但是列表用作此目的效率不高。在列表的末尾添加和弹出元素非常快，但是在列表的开头插入或弹出元素却很慢 (因为所有的其他元素必须向后移一位)。

如果要实现一个队列，可以使用 collections.deque，它设计的目的就是在两端都能够快速添加和弹出元素。例如：

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. 列表解析

列表解析提供了一个生成列表的简洁方法。应用程序通常会从一个序列的每个元素的操作结果生成新的列表，或者生成满足特定条件的元素的子序列。

例如，假设我们想要创建一个列表 squares：

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
```

```
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

注意这个 for 循环中的被创建(或被重写)的名为 x 的变量在循环完毕后依然存在。使用如下方法，我们可以计算 squares 的值而不会产生任何的副作用：

```
squares = list(map(lambda x: x**2, range(10)))
```

或者，等价地:

```
squares = [x**2 for x in range(10)]
```

上面这个方法更加简明且易读.

列表解析由括号括起来，括号里面包含一个表达式，表达式后面跟着一个 for 语句，后面还可以接零个或更多的 for 或 if 语句。结果是一个新的列表，由表达式依据其后面的 for 和 if 字句上下文计算而来的结果构成。例如，下面的 listcomp 组合两个列表中不相等的元素：

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

它等效于：

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

注意在两个代码段中 for 和 if 语句的顺序是相同的。

如果表达式是一个元组（例如前面示例中的 (x, y)），它必须带圆括号。

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
```

```
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['    banana', '    loganberry ', 'passion fruit    ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in ?
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

列表解析可以包含复杂的表达式和嵌套的函数：

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. 嵌套的列表解析

列表解析中的第一个表达式可以是任何表达式，包括列表解析。

考虑下面由三个长度为 4 的列表组成的 3x4 矩阵：

```
>>> matrix = [
...         [1, 2, 3, 4],
...         [5, 6, 7, 8],
...         [9, 10, 11, 12],
... ]
```

下面的列表解析将转置行和列：

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

正如我们在上一节中看到的，嵌套的 listcomp 在跟随它之后的 for 字句中计算，所以此例等同于：

```
>>> transposed = []
>>> for i in range(4):
...         transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

以此下去，还等同于：

```
>>> transposed = []
>>> for i in range(4):
...         # the following 3 lines implement the nested listcomp
...         transposed_row = []
...         for row in matrix:
...                 transposed_row.append(row[i])
...         transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

在实际中，与复杂的控制流比起来，你应该更喜欢内置的函数。针对这种场景，使用 zip() 函数会更好：

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

关于本行中使用的星号的说明，请参阅参数列表的分拆。

5.2. del 语句

有个方法可以从列表中按索引而不是值来删除一个元素： del 语句。这不同于有返回值的 pop() 方法。 del 语句还可以用于从列表中删除切片或清除整个列表 （我们是将空列表赋值给切片）。例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
```

```
>>> a
[]
```

del 也可以用于删除整个变量：

```
>>> del a
```

此后再引用名称 a 将会报错（直到有另一个值被赋给它）。稍后我们将看到 del 的其它用途 。

5.3. 元组和序列

我们已经看到列表和字符串具有很多共同的属性，如索引和切片操作。它们是 序列 数据类型的两个例子(参见 Sequence Types — list, tuple, range)。因为 Python 是一个正在不断进化的语言，其他的序列类型也可能被添加进来。还有另一种标准序列数据类型：元组。

元组由逗号分割的若干值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构 ；在输入时括号可有可无，不过括号经常都是必须的（如果元组是一个更大的表达式的一部分）。不能给元组中单独的一个元素赋值，不过可以创建包含可变对象（例如列表）的元组。

虽然元组看起来类似于列表，它们经常用于不同的场景和不同的目的。元组是不可变的，通常包含不同种类的元素并通过分拆（参阅本节后面的内容） 或索引访问（如果是 namedtuples，甚至可以通过属性）。列表是 可变 的，它们的元素通常是相同类型的并通过迭代访问。

一个特殊的情况是构造包含 0 个或 1 个元素的元组：为了实现这种情况，语法上有一些奇怪。空元组由一对空括号创建；只有一个元素的元组由值后面跟随一个逗号创建（在括号中放入单独一个值还不够）。丑陋，但是有效。例如：

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

语句 t = 12345, 54321, 'hello!' 是一个元组封装的例子： 值 12345, 54321 和 'hello!' 被一起放入一个元组。其逆操作也是可以的：

```
>>> x, y, z = t
```

这被称为 序列分拆 再恰当不过了，且可以用于右边的任何序列。序列分拆要求等号左侧的变量和序列中的元素的数目相同。注意多重赋值只是同时进行元组封装和序列分拆。

5.4. 集合

Python 还包含了一个数据类型 集合。集合中的元素不会重复且没有顺序。集合的基本用途包括成员测试和消除重复条目。集合对象还支持并集、 交集、 差和对称差等数学运算。

花大括号或 set() 函数可以用于创建集合。注意： 若要创建一个空集必须使用 set()，而不能用 {}；后者将创建一个空的字典，我们会在下一节讨论这种数据结构。

这里是一个简短的演示：

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
```

```
>>> a - b                                      # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                                      # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                                      # letters in both a and b
{'a', 'c'}
>>> a ^ b                                      # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

和 列表解析 类似，Python 也支持集合解析：

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5. 字典

Python 中内置的另一种有用的数据类型是字典（见映射的类型 —— 字典)。有时候你会发现字典在其它语言中被称为 “associative memories” 或者 “associative arrays” 。与序列不同，序列由数字做索引，字典由 键 做索引，键可以是任意不可变类型；字符串和数字永远可以拿来做键。如果元组只包含字符串、 数字或元组，它们可以用作键；如果元组直接或间接地包含任何可变对象，不能用作键。不能用列表作为键，因为列表可以用索引、切片或者 append() 和 extend() 方法修改。

理解字典的最佳方式是把它看做无序的 键:值 对集合，要求是键必须是唯一的（在同一个字典内)。一对大括号将创建一个空的字典： {}。大括号中由逗号分隔的 键:值 对将成为字典的初始值；打印字典时也是按照这种方式输出。

字典的主要操作是依据键来存取值。也可以通过 del 删除 键: 值 对。如果用一个已经存在的键存储值，以前为该关键字分配的值就会被遗忘。从一个不存在的键中读取值会导致错误。

list(d.keys())返回字典中所有键组成的列表，列表的顺序是随机的（如果你想它是有序的，只需使用 sorted(d.keys())代替)。[2]要检查某个键是否在字典中，可以使用 in 关键字。

下面是一个使用字典的小示例：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
```

```
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

dict() 构造函数直接从键-值对序列创建字典：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

此外，字典解析可以用于从任意键和值表达式创建字典：

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

如果键都是简单的字符串，有时通过关键字参数指定 键-值 对更为方便：

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

5.6. 遍历的技巧

循环迭代字典的时候，键和对应的值通过使用 items()方法可以同时得到。

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...         print(k, v)
...
gallahad the pure
robin the brave
```

序列中遍历时，使用 enumerate() 函数可以同时得到索引和对应的值。

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...         print(i, v)
...
0 tic
1 tac
```

2 toe

同时遍历两个或更多的序列，使用 zip() 函数可以成对读取元素。

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...         print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

要反向遍历一个序列，首先正向生成这个序列，然后调用 reversed() 函数。

```
>>> for i in reversed(range(1, 10, 2)):
...         print(i)
...
9
7
5
3
1
```

循环一个序列按排序顺序，请使用 sorted()函数，返回一个新的排序的列表，同时保留源不变。

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...         print(f)
...
apple
banana
orange
pear
```

若要在循环内部修改正在遍历的序列（例如复制某些元素），建议您首先制作副本。在序列上循环不会隐式地创建副本。切片表示法使这尤其方便：

```
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...         if len(w) > 6:
...                 words.insert(0, w)
...
```

```
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

5.7. 深入条件控制

while 和 if 语句中使用的条件可以包含任意的操作，而不仅仅是比较。

比较操作符 in 和 not in 检查一个值是否在一个序列中出现（不出现）。is 和 is not 比较两个对象是否为同一对象；这只和列表这样的可变对象有关。所有比较运算符都具有相同的优先级，低于所有数值运算符。

可以级联比较。例如， a < b == c 测试 a 是否小于 b 并且 b 等于 c。

可将布尔运算符 and 和 or 用于比较，比较的结果（或任何其他的布尔表达式）可以用 not 取反。这些操作符的优先级又低于比较操作符；它们之间，not 优先级最高，or 优先级最低，所以 A and not B or C 等效于 (A and (not B)) or C。与往常一样，可以使用括号来表示所需的组合。

布尔运算符 and 和 or 是所谓的 短路 运算符：依参数从左向右求值，结果一旦确定就停止。例如，如果 A 和 C 都为真，但 B 是假， A and B and C 将不计算表达式 C。用作一个普通值而非逻辑值时，短路操作符的返回值通常是最后一个计算的。

可以把比较或其它逻辑表达式的返回值赋给一个变量。例如，

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

注意 Python 与 C 不同，在表达式内部不能赋值。C 程序员可能会抱怨这一点，但它避免了一类 C 程序中常见的问题： 在表达式中输入 = 而真正的意图是== 。

5.8. 序列和其它类型的比较

序列对象可以与具有相同序列类型的其他对象相比较。比较按照 字典序 进行： 首先比较最前面的两个元素，如果不同，就决定了比较的结果；如果相同，就比较下两个元素，依此类推，直到其中一个序列穷举完。如果要比较的两个元素本身就是同一类型的序列，就按字典序递归比较。如果两个序列的所有元素都相等，就认为序列相等。如果一个序列是另一个序列的初始子序列，较短的序列就小于另一个。字符串的排序按照 Unicode 编码点的数值排序单个字符。下面是同类型序列之间比较的一些例子：

```
(1, 2, 3)                   < (1, 2, 4)
[1, 2, 3]                   < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)                < (1, 2, 4)
```

```
(1, 2)                          < (1, 2, -1)
(1, 2, 3)                       == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))     < (1, 2, ('abc', 'a'), 4)
```

注意，使用< 或者 >比较不同类型的对象是合法的，只要这些对象具有合适的比较方法。例如，不同的数字类型按照它们的数值比较，所以 0 等于 0.0，等等。否则，解释器将引发一个 TypeError 异常，而不是随便给一个顺序。

脚注

[1] 其它语言也许会返回可变对象，这么做可以连续地调用方法，比如 d->insert("a")->remove("b")->sort();。

[2] 调用 d.keys() 会返回一个 字典视图 对象。 该对象支持成员测试和迭代操作，但它的内容依赖原字典——它只是一个 视图。

6. 模块

如果你退出 Python 解释器并重新进入，你做的任何定义（变量和方法）都会丢失。因此，如果你想要编写一些更大的程序，最好使用文本编辑器先编写好，然后运行这个文件。 这就是所谓的创建 脚本。随着你的程序变得越来越长，你可能想要将它分成几个文件，这样更易于维护。你还可能想在几个程序中使用你已经编写好的函数，而不用把函数拷贝到每个程序中。

为了支持这个功能，Python 有种方法可以把你定义的内容放到一个文件中，然后在脚本或者交互方式中使用。这种文件称为模块 ;模块中的定义可以 导入 到其它模块或 主模块 中。

模块是包含 Python 定义和声明的文件。文件名就是模块名加上.py 后缀。在模块里面，模块的名字（是一个字符串）可以由全局变量 __name__ 的值得到。例如，用你喜欢的文本编辑器在当前目录下创建一个名为 fibo.py 的文件，内容如下：

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

现在进入 Python 解释器并使用下面的命令导入这个模块：

```
>>> import fibo
```

这不会直接把 fibo 中定义的函数的名字导入当前的符号表中；它只会把模块名字 fibo 导入其中。你可以通过模块名访问这些函数：

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
```

'fibo'

如果你打算频繁使用一个函数，可以将它赋给一个本地的变量：

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. 深入模块

模块可以包含可执行语句以及已定义的函数。这些语句通常用于初始化模块。它们只在 第一次 导入时执行。[1]（如果文件以脚本的方式执行，它们也会运行。）

每个模块都有自己的私有符号表，模块内定义的所有函数用其作为全局符号表。因此，模块的作者可以在模块里使用全局变量，而不用担心与某个用户的全局变量有冲突。另一方面，如果你知道自己在做什么，你可以使用引用模块函数的表示法访问模块的全局变量，modname.itemname。

模块中可以导入其它模块。习惯上将所有的 import 语句放在模块（或者脚本）的开始，但这不是强制性的。被导入的模块的名字放在导入模块的全局符号表中。

import 语句的一个变体直接从被导入的模块中导入名字到模块的符号表中。例如：

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这不会把模块名导入到本地的符号表中（所以在本例中，fibo 将没有定义）。

还有种方式可以导入模块中定义的所有名字：

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这种方式不会导入以下划线（_）开头的名称。大多数情况下 Python 程序员不要使用这个便利的方法，因为它会引入一系列未知的名称到解释器中，这很可能隐藏你已经定义的一些东西。

注意一般情况下不赞成从一个模块或包中导入 * ，因为这通常会导致代码很难读。不过，在交互式会话中这样用是可以的，它可以让你少敲一些代码。

注意

出于性能考虑，每个模块在每个解释器会话中只导入一遍。因此，如果你修改了你的模块，你必需重新启动解释器 —— 或者，如果你就是想交互式的测试这么一个模块，可以使用 imp.reload()，例如 import imp; imp.reload(modulename)。

6.1.1. 执行模块

当你用下面的方式运行一个 Python 模块

```
python fibo.py <arguments>
```

模块中的代码将会被执行，就像导入它一样，不过此时 __name__ 被设置为 "__main__" 。也就是说，如果你在模块后加入如下代码:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

就可以让此文件既可以作为可执行的脚本，也可以当作可以导入的模块，因为解析命令行的那部分代码只有在模块作为 “main” 文件执行时才被调用：

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

如果模块是被导入的，将不会运行这段代码：

```
>>> import fibo
>>>
```

这种方法通常用来为模块提供一个方便的用户接口，或者用来测试（例如直接运行脚本会执行一组测试用例）。

6.1.2. 模块搜索路径

当导入一个名为 spam 的模块时，解释器首先搜索具有该名称的内置模块。如果没有找到，它会接着到 sys.path 变量给出的目录中查找名为 spam.py 的文件。sys.path 变量的初始值来自这些位置：

脚本所在的目录（如果没有指明文件，则为当前目录）。
PYTHONPATH （一个包含目录名的列表，与 shell 变量 PATH 的语法相同）。
与安装相关的默认值。

Note

在支持符号连接的文件系统中，输入的脚本所在的目录是符号连接指向的目录。 换句话说也就是包含符号链接的目录不会被加到目录搜索路径中。

初始化后，Python 程序可以修改 sys.path。脚本所在的目录被放置在搜索路径的最开始，也就是在标准库的路径之前。这意味着将会加载当前目录中的脚本，库目录中具有相同名称的模块不会被加载。除非你是有意想替换标准库，否则这应该被当成是一个错误。更多信息请参阅 标准模块 小节。

6.1.3. "编译过的" Python 文件

为了加快加载模块的速度，Python 会在__pycache__目录下以 module.version.pyc 名字缓存每个模块编译后的版本，这里的版本编制了编译后文件的格式。它通常会包含 Python 的版本号。例如，在 CPython 3.3 版中，spam.py 编译后的版本将缓存为 __pycache__/spam.cpython-33.pyc。这种命名约定允许由不同发布和不同版本的 Python 编译的模块同时存在。

Python 会检查源文件与编译版的修改日期以确定它是否过期并需要重新编译。这是完全自动化的过程。同时，编译后的模块是跨平台的，所以同一个库可以在不同架构的系统之间共享。

Python 在两种情况下不检查缓存。第一，它总会重新编译并且不会存储从命令行直接加载的模块。第二，如果没有源模块它不会检查缓存。若要支持没有源文件（只有编译版）的发布，编译后的模块必须在源目录下，并且必须没有源文件的模块。

部分高级技巧：

{{s.58}}{{s.59}}{{s.60}}{{s.61}}{{s.62}}

{{s.63}}{{s.64}}{{s.65}}

{{s.66}}{{s.67}}

{{s.68}}{{s.69}}{{s.70}}

{{s.71}}{{条例}}

{{}} s.73

6.2. 标准模块

Python 带有一个标准模块库，并发布有单独的文档叫 Python 库参考手册（以下简称"库参考手册"）。有些模块被直接构建在解析器里；这些操作虽然不是语言核心的部分，但是依然被内建进来，一方面是效率的原因，另一方面是为了提供访问操作系统原语如系统调用的功能。这些模块是可配置的，也取决于底层的平台。例如，winreg 模块只在 Windows 系统上提供。有一个特别的模块需要注意： sys，它内置在每一个 Python 解析器中。变量 sys.ps1 和 sys.ps2 定义了主提示符和辅助提示符使用的字符串：

>>> import sys

```
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

只有在交互式模式中，这两个变量才有定义。

变量 sys.path 是一个字符串列表，它决定了解释器搜索模块的路径。它初始的默认路径来自于环境变量 PYTHONPATH，如果 PYTHONPATH 未设置则来自于内置的默认值。你可以使用标准的列表操作修改它：

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3. dir()函数

内置函数 dir() 用来找出模块中定义了哪些名字。它返回一个排好序的字符串列表：

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

如果不带参数， dir() 列出当前已定义的名称：

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

注意它列出了所有类型的名称： 变量、 模块、 函数等。

dir()不会列出内置的函数和变量的名称。如果你想列出这些内容，它们定义在标准模块 builtins 中：

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 'zip']
```

6.4. 包

包是一种管理 Python 模块命名空间的方式，采用“点分模块名称”。例如，模块名 A.B 表示包 A 中一个名为 B 的子模块。就像模块的使用让不同模块的作者不用担心相互间的全局变量名称一样，点分模块的使用让包含多个模块的包（例如 Numpy 和 Python Imaging Library）的作者也不用担心相互之间的模块重名。

假设你想要设计一系列模块（或一个“包”）来统一处理声音文件和声音数据。现存很多种不同的声音文件格式（通常由它们的扩展名来识别，例如：.wav, .aiff, .au），因此你可能需要创建和维护不断增长的模块集合来支持各种文件格式之间的转换。你可能还想针对音频数据做很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果），所以你还需要编写一组永远写不完的模块来处理这些操作。你的包可能会是这个结构（用分层的文件系统表示）：

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

导入这个包时，Python 搜索 sys.path 中的目录以寻找这个包的子目录。

为了让 Python 将目录当做包，目录下必须包含 __init__.py 文件；这样做是为了防止一个具有常见名字（例如 string）的目录无意中隐藏目录搜索路径中正确的模块。最简单的情况下，__init__.py 可以只是一个空的文件，但它也可以为包执行初始化代码或设置__all__变量

（稍后会介绍）。

用户可以从包中导入单独的模块，例如：

import sound.effects.echo

这样就加载了子模块 sound.effects.echo。它必须使用其完整名称来引用。

sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

导入子模块的另一方法是：

from sound.effects import echo

这同样也加载了子模块 echo，但使它可以不用包前缀访问，因此它可以按如下方式使用：

echo.echofilter(input, output, delay=0.7, atten=4)

还有另一种变化方式是直接导入所需的函数或变量：

from sound.effects.echo import echofilter

这再次加载了子模块 echo，但这种方式使函数 echofilter() 可以直接访问：

echofilter(input, output, delay=0.7, atten=4)

注意使用 from package import item 时，item 可以是包的子模块（或子包），也可以是包中定义的一些其它的名称，比如函数、 类或者变量。import 语句首先测试 item 在包中是否有定义；如果没有，它假定它是一个模块，并尝试加载它。如果未能找到，则引发 ImportError 异常。

相反，使用类似 import item.subitem.subsubitem 这样的语法时，除了最后一项其它每项必须是一个包；最后一项可以是一个模块或一个包，但不能是在前一个项目中定义的类、函数或变量。

6.4.1. 从包中导入 *

那么现在如果用户写成 from sound.effects import * 会发生什么？理想情况下，他应该是希望到文件系统中寻找包里面有哪些子模块，并把它们全部导入进来。这可能需要很长时间，而且导入子模块可能会产生想不到的副作用，这些作用本应该只有当子模块是显式导入时才会发生。

唯一的解决办法是包的作者为包提供显式的索引。import 语句使用以下约定： 如果包中的 __init__.py 代码定义了一个名为__all__的列表，那么在遇到 from package import *语句的时候，应该把这个列表中的所有模块名字导入。当包有新版本包发布时，就需要包的作者更新

这个列表了。如果包的作者认为不可以用 import * 方式导入它们的包，也可以决定不支持它。例如，文件 sound/effects/__init__.py 可以包含下面的代码：

```
__all__ = ["echo", "surround", "reverse"]
```

这意味着 from sound.effects import * 将导入 sound 包的三个子模块。

如果 __all__ 没有定义，from sound.effects import * 语句 不 会从 sound.effects 包中导入所有的子模块到当前命名空间；它只保证 sound.effects 包已经被导入（可能会运行 __init__.py 中的任何初始化代码），然后导入包中定义的任何名称。这包括由 __init__.py 定义的任何名称（以及它显式加载的子模块）。还包括这个包中已经由前面的 import 语句显式加载的子模块。请考虑此代码：

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

在这个例子中，执行 from...import 语句时，echo 和 surround 模块被导入到当前命名空间是因为它们在 sound.effects 中有定义。（定义了 __all__时也会同样工作。）

某些模块设计成使用 import * 时只导出符合特定模式的名称。即便如此，在产品代码中使用这种写法仍然是不好的做法。

记住，使用 from Package import specific_submodule 一点没错 ！事实上，这是推荐的写法，除非导入的模块需要使用其它包中的同名子模块。

6.4.2. 包内引用

如果一个包是子包（比如例子中的 sound 包），你可以使用绝对导入来引用兄弟包的子模块。例如，如果模块 sound.filters.vocoder 需要使用 sound.effects 包中的 echo 模块，它可以使用 from sound.effects import echo。

你还可以用 from module import name 形式的 import 语句进行相对导入。这些导入使用前导的点号表示相对导入的是从当前包还是上级的包。以 surround 模块为例，你可以使用：

```
from . import echo
from .. import formats
from ..filters import equalizer
```

注意，相对导入基于当前模块的名称。因为主模块的名字总是 "__main__" ，Python 应用程序的主模块必须总是用绝对导入。

6.4.3. 包含多个目录的包

包还支持一个特殊的属性，__path__。该变量初始化一个包含 __init__.py 所在目录的列表。这个变量可以修改；这样做会影响未来包中包含的模块和子包的搜索。

虽然通常不需要此功能，它可以用于扩展包中的模块的集合。

脚注

[1] 事实上函数定义同样是 ‘被执行’ 的 ‘语句’ ;模块级函数定义的执行使该函数的名称存入了模块的全局符号表中。

7. 输入和输出

展现程序的输出有多种方法 ;可以打印成人类可读的形式, 也可以写入到文件以便后面使用。本章将讨论其中的几种方法。

7.1. 格式化输出

到目前为止我们遇到过两种输出值的方法：表达式语句和 print()函数。（第三个方式是使用文件对象的 write()方法；标准输出文件可以引用 sys.stdout。详细内容参见库参考手册。）

通常你会希望更好地控制输出的格式而不是简单地打印用空格分隔的值。有两种方法来设置输出格式；第一种方式是自己做所有的字符串处理；使用字符串切片和连接操作，你可以创建任何你能想象到的布局。字符串类型有一些方法，用于执行将字符串填充到指定列宽度的有用操作；这些稍后将讨论。第二种方法是使用 str.format()方法。

string 模块包含一个 Template 类，提供另外一种向字符串代入值的方法。

当然还有一个问题：如何将值转换为字符串？幸运的是，Python 有方法将任何值转换为字符串：将它传递给 repr()或 str()函数。

str()函数的用意在于返回人类可读的表现形式，而 repr()的用意在于生成解释器可读的表现形式（如果没有等价的语法将会引发 SyntaxError 异常）。对于对人类并没有特别的表示形式的对象，str()和 repr()将返回相同的值。许多值，例如数字或者列表和字典这样的结构，使用这两个函数中的任意一个都具有相同的表示形式。字符串特殊一些, 有两种不同的表示形式。

一些例子：

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
```

... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"

这里用两种方法输出平方和立方表：

```
>>> for x in range(1, 11):
...         print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...         # Note use of 'end' on previous line
...         print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1, 11):
...         print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

（注意在第一个示例中，每列之间的一个空格由 print()自动添加：它总会在它的参数之间添加空格。）

上面的例子演示了字符串对象的 str.rjust()方法，它通过在左侧填充空格使字符串在给定宽度的列右对齐。类似的方法还有 str.ljust()和 str.center()。这些方法不会输出任何内容，它们只返回新的字符串。如果输入的字符串太长，它们不会截断字符串，而是保持原样返回；这会使列的格式变得混乱，但是通常好于另外一种选择，那可能是一个错误的值。（如果你真的想要截断，可以加上一个切片操作，例如 x.ljust(n)[:n]。）

另外一种方法 str.zfill()，它向数值字符串左侧填充零。该函数可以正确识别正负号：

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

str.format()方法的基本用法如下所示：

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

花括号及其中的字符（称为格式字段）将被替换为传递给 str.format()方法的对象。可以用括号中的数字指定传递给 str.format()方法的对象的位置。

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

如果 str.format()方法使用关键字参数，那么将通过参数名引用它们的值。

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

位置参数和关键字参数可以随意组合：

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
                                                       other='Georg'))
The story of Bill, Manfred, and Georg.
```

'!a'（应用 ascii()），'!s'（应用 str()）和'!r'（应用 repr()）可以用来格式化之前对值进行转换。

```
>>> import math
>>> print('The value of PI is approximately {}.'.format(math.pi))
The value of PI is approximately 3.14159265359.
>>> print('The value of PI is approximately {!r}.'.format(math.pi))
The value of PI is approximately 3.141592653589793.
```

字段名后允许可选的': '和格式指令。这允许更好地控制如何设置值的格式。下面的例子将 Pi 转为三位精度。

```
>>> import math
>>> print('The value of PI is approximately {0:.3f}.'.format(math.pi))
The value of PI is approximately 3.142.
```

':'后面紧跟一个整数可以限定该字段的最小宽度。这在美化表格时很有用。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...         print('{0:10} ==> {1:10d}'.format(name, phone))
...
Jack           ==>           4098
Dcab            ==>            7678
Sjoerd         ==>           4127
```

如果你有一个实在是很长的格式字符串但又不想分开写，要是可以按照名字而不是位置引用变量就好了。有个简单的方法，可以传入一个字典，然后使用'[]'访问。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...             'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

这也可以用 '**' 标志将这个字典以关键字参数的方式传入。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

这种方式与内置函数 vars()组合起来将更加有用，该函数返回一个包含所有局部变量的字典。

关于 str.format()完整的描述，请参见格式字符串语法。

7.1.1. 旧式的字符串格式

%运算符也可以用于字符串格式化。它将左边类似 sprintf()-风格的参数应用到右边的参数，然后返回这种格式化操作生成的字符串。例如：

```
>>> import math
>>> print('The value of PI is approximately %5.3f.' % math.pi)
The value of PI is approximately 3.142.
```

在 printf-style String Formatting 一节，可以找到更多的信息。

7.2. 读写文件

open()返回一个文件对象，最常见的用法带有两个参数：open(filename, mode)。

```
>>> f = open('workfile', 'w')
```

第一个参数是一个含有文件名的字符串。第二个参数也是一个字符串，含有描述如何使用该文件的几个字符。mode 为'r'时表示只是读取文件；w 表示只是写入文件（已经存在的同名文件将被删掉）；'a'表示打开文件进行追加，写入到文件中的任何数据将自动添加到末尾。'r+'表示打开文件进行读取和写入。mode 参数是可选的，默认为'r'。

通常，文件以文本打开，这意味着，你从文件读出和向文件写入的字符串会被特定的编码方式（默认是 UTF-8）编码。模式后面的'b'以二进制模式打开文件：数据会以字节对象的形式读出和写入。这种模式应该用于所有不包含文本的文件。

在文本模式下，读取时默认会将平台有关的行结束符（Unix 上是\n, Windows 上是\r\n）转换为\n。在文本模式下写入时，默认会将出现的\n 转换成平台有关的行结束符。这种暗地里的修改对 ASCII 文本文件没有问题，但会损坏 JPEG 或 EXE 这样的二进制文件中的数据。使用二进制模式读写此类文件时要特别小心。

7.2.1. 文件对象的方法

本节中的示例将假设文件对象 f 已经创建。

要读取文件内容，可以调用 f.read(size)，该方法读取若干数量的数据并以字符串或字节对象返回。size 是可选的数值参数。当 size 被省略或者为负数时，将会读取并返回整个文件；如果文件大小是你机器内存的两倍时，就是你的问题了。否则，至多读取和返回 size 大小的字节数据。如果到了文件末尾，f.read() 会返回一个空字符串('')。

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

f.readline()从文件读取一行数据；字符串结尾会带有一个换行符（\n），只有当文件最后一行没有以换行符结尾时才会省略。这样返回值就不会有混淆，如果 f.readline()返回一个空字符串，那就表示已经达到文件的末尾，而如果返回一个只包含一个换行符的字符串'\n'，则表示遇到一个空行。

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

你可以循环遍历文件对象来读取文件中的每一行。这是既省内存又非常快的简单代码：

```
>>> for line in f:
...         print(line, end='')
...
This is the first line of the file.
Second line of the file
```

如果你想把文件中的所有行读到一个列表中，你也可以使用 list(f)或 f.readlines()。

f.write(string)将 string 的内容写入文件中并返回写入的字节的数目。

```
>>> f.write('This is a test\n')
15
```

如果想写入字符串以外的数据，需要先将它转换为一个字符串：

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
18
```

f.tell()返回一个给出文件对象在文件中当前位置的整数，在二进制模式下表示自文件开头的字节数，在文本模式下是一个无法理解的数。

若要更改该文件对象的位置，可以使用 f.seek(offset, from_what)。新的位置由参考点加上 offset 计算得来，参考点的选择则来自于 from_what 参数。from_what 值为 0 表示以文件的开始为参考点，1 表示以当前的文件位置为参考点，2 表示以文件的结尾为参考点。from_what 可以省略，默认值为 0，表示以文件的开始作为参考点。

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)      # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

在文本文件中（没有以 b 模式打开），只允许从文件头开始寻找（有个例外是用 seek(0, 2)寻找文件的最末尾处）而且合法的偏移值只能是 f.tell()返回的值或者是零。其它任何偏移 值都会产生未定义的行为。

使用完一个文件后，调用 f.close()可以关闭它并释放其占用的所有系统资源。调用 f.close()后，再尝试使用该文件对象将失败。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

处理文件对象时使用with关键字是很好的做法。这样做的好处在于文件用完后会自动关闭，即使过程中发生异常也没关系。它还比编写一个等同的 try-finally 语句要短很多：

```
>>> with open('workfile', 'r') as f:
...         read_data = f.read()
>>> f.closed
True
```

文件对象还有一些不太常用的方法，例如 isatty()和 truncate()；有关文件对象的完整指南，请参阅 Python 库参考手册。

7.2.2. 使用 json 存储结构化数据

从文件中读写字符串很容易。数值就要多费点儿周折，因为 read ()方法只会返回字符串，应将其传入 int()这样的函数，就可以将'123'这样的字符串转换为对应的数值 123。当你想要保存更为复杂的数据类型，例如嵌套的列表和字典，手工解析和序列化它们将变得更复杂。

好在用户不是非得自己编写和调试保存复杂数据类型的代码，Python 允许你使用常用的数据交换格式 JSON（JavaScript Object Notation）。标准模块 json 可以接受 Python 数据结构，并将它们转换为字符串表示形式；此过程称为序列化。从字符串表示形式重新构建数据结构称为反序列化。序列化和反序列化的过程中，表示该对象的字符串可以存储在文件或数据中，也可以通过网络连接传送给远程的机器。

注意

JSON 格式经常用于现代应用程序中进行数据交换。许多程序员都已经熟悉它了，使它成为相互协作的一个不错的选择。

如果你有一个对象 x，你可以用简单的一行代码查看其 JSON 字符串表示形式：

```
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

dumps()函数的另外一个变体 dump()，直接将对象序列化到一个文本文件。所以如果 f 是为写入而打开的一个文本文件对象，我们可以这样做：

```
json.dump(x, f)
```

如果 f 是为读取而打开的文本文件对象，需要重新解码对象，

x = json.load(f)

这种简单的序列化技术可以处理列表和字典，但序列化任意类实例为 JSON 需要一点额外的努力。 Json 模块的手册对此有详细的解释。

另请参阅

pickle - pickle 模块

与 JSON 不同，pickle 是一个协议，它允许任意复杂的 Python 对象的序列化。因此，它只能用于 Python 而不能用来与其他语言编写的应用程序进行通信。默认情况下它也是不安全的：如果数据由熟练的攻击者精心设计， 反序列化来自一个不受信任源的 pickle 数据可以执行任意代码。

8. 错误和异常

直到现在，我们还没有更多的提及错误信息，但是如果你真的尝试了前面的例子，也许你已经见到过一些。Python（至少）有两种错误很容易区分：语法错误 和异常。

8.1. 语法错误

语法错误，或者称之为解析错误，可能是你在学习 Python 过程中最烦的一种：

```
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的’ 箭头’ 。错误是由箭头前面 的标记引起的（至少检测到是这样的）： 在这个例子中，检测到错误发生在函数 print()，因为在它之前缺少一个冒号（':'）。文件名和行号会一并输出，所以如果运行的是一个脚本你就知道去哪里检查错误了。

8.2. 异常

即使一条语句或表达式在语法上是正确的，在运行它的时候，也有可能发生错误。在执行期间检测到的错误被称为异常 并且程序不会无条件地崩溃：你很快就会知道如何在 Python 程序中处理它们。然而大多数异常都不会被程序处理，导致产生类似下面的错误信息：

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

最后一行的错误消息指示发生了什么事。异常有不同的类型，其类型会作为消息的一部分打印出来：在这个例子中的类型有 ZeroDivisionError、 NameError 和 TypeError。打印出来的异常类型的字符串就是内置的异常的名称。这对于所有内置的异常是正确的，但是对于用户自定义的异常就不一定了（尽管这是非常有用的惯例）。标准异常的名称都是内置的标识符（不是保留的关键字）。

这一行最后一部分给出了异常的详细信息和引起异常的原因。

错误信息的前面部分以堆栈回溯的形式显示了异常发生的上下文。通常调用栈里会包含源代码的行信息，但是来自标准输入的源码不会显示行信息。

内置的异常 列出了内置的异常以及它们的含义。

8.3. 抛出异常

可以通过编程来选择处理部分异常。看一下下面的例子，它会一直要求用户输入直到输入一个合法的整数为止，但允许用户中断这个程序（使用 Control-C 或系统支持的任何方法）；注意用户产生的中断引发的是 KeyboardInterrupt 异常。

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

Try 语句按以下方式工作。

首先，执行 try 子句（try 和 except 关键字之间的语句）。

如果未发生任何异常，忽略 except 子句且 try 语句执行完毕。

如果在 try 子句执行过程中发生异常，跳过该子句的其余部分。如果异常的类型与 except 关键字后面的异常名匹配，则执行 except 子句，然后继续执行 try 语句之后的代码。

如果异常的类型与 except 关键字后面的异常名不匹配，它将被传递给上层的 try 语句；如果没有找到处理这个异常的代码，它就成为一个未处理异常，程序会终止运行并显示一条如上所示的信息。

Try 语句可能有多个子句，以指定不同的异常处理程序。不过至多只有一个处理程序将被执行。处理程序只处理发生在相应 try 子句中的异常，不会处理同一个 try 子句的其他处理程序中发生的异常。一个 except 子句可以用带括号的元组列出多个异常的名字，例如：

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

最后一个 except 子句可以省略异常名称，以当作通配符使用。使用这种方式要特别小心，因为它会隐藏一个真实的程序错误！它还可以用来打印一条错误消息，然后重新引发异常（让调用者也去处理这个异常）：

```
import sys

try:
    f = open('myfile.txt')
```

```
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

try...except 语句有一个可选的 else 子句，其出现时，必须放在所有 except 子句的后面。如果需要在 try 语句没有抛出异常时执行一些代码，可以使用这个子句。例如：

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

使用 else 子句比把额外的代码放在 try 子句中要好，因为它可以避免意外捕获不是由 try ... 保护的代码所引发的异常。except 语句。

当异常发生时，它可能带有相关数据，也称为异常的参数。参数的有无和类型取决于异常的类型。

except 子句可以在异常名之后指定一个变量。这个变量将绑定于一个异常实例，同时异常的参数将存放在实例的 args 中。为方便起见，异常实例定义了__str__() ，因此异常的参数可以直接打印而不必引用.args。也可以在引发异常之前先实例化一个异常，然后向它添加任何想要的属性。

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)           # __str__ allows args to be printed directly,
...                           # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
```

```
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

对于未处理的异常，如果它含有参数，那么参数会作为异常信息的最后一部分打印出来。

异常处理程序不仅处理直接发生在 try 子句中的异常，而且还处理 try 子句中调用的函数（甚至间接调用的函数）引发的异常。例如：

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: int division or modulo by zero
```

8.4. 引发异常

raise 语句允许程序员强行引发一个指定的异常。例如：

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

raise 的唯一参数指示要引发的异常。它必须是一个异常实例或异常类（从 Exception 派生的类）。

如果你确定需要引发异常，但不打算处理它，一个简单形式的 raise 语句允许你重新引发异常：

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
```

```
  File "<stdin>", line 2, in ?
NameError: HiThere
```

8.5. 用户定义的异常

程序可以通过创建新的异常类来命名自己的异常（Python 类的更多内容请参见类）。异常通常应该继承 Exception 类，直接继承或者间接继承都可以。例如：

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print('My exception occurred, value:', e.value)
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

在此示例中，Exception 默认的__init__()被覆盖了。新的行为简单地创建了 value 属性。这将替换默认的创建 args 属性的行为。

异常类可以像其他类一样做任何事情，但是通常都会比较简单，只提供一些属性以允许异常处理程序获取错误相关的信息。创建一个能够引发几种不同错误的模块时，一个通常的做法是为该模块定义的异常创建一个基类，然后基于这个基类为不同的错误情况创建特定的子类：

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """
```

```
        def __init__(self, expression, message):
            self.expression = expression
            self.message = message

class TransitionError(Error):
        """Raised when an operation attempts a state transition that's not
        allowed.

        Attributes:
            previous -- state at beginning of transition
            next -- attempted new state
            message -- explanation of why the specific transition is not allowed
        """

        def __init__(self, previous, next, message):
            self.previous = previous
            self.next = next
            self.message = message
```

大多数异常的名字都以"Error"结尾，类似于标准异常的命名。

很多标准模块中都定义了自己的异常来报告在它们所定义的函数中可能发生的错误。类 这一章给出了类的详细信息。

8.6. 定义清理操作

Try 语句有另一个可选的子句，目的在于定义必须在所有情况下执行的清理操作。例如：

```
>>> try:
...         raise KeyboardInterrupt
... finally:
...         print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
```

不管有没有发生异常，在离开 try 语句之前总是会执行 finally 子句。当 try 子句中发生了一个异常，并且没有 except 字句处理(或者异常发生在 except 或 else 子句中)，在执行完 finally 子句后将重新引发这个异常。try 语句由于 break、contine 或 return 语句离开时，同样会执行 finally 子句。下面是一个更复杂些的例子：

```
>>> def divide(x, y):
...         try:
...                 result = x / y
```

```
...         except ZeroDivisionError:
...             print("division by zero!")
...         else:
...             print("result is", result)
...         finally:
...             print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

正如您所看到的，在任何情况下都会执行 finally 子句。由两个字符串相除引发的 TypeError 异常没有被 except 子句处理，因此在执行 finally 子句后被重新引发。

在真实的应用程序中， finally 子句用于释放外部资源（例如文件或网络连接），不管资源的使用是否成功。

8.7. 清理操作的预定义

有些对象定义了在不需要该对象时的标准清理操作，无论该对象的使用是成功还是失败。看看下面的示例，它尝试打开一个文件并打印其内容到屏幕。

```
for line in open("myfile.txt"):
    print(line, end="")
```

这段代码的问题就是这部分代码执行完之后它还会让文件在一段不确定的时间内保持打开状态。这在简单的脚本中没什么，但是在大型应用程序中可能是一个问题。 With 语句可以确保像文件这样的对象总能及时准确地被清理掉。

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

执行该语句后，文件 f 将始终被关闭，即使在处理某一行时遇到了问题。提供预定义的清理行为的对象，和文件一样，会在它们的文档里说明。

9. 类

与其他编程语言相比，Python 的类机制用最少的语法和语义引入了类。它是 C++ 和 Modula-3 类机制的混合。Python 的类提供了面向对象编程的所有标准功能： 类继承机制允许有多个基类，继承的类可以覆盖其基类或类的任何方法，方法能够以相同的名称调用基类中的方法。对象可以包含任意数量和种类的数据。和模块一样，类同样具有 Python 的动态性质：它们在运行时创建，并可以在创建之后进一步修改。

用 C++ 术语来讲，通常情况下类成员（包括数据成员）是公有的（其它情况见下文私有变量），所有的成员函数都是虚 的。与 Modula-3 一样，在成员方法中没有简便的方式引用对象的成员：方法函数的声明用显式的第一个参数表示对象本身，调用时会隐式地引用该对象。与 Smalltalk 一样，类本身也是对象。这给导入类和重命名类提供了语义上的合理性。与 C++ 和 Modula-3 不同，用户可以用内置类型作为基类进行扩展。此外，像 C++ 一样，类实例可以重定义大多数带有特殊语法的内置操作符（算术运算符、 下标等）。

（由于没有统一的达成共识的术语，我会偶尔使用 SmallTalk 和 C++ 的术语。我比较喜欢用 Modula-3 的术语，因为比起 C++，Python 的面向对象语法更像它，但是我想很少有读者听说过它。）

9.1. 名称和对象

对象是独立的，多个名字（在多个作用域中）可以绑定到同一个对象。这在其他语言中称为别名。第一次粗略浏览 Python 时经常不会注意到这个特性，而且处理不可变的基本类型（数字，字符串，元组）时忽略这一点也没什么问题。然而， 在 Python 代码涉及可变对象如列表、 字典和大多数其它类型时，别名可能具有意想不到语义效果。这通常有助于优化程序，因为别名的行为在某些方面类似指针。例如，传递一个对象的开销是很小的，因为在实现上只是传递了一个指针；如果函数修改了参数传递的对象，调用者也将看到变化 —— 这就避免了类似 Pascal 中需要两个不同参数的传递机制。

9.2. Python 作用域和命名空间

在介绍类之前，首先我要告诉你一些有关 Python 作用域的的规则。类的定义非常巧妙的运用了命名空间，要完全理解接下来的知识，需要先理解作用域和命名空间的工作原理。另外，这一切的知识对于任何高级 Python 程序员都非常有用。

让我们从一些定义开始。

命名空间是从名称到对象的映射。当前命名空间主要是通过 Python 字典实现的，不过通常不会引起任何关注（除了性能方面），它以后也有可能会改变。以下有一些命名空间的例子：内置名称集（包括函数名列如 abs()和内置异常的名称）；模块中的全局名称；函数调用中的局部名称。在某种意义上的一组对象的属性也形成一个命名空间。关于命名空间需要知道的重要一点是不同命名空间的名称绝对没有任何关系；例如，两个不同模块可以都定义函数 maximize 而不会产生混淆 —— 模块的使用者必须以模块名为前缀引用它们。

顺便说一句，我使用属性 这个词称呼点后面的任何名称 —— 例如，在表达式 z.real 中，real 是 z 对象的一个属性。严格地说，对模块中的名称的引用是属性引用：在表达式

modname.funcname 中， modname 是一个模块对象，funcname 是它的一个属性。在这种情况下，模块的属性和模块中定义的全局名称之间碰巧是直接的映射：它们共享同一命名空间 ！[1]

属性可以是只读的也可以是可写的。在后一种情况下，可以对属性赋值。模块的属性都是可写的 :你可以这样写 modname.the_answer = 42。可写的属性也可以用 del 语句删除。例如，del modname.the_answer 将会删除对象 modname 中的 the_answer 属性。

各个命名空间创建的时刻是不一样的，且有着不同的生命周期。包含内置名称的命名空间在 Python 解释器启动时创建，永远不会被删除。模块的全局命名空间在读入模块定义时创建；通常情况下，模块命名空间也会一直保存到解释器退出。在解释器最外层调用执行的语句，不管是从脚本文件中读入还是来自交互式输入，都被当作模块__main__的一部分，所以它们有它们自己的全局命名空间。(内置名称实际上也存在于一个模块中，这个模块叫 builtins。)

函数的局部命名空间在函数调用时创建，在函数返回或者引发了一个函数内部没有处理的异常时删除。(实际上，用遗忘来形容到底发生了什么更为贴切。)当然，每个递归调用有它们自己的局部命名空间。

作用域 是 Python 程序中可以直接访问一个命名空间的代码区域。这里的 “直接访问” 的意思是用没有前缀的引用在命名空间中找到的相应的名称。

虽然作用域是静态确定的，但是使用它们时是动态的。程序执行过程中的任何时候，至少有三个嵌套的作用域，它们的命名空间是可以直接访问的：

首先搜索最里面包含局部命名的作用域
其次从里向外搜索所有父函数的作用域，其中的命名既非局部也非全局
倒数第二个搜索的作用域是包含当前模块全局命名的作用域
最后搜索的作用域是最外面包含内置命名的命名空间

如果一个命名声明为全局的，那么对它的所有引用和赋值会直接搜索包含这个模块全局命名的作用域。如果要重新绑定最里层作用域之外的变量，可以使用 nonlocal 语句；如果不声明为 nonlocal，这些变量将是只读的（对这样的变量赋值会在最里面的作用域创建一个新 的局部变量，外部具有相同命名的那个变量不会改变)。

通常情况下，局部作用域引用当前函数的本地命名。函数之外，局部作用域引用的命名空间与全局作用域相同：模块的命名空间。类定义在局部命名空间中创建了另一个命名空间。

认识到作用域是由代码确定的是非常重要的 :函数的全局作用域是函数的定义所在的模块的命名空间，与函数调用的位置或者别名无关。另一方面，命名的实际搜索过程是动态的，在运行时确定的——然而，Python 语言也在不断发展，以后有可能会成为静态的 “编译” 时确定，所以不要依赖动态解析！(事实上，本地变量是已经确定静态。)

Python 的一个特别之处在于——如果没有使用 global 语法——其赋值操作总是在最里层的作用域。赋值不会复制数据——只是将命名绑定到对象。删除也是如此：del x 只是从局部

作用域的命名空间中删除命名 x。事实上，所有引入新命名的操作都作用于局部作用域： 特别是 import 语句和函数定将模块名或函数绑定于局部作用域。(可以使用 Global 语句将变量引入到全局作用域。)

global 语句可以用来指明某个特定的变量位于全局作用域并且应该在那里重新绑定；nonlocal 语句表示特定的变量位于一个封闭的作用域并且应该在那里重新绑定。

9.2.1. 作用域和命名空间示例

下面这个示例演示如何访问不同作用域和命名空间，以及 global 和 nonlocal 如何影响变量的绑定:

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

示例代码的输出为：

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spams.
In global scope: global spam
```

注意，local 赋值（默认行为）没有改变 scope_testspam 的绑定。nonlocal 赋值改变了 scope_test 对 spam 的绑定，global 赋值改变了模块级别的绑定。

你也可以看到在 global 语句之前没有对 spam 的绑定。

9.3. 初识类

类引入了少量的新语法、三种新对象类型和一些新语义。

9.3.1. 类定义语法

类定义的最简单形式如下所示：

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

类的定义就像函数定义（def 语句），要先执行才能生效。(你当然可以把它放进 if 语句的某一分支，或者一个函数的内部。)

实际应用中，类定义包含的语句通常是函数定义，不过其它语句也是可以的而且有时还会很有用——后面我们会再回来讨论。类中的函数定义通常有一个特殊形式的参数列表，这是由方法调用的协议决定的——同样后面会解释这些。

进入类定义部分后，会创建出一个新的命名空间，作为局部作用域——因此，所有的赋值成为这个新命名空间的局部变量。特别是这里的函数定义会绑定新函数的名字。

类定义正常退出时，一个类对象也就创建了。基本上它是对类定义创建的命名空间进行了一个包装；我们在下一节将进一步学习类对象的知识。原始的局部作用域（类定义引入之前生效的那个）得到恢复，类对象在这里绑定到类定义头部的类名（例子中是 ClassName）。

9.3.2. 类对象

类对象支持两种操作：属性引用和实例化。

属性引用使用的所有属性引用在 Python 中使用的标准语法： obj.name。有效的属性名称是在该类的命名空间中的类对象被创建时的名称。因此，如果类定义看起来像这样：

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

那么 MyClass.i 和 MyClass.f 是有效的属性引用，分别返回一个整数和一个方法对象。也可以对类属性赋值，你可以通过给 MyClass.i 赋值来修改它。__doc__ 也是一个有效的属性，返回类的文档字符串： "A simple example class"。

类的实例化 使用函数的符号。可以假设类对象是一个不带参数的函数，该函数返回这个类的一个新的实例。例如（假设沿用上面的类）：

```
x = MyClass()
```

创建这个类的一个新实例，并将该对象赋给局部变量 x。

实例化操作（“调用” 一个类对象）将创建一个空对象。很多类希望创建的对象可以自定义一个初始状态。因此类可以定义一个名为__init__()的特殊方法，像下面这样：

```
def __init__(self):
    self.data = []
```

当类定义了__init__()方法，类的实例化会为新创建的类实例自动调用__init__()。所以在下面的示例中，可以获得一个新的、已初始化的实例：

```
x = MyClass()
```

当然，__init__()方法可以带有参数，这将带来更大的灵活性。在这种情况下，类实例化操作的参数将传递给__init__()。例如，

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. 实例对象

现在我们可以用实例对象做什么？实例对象唯一可用的操作就是属性引用。有两种有效的属性名：数据属性和方法。

数据属性相当于 Smalltalk 中的"实例变量"或 C++ 中的"数据成员"。数据属性不需要声明；和局部变量一样，它们会在第一次给它们赋值时生成。例如，如果 x 是上面创建的 MyClass 的实例，下面的代码段将打印出值 16 而不会出现错误：

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

实例属性引用的另一种类型是方法。方法是"属于"一个对象的函数。（在 Python，方法这个

术语不只针对类实例：其他对象类型也可以具有方法。例如，列表对象有 append、insert、remove、sort 方法等等。但是在后面的讨论中，除非明确说明，我们提到的方法特指类实例对象的方法。）

实例对象的方法的有效名称依赖于它的类。根据定义，类中所有函数对象的属性定义了其实例中相应的方法。所以在我们的示例中， x.f 是一个有效的方法的引用，因为 MyClass.f 是一个函数，但 x.i 不是，因为 MyClass.i 不是一个函数。但 x.f 与 MyClass.f 也不是一回事 —— 它是一个方法对象，不是一个函数对象。

9.3.4. 方法对象

通常情况下，方法在绑定之后被直接调用：

```
x.f()
```

在 MyClass 的示例中，这将返回字符串'hello world'。然而，也不是一定要直接调用方法：x.f 是一个方法对象，可以存储起来以后调用。例如：

```
xf = x.f
while True:
    print(xf())
```

会不断地打印 hello world。

调用方法时到底发生了什么？你可能已经注意到，上面 x.f()的调用没有参数，即使 f ()函数的定义指定了一个参数。该参数发生了什么问题？当然如果函数调用中缺少参数 Python 会抛出异常——即使这个参数实际上没有使用……

实际上，你可能已经猜到了答案：方法的特别之处在于实例对象被作为函数的第一个参数传给了函数。在我们的示例中，调用 x.f()完全等同于 MyClass.f(x)。一般情况下，以 n 个参数的列表调用一个方法就相当于将方法所属的对象插入到列表的第一个参数的前面，然后以新的列表调用相应的函数。

如果你还是不明白方法的工作原理，了解一下它的实现或许有帮助。引用非数据属性的实例属性时，会搜索它的类。如果这个命名确认为一个有效的函数对象类属性，就会将实例对象和函数对象封装进一个抽象对象：这就是方法对象。以一个参数列表调用方法对象时，它被重新拆封，用实例对象和原始的参数列表构造一个新的参数列表，然后函数对象调用这个新的参数列表。

9.3.5. 类和实例变量

一般来说，实例变量用于对每一个实例都是唯一的数据，类变量用于类的所有实例共享的属性和方法：

```
class Dog:
```

```
    kind = 'canine'                 # class variable shared by all instances

    def __init__(self, name):
        self.name = name        # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                              # shared by all dogs
'canine'
>>> e.kind                              # shared by all dogs
'canine'
>>> d.name                              # unique to d
'Fido'
>>> e.name                              # unique to e
'Buddy'
```

正如在名称和对象讨论的，可变对象，例如列表和字典，的共享数据可能带来意外的效果。例如，下面代码中的 tricks 列表不应该用作类变量，因为所有的 Dog 实例将共享同一个列表：

```
class Dog:

    tricks = []                     # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                            # unexpectedly shared by all dogs
['roll over', 'play dead']
```

这个类的正确设计应该使用一个实例变量：

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []        # creates a new empty list for each dog
```

```
    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4. 补充说明

数据属性会覆盖同名的方法属性；为了避免意外的命名冲突，这在大型程序中可能带来极难发现的 bug，使用一些约定来减少冲突的机会是明智的。可能的约定包括大写方法名称的首字母，使用一个唯一的小写的字符串（也许只是一个下划线）作为数据属性名称的前缀，或者方法使用动词而数据属性使用名词。

数据属性可以被方法引用，也可以由一个对象的普通用户（“客户端”）使用。换句话说，类是不能用来实现纯抽象数据类型的。事实上，Python 中不可能强制隐藏数据——一切基于约定。（另一方面，如果需要，使用 C 编写的 Python 实现可以完全隐藏实现细节并控制对象的访问；这可以用来通过 C 语言扩展 Python。）

客户应该谨慎的使用数据属性——客户可能通过随意使用他们的数据属性而使那些由方法维护的常量变得混乱。注意：只要能避免冲突，客户可以向一个实例对象添加他们自己的数据属性，而不会影响方法的正确性——再次强调，命名约定可以避免很多麻烦。

从方法内部引用数据属性（或其他方法）并没有快捷方式。我觉得这实际上增加了方法的可读性：当浏览一个方法时，在局部变量和实例变量之间不会出现令人费解的情况。

通常，方法的第一个参数称为 self。这仅仅是一个约定：名字 self 对 Python 而言绝对没有任何特殊含义。但是请注意：如果不遵循这个约定，对其他的 Python 程序员而言你的代码可读性就会变差，而且有些类 查看 器程序也可能是遵循此约定编写的。

类属性的任何函数对象都为那个类的实例定义了一个方法。函数定义代码不一定非得定义在类中：也可以将一个函数对象赋值给类中的一个局部变量。例如：

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
```

```
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

现在 f、 g 和 h 都是类 C 中引用函数对象的属性，因此它们都是 c 的实例的方法 —— h 完全等同于 g。请注意，这种做法通常只会使阅读程序的人产生困惑。

方法可以通过使用 self 参数的方法属性，调用其他方法：

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

方法可以像普通函数那样引用全局命名。与方法关联的全局作用域是包含这个方法的定义的模块（类）。（类本身永远不会作为全局作用域使用。）尽管很少有好的理由在方法中使用全局数据，全局作用域确有很多合法的用途：其一是方法可以调用导入全局作用域的函数和方法，也可以调用定义在其中的类和函数。通常，包含此方法的类也会定义在这个全局作用域，在下一节我们会了解为何一个方法要引用自己的类。

每个值都是一个对象，因此每个值都有一个类（也称它的类型）。它存储为 object.__class__。
9.5. 继承

当然，一个语言特性不支持继承是配不上 “类” 这个名字的。派生类定义的语法如下所示：

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

BaseClassName 必须与派生类定义在一个作用域内。用其他任意表达式代替基类的名称也是允许的。这可以是有用的，例如，当基类定义在另一个模块中时：

```
class DerivedClassName(modname.BaseClassName):
```

派生类定义的执行过程和基类是相同的。类对象创建后，基类会被保存。这用于解析属性的引用：如果在类中找不到请求的属性，搜索会在基类中继续。如果基类本身是由别的类派生

而来，这个规则会递归应用。

派生类的实例化没有什么特殊之处：DerivedClassName()创建类的一个新的实例。方法的引用按如下规则解析： 搜索对应的类的属性，必要时沿基类链逐级搜索，如果找到了函数对象这个方法引用就是合法的。

派生的类可能重写其基类的方法。因为方法调用同一个对象中的其它方法时没有特权，基类的方法调用同一个基类的方法时，可能实际上最终调用了派生类中的覆盖方法。(对于 C++ 程序员：Python 中的所有方法实际上都是虚的。)

派生类中的覆盖方法可能是想要扩充而不是简单的替代基类中的重名方法。有一个简单的方法可以直接调用基类方法：只要调用 BaseClassName.methodname(self, arguments)。有时这对于客户端也很有用。(要注意只有 BaseClassName 在同一全局作用域定义或导入时才能这样用。)

Python 有两个用于继承的函数：

使用 isinstance()来检查实例类型 :isinstance(obj, int)只有 obj.__class__是 int 或者是从 int 派生的类时才为 True。

使用 issubclass()来检查类的继承: issubclas (bool， int)是 True 因为 bool 是 int 的子类。然而， issubclass(float, int)为 False， 因为 float 不是 int 的子类。

9.5.1. 多继承

Python 也支持一种形式的多继承。具有多个基类的类定义如下所示：

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

对于大多数用途，在最简单的情况下，你可以认为继承自父类的属性搜索是从左到右的深度优先搜索，不会在同一个类中搜索两次，即使层次会有重叠。因此，如果在 DerivedClassName 中找不到属性，它搜索 Base1，然后(递归)基类中的 Base1，如果没有找到，它会搜索 base2，依此类推。

事实要稍微复杂一些；为了支持合作调用 super()，方法解析的顺序会动态改变。这种方法在某些其它多继承的语言中也有并叫做 call-next-method，它比单继承语言中的 super 调用更强大。

动态调整顺序是必要的，因为所有的多继承都会有一个或多个菱形关系(从最底部的类向上，至少会有一个父类可以通过多条路径访问到)。例如，所有的类都继承自 object，所以任何

多继承都会有多条路径到达 object。为了防止基类被重复访问，动态算法线性化搜索顺序，每个类都按从左到右的顺序特别指定了顺序，每个父类只调用一次，这是单调的（也就是说一个类被继承时不会影响它祖先的次序）。所有这些特性使得设计可靠并且可扩展的多继承类成为可能。有关详细信息，请参阅 https://www.python.org/download/releases/2.3/mro/。

9.6. 私有变量

在 Python 中不存在只能从对象内部访问的 “私有” 实例变量。然而，有一项大多数 Python 代码都遵循的公约：带有下划线（例如_spam）前缀的名称应被视为非公开的 API 的一部分（无论是函数、 方法还是数据成员）。它应该被当做一个实现细节，将来如果有变化恕不另行通知。

因为有一个合理的类私有成员的使用场景（即为了避免名称与子类定义的名称冲突），Python 对这种机制有简单的支持，叫做 name mangling。__spam 形式的任何标识符(前面至少两个下划线，后面至多一个下划线）将被替换为_classname__spam，classname 是当前类的名字。此重整无需考虑该标识符的句法位置，只要它出现在类的定义的范围内。

Name mangling 有利于子类重写父类的方法而不会破坏类内部的方法调用。例如：

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

请注意名称改编的目的主要是避免发生意外；访问或者修改私有变量仍然是可能的。这在特殊情况下，例如调试的时候，还是有用的。

注意传递给 exec 或 eval()的代码没有考虑要将调用类的类名当作当前类；这类似于 global 语句的效果，影响只限于一起进行字节编译的代码。相同的限制适用于 getattr()、 setattr() 和 delattr()，以及直接引用__dict__时。

9.7. 零碎的说明

有时候类似于 Pascal 的"record" 或 C 的"struct"的数据类型很有用，它们把几个已命名的数据项目绑定在一起。一个空的类定义可以很好地做到：

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

某一段 Python 代码需要一个特殊的抽象数据结构的话，通常可以传入一个类来模拟该数据类型的方法。例如，如果你有一个用于从文件对象中格式化数据的函数，你可以定义一个带有 read ()和 readline () 方法的类，以此从字符串缓冲读取数据，然后将该类的对象作为参数传入前述的函数。

实例的方法对象也有属性：m.__self__是具有方法 m()的实例对象，m.__func__是方法的函数对象。

9.8. 异常也是类

用户定义的异常类也由类标识。利用这个机制可以创建可扩展的异常层次。

raise 语句有两种新的有效的（语义上的）形式：

```
raise Class

raise Instance
```

第一种形式中，Class 必须是 type 或者它的子类的一个实例。第一种形式是一种简写：

```
raise Class()
```

except 子句中的类如果与异常是同一个类或者是其基类，那么它们就是相容的（但是反过来是不行的——except 子句列出的子类与基类是不相容的）。例如，下面的代码将按该顺序打印 B、 C、 D：

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
```

```
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

请注意，如果 except 子句的顺序倒过来（excpet B 在最前面），它就会打印 B，B，B —— 第一个匹配的异常被触发。

打印一个异常类的错误信息时，先打印类名，然后是一个空格、一个冒号，然后是用内置函数 str()将类转换得到的完整字符串。

9.9. 迭代器

现在你可能注意到大多数容器对象都可以用 for 遍历：

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

这种访问风格清晰、 简洁又方便。迭代器的用法在 Python 中普遍而且统一。在后台， for 语句调用传入了容器对象的 iter() 。该函数返回一个定义了__next__()方法的迭代器对象，它在容器中逐一访问元素。没有后续的元素时，__next__()会引发 StopIteration 异常，告诉 for 循环终止。你可以是用内建的 next()函数调用__next__()方法；此示例显示它是如何工作：

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
```

```
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    next(it)
StopIteration
```

看过迭代器协议背后的机制后，将很容易将迭代器的行为添加到你的类中。定义一个__iter__()方法，它使用__next__()方法返回一个对象。如果类定义了__next__()，__iter__()可以只返回 self：

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.10. 生成器

生成器是简单且功能强大的工具，用于创建迭代器。它们写起来就像是正规的函数，需要返回数据的时候使用 yield 语句。每次在它上面调用 next()时，生成器恢复它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）。以下示例演示了生成器可以非常简单地创建出来：

```
def reverse(data):
```

```
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...         print(char)
...
f
l
o
g
```

生成器能做到的什么事，前一节所述的基于类的迭代器也能做到。生成器这么简洁是因为__iter__()和__next__()方法是自动创建的。

另一个关键特征是在调用中本地变量和执行状态会自动保存，这使得该函数更容易写，也比使用实例变量的方法，如 self.index 和 self.data 更清晰。

除了自动创建方法和保存程序的状态，当生成器终止时，它们会自动引发 StopIteration。结合这些特点，创建迭代器就和写一个普通函数一样简单

9.11. 生成器表达式

一些简单的生成器可以简洁地使用表达式，语法类似于列表格式，但用圆括号()而不是方括号[]。这些表达式用于闭包函数马上使用生成器的情况。相对于完整的生成器定义，生成器表达式可读性差但是更加紧凑，而且比等价的列表推导式更加的内存友好。

例子：

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

>>> unique_words = set(word  for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
```

['f', 'l', 'o', 'g']

脚注

[1] Except for one thing. Module objects have a secret read-only attribute called __dict__ which returns the dictionary used to implement the module' s namespace; the name __dict__ is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

10. 标准库概览
10.1. 操作系统接口

os 模块提供了几十个函数与操作系统交互：

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\\Python34'
>>> os.chdir('/server/accesslogs')     # Change current working directory
>>> os.system('mkdir today')     # Run the command mkdir in the system shell
0
```

一定要使用 import os 的形式而不要用 from os import *。这将使用 os.open()从而避免屏蔽内置的 open()函数，它们的功能完全不同。

内置的 dir()和 help()函数对于使用像 os 这样的大型模块可以提供非常有用的交互式帮助：

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

对于日常的文件和目录管理任务，shutil 模块提供了一个易于使用的高级接口：

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2. 文件通配符

glob 模块提供了一个函数用于在目录中以通配符搜索文件，并生成匹配的文件列表：

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3. 命令行参数

常见的实用程序脚本通常需要处理命令行参数。这些参数以一个列表存储在 sys 模块的 argv 属性中。例如下面的输出结果来自于从命令行运行 python demo.py one two three：

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

getopt 模块使用 Unix getopt()函数的约定处理 sys.argv。argparse 模块提供更强大、 更灵活的命令行处理功能。

10.4. 错误输出重定向和程序终止

sys 模块还具有 stdin、stdout 和 stderr 属性。即使在 stdout 被重定向时，后者也可以用于显示警告和错误信息：

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

终止脚本最直接的方法是使用 sys.exit()。

10.5. 字符串模式匹配

re 模块为高级的字符串处理提供了正则表达式工具。对于复杂的匹配和操作，正则表达式提供了简洁、优化的解决方案：

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

当只需要简单的功能时，最好使用字符串方法，因为它们更容易阅读和调试：

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6. 数学

math 模块为浮点运算提供了对底层 C 函数库的访问：

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

random 模块提供了进行随机选择的工具：

```
>>> import random
```

```
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

SciPy 项目<http://scipy.org>有很多其它用于数值计算的模块。

10.7. 互联网访问

有很多的模块用于访问互联网和处理的互联网协议。最简单的两个是从 URL 获取数据的 urllib.request 和发送邮件的 smtplib：

```
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8')  # Decoding the binary data to text.
...         if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...             print(line)

<BR>Nov. 25, 09:43:32 PM EST
```

```
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

（请注意第二个示例需要在本地主机上运行邮件服务器）。

10.8. 日期和时间

datetime 模块提供了处理日期和时间的类，既有简单的方法也有复杂的方法。支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。该模块还支持处理时区。

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
```

```
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9. 数据压缩

常见的数据打包和压缩格式有模块直接支持，包括：zlib, gzip, bz2, lzma, zipfile 和 tarfile。

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10. 性能度量

一些 Python 用户对同一问题的不同解决方法之间的性能差异深有兴趣。Python 提供了的一个度量工具可以立即解决这些问题。

例如，使用元组封装和拆封功能而不是传统的方法来交换参数可能会更吸引人。timeit 模块快速展示了一个温和的性能优势

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

与 timeit 的精细的粒度相反，profile 和 pstats 模块提供了针对更大代码块的时间度量工具。

10.11. 质量控制

开发高质量软件的方法之一是为每一个函数编写测试代码，并且在开发过程中经常性的运行

这些测试代码。

doctest 模块提供一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。测试的构造像一个把结果剪切并粘贴到文档字符串的典型调用一样简单。通过用户提供的例子，它发展了文档，允许 doctest 模块确认代码的结果是否与文档一致：

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()    # automatically validate the embedded tests
```

unittest 模块不像 doctest 模块那样容易，不过它可以在一个独立的文件里提供一个更全面的测试集：

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

10.12. Batteries Included

Python 有"Batteries Included"的哲学。这最好是通过其较大的文件包的先进和强大功能。例如：

xmlrpc.client 和 xmlrpc.server 模块让远程过程调用变得轻而易举。尽管模块有这样的名字，它不需要直接 XML 知识或处理 XML 。

email 包是是一个处理电子邮件的库，包括 MIME 和其它基于 RFC 2822 的邮件。与 smtplib 和 poplib 用于实际发送和接收邮件，email 包有一个完整的工具集用于构建或者解码复杂邮件结构（包括附件），并实现互联网编码和头协议。

xml.dom 和 xml.sax 的包为这种流行的数据交换格式提供了强大的支持。同样，csv 模块支持以常见的数据库格式直接读取和写入。这些模块和包一起大大简化了 Python 应用程序和其他工具之间的数据交换。

国际化支持模块包括 gettext、locale 和 codecs 包。

11. 标准库概览 — 第 II 部分

第二部分提供了更高级的模块用来支持专业编程的需要。这些模块很少出现在小型的脚本里。

11.1. 输出格式

reprlib 模块提供一个定制版的 repr()用于显示大型或者深层嵌套容器：

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

pprint 模块提供更复杂的打印控制，以解释器可读的方式打印出内置对象和用户定义的对象。当结果超过一行时，这个"漂亮的打印机"将添加分行符和缩进，以更清楚地显示数据结构：

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...         'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

textwrap 模块格式化文本段落以适应设定的屏宽：

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

locale 模块会访问区域性特定数据格式的数据库。分组属性的区域设置的格式函数的格式设置的数字以直接的方式提供了组分隔符：

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
```

```
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2. 模板

string 模块包括一个通用 Template 类，它用简化的语法适合最终用户编辑。这允许用户自定义他们的应用程序无需修改应用程序。

这种格式使用的占位符名称由$与有效的 Python 标识符（字母数字字符和下划线）组成。周围的大括号与占位符允许它应遵循的更多字母数字字母并且中间没有空格。$$创建一个转义的$：

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

当字典或关键字参数中没有提供占位符时，substitute()方法将引发 KeyError。对于邮件-合并风格的应用程序，用户提供的数据可能不完整，这时 safe_substitute()方法可能会更合适 —— 如果没有数据它将保持占位符不变：

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template 类的子类可以指定自定义的分隔符。例如，图像浏览器的批量命名工具可能选用百分号作为表示当前日期、图像 序列号或文件格式的占位符：

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
```

```
Enter rename style (%d-date %n-seqnum %f-format):    Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...         base, ext = os.path.splitext(filename)
...         newname = t.substitute(d=date, n=i, f=ext)
...         print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

模板的另一个应用是把多样的输出格式细节从程序逻辑中分类出来。这使它能够替代用户的 XML 文件、 纯文本报告和 HTML 网页报表。

11.3. 二进制数据记录格式

The struct module provides pack() and unpack() functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the zipfile module. Pack codes "H" and "I" represent two and four byte unsigned numbers respectively. The "<" indicates that they are standard size and in little-endian byte order:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                                  # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size         # skip to the next header
```

11.4. 多线程

线程是一种解耦非顺序依赖任务的技术。线程可以用来提高接应用程序受用户输入的响应速

度，而其他任务同时在后台运行。一个相关的使用场景是 I/O 操作与另一个线程中的计算并行执行。

下面的代码演示在主程序连续运行的同时，threading 模块如何在后台运行任务：

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

多线程应用程序的最主要挑战是协调线程间共享的数据或其他资源。为此目的，该线程模块提供了许多同步原语包括锁、 事件、 条件变量和信号量。

尽管这些工具很强大，很小的设计错误也可能导致很难复现的问题。因此，任务协调的首选方法是把对一个资源的所有访问集中在一个单独的线程中，然后使用 queue 模块用那个线程服务其他线程的请求。应用程序使用 Queue 对象进行线程间的通信和协调将更容易设计、更具可读性和更可靠。

11.5. 日志

logging 模块提供了一个具有完整功能并且非常灵活的日志系统。最简单的，发送消息到一个文件或者 sys.stderr：

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

这将生成以下输出：

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

默认情况下，信息和调试消息被压制并输出到标准错误。其他输出选项包括将消息通过 email、datagrams、sockets 发送，或者发送到 HTTP 服务器。根据消息的优先级，新的过滤器可以选择不同的方式：DEBUG、INFO、WARNING、ERROR 和 CRITICAL。

日志系统可以直接在 Python 代码中定制，也可以不经过应用程序直接在一个用户可编辑的配置文件中加载。

11.6. 弱引用

Python 会自动进行内存管理（对大多数的对象进行引用计数和垃圾回收以循环利用）。在最后一个引用消失后，内存会立即释放。

这个方式对大多数应用程序工作良好，但是有时候会需要跟踪对象，只要它们还被其它地方所使用。不幸的是，只是跟踪它们也会创建一个引用，这将使它们永久保留。weakref 模块提供工具用来无需创建一个引用跟踪对象。当不再需要该对象时，它会自动从 weakref 表中删除并且会为 weakref 对象触发一个回调。典型的应用包括缓存创建的时候需要很大开销的对象：

```
>>> import weakref, gc
>>> class A:
...         def __init__(self, value):
...                 self.value = value
...         def __repr__(self):
...                 return str(self.value)
...
>>> a = A(10)                          # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a                   # does not create a reference
>>> d['primary']                       # fetch the object if it is still alive
10
>>> del a                              # remove the one reference
>>> gc.collect()                       # run garbage collection right away
0
>>> d['primary']                       # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                      # entry was automatically removed
  File "C:/python34/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
```

KeyError: 'primary'

11.7. 列表工具

很多数据结构使用内置列表类型就可以满足需求。然而，有时需要其它具有不同性能的替代实现。
The array module provides an array() object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

collections 模块提供了一个 deque()对象，就像一个列表,不过它从左边添加和弹出更快，但是在内部查询更慢。这些对象非常实现队列和广度优先的树搜索：

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

除了列表的替代实现，该库还提供了其它工具例如 bisect 模块中包含处理排好序的列表的函数：

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

heapq 模块提供的函数可以实现基于常规列表的堆。最小的值总是保持在第零个位置。这对循环访问最小元素，但是不想运行完整列表排序的应用非常有用：

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]   # fetch the three smallest entries
[-5, 0, 1]
```

11.8. 十进制浮点数运算

decimal 模块提供一个 Decimal 数据类型用于为十进制浮点运算。相比二进制浮点数内置的 float 实现，这个类对于以下情形特别有用：

财务应用程序和其他用途，需要精确的十进制表示形式，
控制精度，
对符合法律或法规要求，舍入的控制
跟踪有效小数位
用户希望计算结果与手工计算相符的应用程序。

例如，计算上 70%电话费的 5%税给不同的十进制浮点和二进制浮点结果。区别变得明显如果结果舍入到最接近的分:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

Decimal 的结果总是保有结尾的 0，自动从两位精度延伸到 4 位。Decimal 类似手工完成的数学运算，这就避免了二进制浮点数无法精确表达数据精度产生的问题。

精确地表示允许 Decimal 可以执行二进制浮点数无法进行的模运算和等值测试：

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

decimal 模块提供任意精度的运算：

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

12.现在怎么办?

阅读本教程可能让你对使用 Python 更感兴趣了——你应该会渴望将 Python 应用于解决实际问题。你应该到哪里去了解更多 Python 的内容呢？

本教程是 Python 文档集的一部分。文档集中的一些其它文件有：

Python 标准库：

你应该浏览本手册，它给出了标准库中关于类型、 函数和模块的完整（虽然简洁）的参考资料。标准的 Python 发布包含大量 的附加模块。其中有读取 Unix 邮箱、收取 HTTP 文档、生成随机数、解析命令行选项、编写 CGI 程序、压缩数据以及很多其它任务的模块。浏览一下这个库参考手册会让你知道有什么是现成可用的。

安装 Python 模块 解释如何安装由其他 Python 用户编写的外部模块。

Python 语言参考：详细地讲述了 Python 的语法和语义。它读起来很难，但是作为语言本身的完整指南非常有用。

更多的 Python 资源：

https://www.python.org：主要的 Python Web 站点。它包含代码、文档和网上 Python 相关页面的链接。该网站在世界各地都有镜像，如欧洲、 日本和澳大利亚；镜像可能会比主站快，这取决于你的地理位置。

https://docs.python.org：快速访问 Python 的文档。

https://pypi.python.org/pypi： Python 包索引，以前的绰号叫奶酪店，是用户创建的 Python 模块的索引，这些模块可供下载。一旦你开始发布代码，你可以在这里注册你的代码这样其他人可以找到它。

http://code.activestate.com/recipes/langs/python/: 这本 Python 食谱收集了相当多的代码示例、大型的模块，以及有用的脚本。其中尤其显著的贡献被收集成一书，这本书也叫做 Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3)。

http://scipy.org：The Scientific Python 项目包括数组快速计算和处理模块，和大量线性代数、傅里叶变换、非线性 solvers、随机数分布，统计分析以及类似的包。

Python 相关的问题和问题报告，你可以发布到新闻组 comp.lang.python，或将它们发送到邮件列表 python-list@python.org。新闻组和邮件列表是互通的，因此发布到其中的一个消息将自动转发给另外一个。一天大约有 120 个帖子 (最高峰到好几百)，包括询问 (和回答) 问题，建议新的功能和宣布新的模块。在发帖之前，一定要检查常见问题（也称为 FAQ）的列表。可在 https://mail.python.org/pipermail/查看邮件列表的归档。FAQ 回答了很多经常出现的问题，可能已经包含你的问题的解决方法。

13. 交互式输入的编辑和历史记录

某些版本的 Python 解释器支持编辑当前的输入行和历史记录，类似于在 Korn shell 和 GNU Bash shell 中看到的功能。这是使用 GNU Readline 库实现的，它支持各种编辑风格。这个库有它自己的文档，在这里我们就不重复了。

13.1. Tab 补全和历史记录

变量和模块名的补全在解释器启动时自动打开以便 Tab 键调用补全功能；它会查看 Python 语句的名字，当前局部变量以及可以访问的模块名。对于点分表达式如 string.a，它将求出表达式最后一个'.'之前的值，然后根据结果的属性给出补全的建议。注意，如果一个具有__getattr__()方法的对象是表达式的某部分，这可能执行应用程序定义的代码。默认的配置同时会把历史记录保存在你的用户目录下一个名为.python_history 的文件中。在下次与交互式解释器的回话中，历史记录将还可以访问。

13.2. 其它交互式解释器

与早期版本的解释器相比，现在是向前巨大的进步；然而，有些愿望还是没有实现：如果能对连续的行给出正确的建议就更好了（解析器知道下一行是否需要缩进）。补全机制可以使用解释器的符号表。检查（或者只是建议）匹配的括号、 引号的命令等也会非常有用。

一个增强的交互式解释器是 IPython，它已经存在相当一段时间，具有 tab 补全、 对象 exploration 和高级的历史记录功能。它也可以彻底定制并嵌入到其他应用程序中。另一个类似的增强的交互式环境是 bpython。

14. 浮点数运算：问题和局限

浮点数在计算机硬件中表示为以 2 为底（二进制）的小数。例如，十进制小数

0.125

是 1/10 + 2/100 + 5/1000 的值，同样二进制小数

0.001

是 0/2 + 0/4 + 1/8 的值。这两个小数具有相同的值，唯一真正的区别是，第一个小数是十进制表示法，第二个是二进制表示法。

不幸的是，大多数十进制小数不能完全用二进制小数表示。结果是，一般情况下，你输入的十进制浮点数仅由实际存储在计算机中的近似的二进制浮点数表示。

这个问题在十进制情况下很容易理解。考虑分数 1/3，你可以用十进制小数近似它：

0.3

或者更接近的

0.33

或者再接近一点的

0.333

等等。无论你愿意写多少位数字，结果永远不会是精确的 1/3，但将会越来越好地逼近 1/3。

同样地，无论你使用多少位的二进制数，都无法确切地表示十进制值 0.1。1/10 用二进制表示是一个无限循环的小数。

0.0001100110011001100110011001100110011001100110011...

在任何有限数量的位停下来，你得到的都是近似值。今天在大多数机器上，浮点数的近似使用的小数以最高的 53 位为分子，2 的幂为分母。至于 1/10 这种情况，其二进制小数是 3602879701896397 / 2 ** 55，它非常接近但不完全等于 1/10 真实的值。

由于显示方式的原因，许多使用者意识不到是近似值。Python 只打印机器中存储的二进制值的十进制近似值。在大多数机器上，如果 Python 要打印 0.1 存储的二进制的真正近似值，将会显示

>>> 0.1

0.1000000000000000055511151231257827021181583404541015625

这么多位的数字对大多数人是没有用的，所以 Python 显示一个舍入的值

```
>>> 1 / 10
0.1
```

只要记住即使打印的结果看上去是精确的 1/10，真正存储的值是最近似的二进制小数。

例如，数字 0.1、0.10000000000000001 和 0.1000000000000000055511151231257827021181583404541015625 都以 3602879701896397 / 2 ** 55 为近似值。因为所有这些十进制数共享相同的近似值，在保持恒等式 eval(repr(x)) == x 的同时，显示的可能是它们中的任何一个。

现在从 Python 3.1 开始，Python（在大多数系统上）能够从这些数字当中选择最短的一个并简单地显示 0.1。

注意，这是二进制浮点数的自然性质：它不是 Python 中的一个 bug，也不是你的代码中的 bug。你会看到所有支持硬件浮点数算法的语言都会有这个现象（尽管有些语言默认情况下或者在所有输出模式下可能不会显示 出差异）。

为了输出更好看，你可能想用字符串格式化来生成固定位数的有效数字：

```
>>> format(math.pi, '.12g')  # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

认识到这，在真正意义上，是一种错觉是很重要的：你在简单地舍入真实机器值的显示。

例如，既然 0.1 不是精确的 1/10，3 个 0.1 的值相加可能也不会得到精确的 0.3：

```
>>> .1 + .1 + .1 == .3
False
```

另外，既然 0.1 不能更接近 1/10 的精确值而且 0.3 不能更接近 3/10 的精确值，使用 round() 函数提前舍入也没有帮助：

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

虽然这些数字不可能再更接近它们想要的精确值，round()函数可以用于在计算之后进行舍入，这样的话不精确的结果就可以和另外一个相比较了：

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

二进制浮点数计算有很多这样意想不到的结果。“0.1” 的问题在下面"误差的表示"一节中有准确详细的解释。更完整的常见怪异现象请参见浮点数的危险。

最后我要说，“没有简单的答案”。也不要过分小心浮点数！Python 浮点数计算中的误差源之于浮点数硬件，大多数机器上每次计算误差不超过 2**53 分之一。对于大多数任务这已经足够了，但是你要在心中记住这不是十进制算法，每个浮点数计算可能会带来一个新的舍入错误。

虽然确实有问题存在，对于大多数平常的浮点数运算，你只要简单地将最终显示的结果舍入到你期望的十进制位数，你就会得到你期望的最终结果。str()通常已经足够用了，对于更好的控制可以参阅格式化字符串语法中 str.format()方法的格式说明符。

对于需要精确十进制表示的情况，可以尝试使用 decimal 模块，它实现的十进制运算适合会计方面的应用和高精度要求的应用。

fractions 模块支持另外一种形式的运算，它实现的运算基于有理数（因此像 1/3 这样的数字可以精确地表示）。

如果你是浮点数操作的重度使用者，你应该看一下由 SciPy 项目提供的 Numerical Python 包和其它用于数学和统计学的包。参看<http://scipy.org>。

当你真的真想要知道浮点数精确值的时候，Python 提供这样的工具可以帮助你。float.as_integer_ratio()方法以分数的形式表示一个浮点数的值：

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

因为比值是精确的，它可以用来无损地重新生成初始值：

```
>>> x == 3537115888337719 / 1125899906842624
True
```

float.hex()方法以十六进制表示浮点数，给出的同样是计算机存储的精确值：

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

精确的十六进制表示可以用来准确地重新构建浮点数：

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

因为可以精确表示，所以可以用在不同版本的 Python（与平台相关）之间可靠地移植数据以及与支持同样格式的其它语言（例如 Java 和 C99）交换数据。

另外一个有用的工具是 math.fsum()函数，它帮助求和过程中减少精度的损失。当数值在不停地相加的时候，它会跟踪“丢弃的数字”。这可以给总体的准确度带来不同，以至于错误不会累积到影响最终结果的点：

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

14.1. 二进制表示的误差

这一节将详细解释“0.1”那个示例，并向你演示对于类似的情况自己如何做一个精确的分析。假设你已经基本了解浮点数的二进制表示。

二进制表示的误差 指的是这一事实，一些（实际上是大多数）十进制小数不能精确地用二进制小数表示。这是为什么 Python（或者 Perl、 C、 C++、 Java、 Fortran 和其他许多语言）通常不会显示你期望的精确的十进制数的主要原因。

这是为什么？1/10 和 2/10 不能用二进制小数精确表示。今天（2010 年 7 月）几乎所有的机器都使用 IEEE-754 浮点数算法，几乎所有的平台都将 Python 的浮点数映射成 IEEE-754“双精度浮点数”。754 双精度浮点数包含 53 位的精度，所以输入时计算机努力将 0.1 转换为最接近的 J/2**N 形式的小数，其中 J 是一个 53 位的整数。改写

```
1 / 10 ~= J / (2**N)
```

为

```
J ~= 2**N / 10
```

回想一下 J 有 53 位（>= 2**52 但< 2**53），所以 N 的最佳值是 56：

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

即 56 是 N 保证 J 具有 53 位精度的唯一可能的值。J 可能的最佳值是商的舍入：

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

由于余数大于 10 的一半，最佳的近似值是向上舍入：

```
>>> q+1
7205759403792794
```

因此在 754 双精度下 1/10 的最佳近似是：

```
7205759403792794 / 2 ** 56
```

分子和分母都除以 2 将小数缩小到：

```
3602879701896397 / 2 ** 55
```

请注意由于我们向上舍入，这其实有点大于 1/10；如果我们没有向上舍入，商数就会有点小于 1/10。但在任何情况下它都不可能是精确的 1/10!

所以计算机从来没有"看到"1/10： 它看到的是上面给出的精确的小数，754 双精度下可以获得的最佳的近似了：

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

如果我们把这小数乘以 10**55，我们可以看到其 55 位十进制数的值：

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

也就是说存储在计算机中的精确数字等于十进制值 0.1000000000000000055511151231257827021181583404541015625。许多语言（包括旧版本的 Python）会把结果舍入到 17 位有效数字，而不是显示全部的十进制值：

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

fractions 和 decimal 模块使得这些计算很简单：

```
>>> from decimal import Decimal
>>> from fractions import Fraction
```

```
>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```