Python 手册

Guido van Rossum

Fred L. Drake, Jr., editor

PythonLabs

Email: python-docs@python.org

Release 2.3

July 29, 2003

前言

前言

概要:

Python 是一种容易学习的强大语言。 它包括了高效的高级数据结构，提供了一个简单但很有有效的方式以便 进行面向对象编程。Python 优雅的语法，动态数据类型，以及它的解释器，使其成为了大多数平台上应用于各领域理想的脚本语言以及开发环境。

Python 解释器及其扩展标准库的源码和编译版本可以从 Python 的 Web 站点 http://www.python.org/及其所有镜像站上免费获得，并且可以自由发布。该站点上也提供了 Python 的一些第三方模块，程序，工具，以及附加的文档。

Python 的解释器很容易通过 C 或 C++（或者其它可以由 C 来调用的语言）来实现功能和数据结构的扩展。因些 Python 也很适于做为定制应用的一种扩展语言。

这个手册介绍了一些 Python 语言及其系统的基本知识与根念。这有助于对 Python 有一个基本的认识，当然所有的例子都包括在里面了，所以这本手册很适合离线阅读。

需要有关标准对象和模块的详细介绍的话，请查询 *Python 程序库参考手册* 文档。*Python 参考手册* 提供了更多的关于语言方面的正式说明。需要编写 C 或 C++扩展，请阅读 *Python 解释器的扩展和集成* 以及 *Python/C API 参考手册*。 这几本书涵盖了各个深度上的 Python 知识。

本手册不会涵盖 Python 的所有功能，也不会去解释所用到的所有相关的知识。相反，它介绍了许多 Python 中最引人注目的功能，这会对读者掌握这门语言的风格大有帮助。读过它后，你应该可以阅读和编写 Python 模块和程序了，接下来你可以从 *Python 库参考手册*中进一步学习 Python 复杂多变的库和模块了。

1. 开胃菜

如果你写过大规模的 Shell 脚本，应该会有过这样的体会：你还非常想再加一些别的功能进去，但它已经太大、太慢、太复杂了；或者这个功能需要调用一个系统函数，或者它只适合通过 C 来调用……通常这些问题还不足以严肃到需要用 C 重写这个 Shell；可能这个功能需要一些类似变长字符串或其它一些在 Shell 脚本中很容易找到的数据类型（比如文件名的有序列表），但它们用 C 来实现就要做大量的工作，或者，你对 C 还不是很熟悉。

另一种情况：可能你需要使用几个 C 库来工作，通常 C 的编写/编译/测试/重编译周期太慢。你需要尽快的开发软件。也许你需要写一个使用扩展语言的程序，但不想设计一个语言，并为此编写调试一个解释器，然后再把它集成进你的程序。

遇到以上情况，Python 可能就是你要找的语言。Python 很容易上手，但它是一门真正的编程语言，提供了比 Shell 多的多的结构，支持大型程序。另一方面，它提供了比 C 更多的错误检查，并且，做为一门高级语言，它拥有内置的高级数据类型，例如可变数组和字典，如果通过 C 来实现的话，这些工作可能让你大干上几天的时间。因为拥有更多的通用数据类型，Python 适合比 Awk 甚至 Perl 更广泛的问题领域，在其它的很多领域，Python 至少比别的语言要易用得多。

Python 可以让你把自己的程序分隔成不同的模块，这样就可以在其它的 Python 程序中重用。这样你就可以让自己的程序基于一个很大的标准模块集或者用它们做为示例来学习 Python 编程。Python 中集成了一些类似文件 I/O，系统调用，sockets，甚至像 Tk 这样的用户图形接口。

Python 是一门解释型语言，因为不需要编译和链接的时间，它可以帮你省下一些开发时间。解释器可以交互式使用，这样就可以很方便的测试语言中的各种功能，以便于编写发布用的程序，或者进行自下而上的开发。还可以当它是一个随手可用的计算器。

Python 可以写出很紧凑和可读性很强的程序。用 Python 写的程序通常比同样的 C 或 C++ 程序要短得多，这是因为以下几个原因：

- 高级数据结构使你可以在一个单独的语句中表达出很复杂的操作；
- 语句的组织依赖于缩进而不是 begin/end 块；
- 不需要变量或参数声明。

Python 是*可执行*的：如果你会用 C 语言写程序，那就可以很容易的为解释器添加新的集成模块和功能，或者优化瓶颈，使其达到最大速度，或者使 Python 能够链接到所需的二进制架构上（比如某个专用的商业图形库）。等你真正熟悉这一切了，你就可以把 Python 集成进由 C 写成的程序，把 Python 当做这个程序的扩展或命令行语言。

顺便说一下，这个语言的名字来源于 BBC 的“Monty Python's Flying Circus”节目，和凶猛的爬虫没有任何关系。在文档中引用 Monty Python 典故不仅是允许的，而且还受到鼓励！

现在你已经了解了 Python 中所有激动人心的东西，大概你想仔细的试试它了。学习一门语言最好的办法就是使用它，你会很乐于这样做。

下一节中，我们会很机械的说明解释器的用法。这没有什么神秘的，不过有助于我们练习后面展示的例子。

本指南其它部分通过例子介绍了 Python 语言和系统的各种功能，开始是简单表达式、语法和数据类型，接下来是函数和模块，最后是诸如异常和自定义类这样的高级内容。

2. 使用 Python 解释器

2.1 调用解释器

通常 Python 的解释器被安装在目标机器的 /usr/local/bin/python 目录下；把 /usr/local/bin 目录放进你的 UNIX Shell 的搜索路径里，确保它可以通过输入

```
python
```

来启动。因为安装路径是可选的，所以也有可能安装在其它位置，你可以与安装 Python 的用户或系统管理员联系。（例如，/usr/local/python 就是一个很常见的选择）

输入一个文件结束符（UNIX 上是 **Ctrl+D** ，Windwos 上是 **Ctrl+Z**）解释器会以 0 值退出（就是说，没有什么错误，正常退出——译者）。如果这没有起作用，你可以输入以下命令退出：“`import sys; sys.exit()`”。

解释器的行编辑功能并不很复杂。装在 Unix 上的解释器可能会有 GNU readline 库支持，这样就可以额外得到精巧的交互编辑和历史记录功能。可能检查命令行编辑器支持能力最方便的方式是在主提示符下输入 Ctrl+P。如果有嘟嘟声（计算机扬声器），说明你可以使用命令行编辑功能，从附录 A 可以查到快捷键的介绍。如果什么也没有发声，或者 P 显示了出来，说明命令行编辑功能不可用，你只有用退格键删掉输入的命令了。

解释器的操作有些像 Unix Shell：使用终端设备做为标准输入来调用它时，解释器交互的解读和执行命令，通过文件名参数或以文件做为标准输入设备时，它从文件中解读并执行*脚本*。

启动解释器的第三个方法是“**python -c** *command* [arg] ...”，这种方法可以在命令行中直接执行语句，等同于 Shell 的 -c 选项。因为 Python 语句通常会包括空格之类的特殊字

符，所以最好把整个语句用双引号包起来。

注意“python file”和“python <file”是有区别的。对于后一种情况，程序中类似于调用 input() 和 raw_input()这样的输入请求，来自于确定的文件。因为在解析器开始执行之前，文件已经完全读入，所以程序指向文件尾。在前一种情况（这通常是你需要的）它们从来自于任何联接到 Python 解释器的标准输入，无论它们是文件还是其它设备。

使用脚本文件时，经常会运行脚本然后进入交互模式。这也可以通过在脚本之前加上**-i** 参数来实现。（如果脚本来自标准输入，就不能这样运行，与前一段提到的原因一样。）

2.1.1 参数传递

调用解释器时，脚本名和附加参数之传入一个名为 sys.argv 的字符串列表。没有脚本和参数时，它至少也有一个元素：sys.argv[0]此时为空字符串。脚本名指定为‘ - ’（表示标准输入）时，sys.argv[0]被设置为‘ - ’，使用 **-c** 指令时，sys.argv[0]被设定为‘ -c ’。 **-c** 命令之后的参数不会被 Python 解释器的选项处理机制所截获，而是留在 sys.argv 中，供脚本命令操作。

2.1.2 交互模式

从 tty 读取命令时，我们称解释器工作于*交互模式*。这种模式下它根据*主提示符*来执行，主提示符通常标识为三个大于号（“>>>”）；继续的部分被称为*从属提示符*，由三个点标识（“…”）。在第一行之前，解释器打印欢迎信息、版本号和授权提示：

```
python
Python 2.3 (#1, Jul 30 2003, 23:22:59)
[GCC 3.2 20020927 (prerelease)] on cygwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

输入多行结构时需要从属提示符了，例如，下面这个 if 语句：

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

2.2 解释器及其 环境

2.2.1 错误 处理

有错误发生时，解释器打印一个错误信息和栈跟踪（监视）器？ 。交互模式下，它返回主提示符，如果从文件输入执行，它在打印栈跟踪器后以非零状态退出。（异常可以由 try 语句中的 except 子句来控制，这样就不会出现上文中的错误信息） 有一些非常致命的错误会导致非零状态下退出，这由通常由内部矛盾和内存溢出造成。所有的错误信息都写入标准错误流；命令中执行的普通输出写入标准输出。

在主提示符或附属提示符输入中断符（通常是 Control-C or DEL）就会取消当前输入，回到主命令行。[2.1]执行命令时输入一个中断符会抛出一个 KeyboardInterrupt 异常，它可以被 try 句截获。

2.2.2 执行 Python 脚本

BSD 系统中，Python 脚本可以像 She11 脚本那样直接执行，只要在脚本文件开头写一行命令，指定文件和模式：

```
#! /usr/bin/env python
```

(将用户路径通知解释器) “#!”必须是文件的前两个字符，在某些平台上，第一行必须以 Unix 风格的行结束符（“\n”）结束，不能用 Mac（“\r”）或 Windows（“\r\n”）的结束符。注意，“#”是 Python 中是行注释的起始符。

脚本可以通过 chm od 命令指定执行模式和许可权。

```
$ chmod +x myscript.py
```

2.2.3 源程序 编码

Python 的源文件可以通过编码使用 ASCII 以外的字符集。 最好的做法是在 #! 行后面用一个特殊的注释行来定义字符集。

```
# -*- coding: iso-8859-1 -*-
```

根据这个声明，Python 会将文件中的字符尽可能的从指定的编码转为 Unicode，在本例中，这个字符集是 iso-8859-1 。在 *Python 库参考手册* 中可以找到可用的编码列表（根据我的实验，中文似乎只能用 cp-936 或 utf-8，不直接支持 GB，GBK，GB-18030 或 ISO-10646－－译者注）。

如果你的文件编辑器支持 UTF-8 格式，并且可以保存 UTF-8 标记（aka BOM - Byte Order Mark），你可以用这个来代替编码声明（看来至少 Jext 还不支持这样做，而 Vim，我还没找到它的编码设置在哪里，还是老老实实的用注释行指定源代码的编码吧－－译者注）。IDLE 可以通过设定 Options/General/Default Source Encoding/UTF-8 来支持它。需要注意的是旧版 Python 不支持这个标记（Python 2.2 或更早的版本），也同样不能使操作系统支持#!文件。

使用 UTF-8 内码（无论是用标记还是编码声明），我们可以在字符串和注释中使用世界上的大部分语言。标识符中不能使用非 ASCII 字符集。为了正确显示所有的字符，你一定要在编辑器中将文件保存为 UTF-8 格式，而且要使用支持文件中所有字符的字体。

2.2.4 交互式环境的启动文件

使用 Python 解释器的时候，我们可能需要在每次解释器启动时执行一些命令。你可以在一个文件中包含你想要执行的命令，设定一个名为 PYTHONSTARTUP 的环境变量来指定这个文件。这类似于 Unix shell 的.profile 文件。

这个文件在交互会话期是只读的，当 Python 从脚本中解读文件或以终端做为外部命令源时则不会如此（尽管它们的行为很像是处在交互会话期。）它与解释器执行的命令处在同一个命名空间，所以由它定义或引用的一切可以在解释器中不受限制的使用。你也可以在这个文件中改变 sys.ps1 和 sys.ps2 指令。

如果你想要在当前目录中执行附加的启动文件，你可以在全局启动文件中加入类似以下的代码：“if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')”。如果你想要在某个脚本中使用启动文件，必须要在脚本中写入这样的语句：

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

3. Python 的非正式介绍

在后面的例子中，区分输入和输出的方法是看是否有提示符（“>>>”和“…”）：想要重复这些例子的话，你就要在提示符显示后输入所有的一切；没有以提示符开始的行，是解释器输出的信息。需要注意的是示例中的从属提示符用于多行命令的结束，它表示你需要输入一个空行。

本手册中的很多示例都包括注释，甚至有一些在交互提示符中折行。Python 中的注释以符号“#”起始，一直到当前行的结尾。注释可能出现在一行的开始，也可能跟在空格或程序代码之后，但不会出现在字符串中，字符串中的#号只代表#号。

示例：

```
# this is the first comment
SPAM = 1                 # and this is the second comment
                         # … and now a third!
STRING = "# This is not a comment."
```

3.1 初步认识 Python

让我们试验一些简单的 Python 命令。启动解释器然后等待主提示符“>>>”出现（这用不了太久）。

3.1.1 数值

解释器的行为就像是一个计算器。你可以向它输入一个表达式，它会返回结果。表达式的语法简明易懂：+，-，*，/和大多数语言中的用法一样（比如 C 或 Pascal），括号用于分组。例如：

```
>>> 2+2
4
>>> # This is a comment
… 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
```

```
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

像 c 一样，等号（“=”）用于给变量赋值。被分配的值是只读的。

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

同一个值可以同时赋给几个变量：

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

Python 完全支持浮点数，不同类型的操作数混在一起时，操作符会把整型转化为浮点数。

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

复数也同样得到了支持，虚部由一个后缀“j”或者“J”来表示。带有非零实部的复数记为“(*real*+*imag*j)”，或者也可以通过“complex(*real*, *imag*)”函数创建。

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

复数总是由实部和虚部两部分浮点数来表示。可能从 *z*.real 和 *z*.imag 得到复数 z 的实部和虚部。

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

用于向浮点数和整型转化的函数（float(), int() 和 long()）不能对复数起作用——没有什么方法可以将复数转化为实数。可以使用 abs(z)取得它的模，也可以通过 z.real 得到它的实部。

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

交互模式下，最近一次表达式输出保存在_变量中。这意味着把 Python 当做桌面计算器使用时，它可以更容易的进行连续计算，例如：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
>>>
```

这个变量对于用户来说是只读的。不要试图去给它赋值——由于 Python 的语法效果，你只会创建一个同名的局部变量覆盖它。

3.1.2 字符串

除了数值，Python 还可以通过几种不同的方法操作字符串。字符串用单引号或双引号标识：

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

字符串可以通过几种方式分行。可以在行加反斜杠做为继续符，这表示下一行是当前行的逻辑沿续。

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
 significant."

print hello
```

注意换行用 \n 来表示；反斜杠后面的新行标识（newline，缩写“n”）会转换为换行符，示例会按如下格式打印：

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

然而，如果我们创建一个“raw”行，\n 序列就不会转为换行，示例源码最后的反斜杠和换行符 n 都会做为字符串中的数据处理。如下所示：

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."

print hello
```

会打印为：

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

或者，字符串可以用一对三重引号”””或'''来标识。三重引号中的字符串在行尾不需要换行标记，所有的格式都会包括在字符串中。

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

produces the following output:

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

解释器打印出来的字符串与它们输入的形式完全相同：内部的引号，用反斜杠标识的引号和各种怪字符，都精确的显示出来 。如果字符串中包含单引号，不包含双引号，可以用双引号引用它，反之可以用单引号。（后面介绍的 print 语句，可以可以用来写没有引号和反斜杠的字符串）。

字符串可以用+号联接（或者说粘合），也可以用*号循环。

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

两个字符串值之间的联接是自动的，上例第一行可以写成“word = 'Help' 'A'”这种方式只对字符串值有效，任何字符串表达式都不适用这种方法。

```
>>> import string
>>> 'str' 'ing'                   #  <-  This is ok
'string'
>>> string.strip('str') + 'ing'   #  <-  This is ok
'string'
>>> string.strip('str') 'ing'     #  <-  This is invalid
  File "<stdin>", line 1, in ?
    string.strip('str') 'ing'
                        ^
SyntaxError: invalid syntax
```

字符串可以用下标（索引）查询；就像 C 一样，字符串的第一个字符下标是 0。这里没有独立的字符类型，字符仅仅是大小为一的字符串。就像在 Icon 中那样，字符串的子串可

以通过切片标志来表示：两个由冒号隔开的索引。

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

切片索引可以使用默认值；省略前一个索引表示 0，省略后一个索引表示被切片的字符串的长度。

```
>>> word[:2]    # The first two characters
'He'
>>> word[2:]    # All but the first two characters
'lpA'
```

和 C 字符串不同，Python 字符串不能改写。按字符串索引赋值会产生错误。

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

然而，可以通过简单有效的组合方式生成新的字符串：

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

切片操作有一个很有用的不变性： s[:i] + s[i:] 等于 s。

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

退化的切片索引被处理的很优美：过大的索引代替为字符串大小，下界比上界大的返回空字符串。

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

索引可以是负数，计数从右边开始，例如：

```
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # All but the last two characters
'Hel'
```

不过-0 还是 0，所以它不是从右边计数的！

```
>>> word[-0]     # (since -0 equals 0)
'H'
```

越界的负切片索引会被截断，不过不要尝试在前元素索引（非切片的）中这样做：

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

理解切片的最好方式是把索引视为*两个字符之间*的点，第一个字符的左边是 0，字符串中第 n 个字符的右边是索引 n，例如：

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

第一行是字符串中给定的 0 到 5 各个索引的位置，第二行是对应的负索引。从 i 到 j 的切片由这两个标志之间的字符组成。

对于非负索引，切片长度就是两索引的差。例如，word[1:3]的长度是 2。

内置函数 len() 返回字符串长度：

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

3.1.3 Unicode 字符串

从 Python2.0 开始，程序员们可以使用一种新的数据类型来存储文本数据：Unicode 对象。它可以用于存储多种 Unicode 数据（请参阅 http://www.unicode.org/ ），并且，通过必要时的自动转换，它可以与现有的字符串对象良好的结合。

Unicode 针对现代和旧式的文本中所有的字符提供了一个序列。 以前，字符只能使用 256 个序号，文本通常通过绑定代码页来与字符映射。这很容易导致混乱，特别是软件的国际化（internationalization－－通常写做“i18n”－－“i”+18 characters +“n”）。Unicode 通过为所有字符定义一个统一的代码页解决了这个问题。

Python 中定义一个 Unicode 字符串和定义一个普通字符串一样简单：

```
>>> u'Hello World !'
u'Hello World !'
```

引号前小写的“u”表示这里创建的是一个 Unicode 字符串。如果你想加入一个特殊字符，可以使用 Python 的 *Unicode-Escape* 编码。如下例所示：

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

被替换的 \u0020 标识表示在给定位置插入编码值为 0x0020 的 Unicode 字符（空格符）。

其它字符也会被直接解释成对应的 Unicode 码。如果你有一个在西方国家常用的 Latin-1 编码字符串，你可以发现 Unicode 字符集的前 256 个字符与 Lation-1 的对应字符编码完全相同。

另外，有一种与普通字符串相同的行模式。想要使用 Python 的 *Raw-Unicode-Escape* 编码，你需要在字符串的引号前加上 ur 前缀。如果在小写“u”前可能有不止一个反斜杠，它只会把那些单独的 \uXXXX 转化为 Unicode 字符。

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\u0020World !'
```

行模式在你需要输入很多个反斜杠时很有用，可能会用于正规表达式。

作为这些编码标准的一部分，Python 提供了一个完备的方法集用于从已知的编码集创建 Unicode 字符串。

内置函数 unicode()提供了访问（编码和解码）所有已注册的 Unicode 编码的方法。它能转换众所周知的 *Latin-1*, *ASCII*, *UTF-8*, 和 *UTF-16*。后面的两个可变长编码字符集用一个或多个 byte 存储 Unicode 字符。 默认的字符集是 ASCII ，它只处理 0 到 127 的编码，拒绝其它的字符并返回一个错误。当一个 Unicode 字符串被打印、写入文件或通过 str()转化时，它们被替换为默认的编码。

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal not in range(128)
```

要把一个 Unicode 字符串用指定的字符集转化成 8 位字符串，可以使用 Unicode 对象提供的 encode()方法，它有一个参数用以指定编码名称。编码名称小写。

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

如果你有一个特定编码的字符串，想要把它转为 Unicode 字符集，，可以使用 uncode()函数，它以编码名做为第二个参数。

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 链表

Python 已经有了几个复合数据类型，用于组织其它的值。最通用的是*链表*，它写为中括之间用逗号分隔的一列值（子项），链表的子项不一定是同一类型的值。

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

像字符串一样，链表也以零开始，可以被切片，联接，等等：

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
```

与不变的字符串不同，链表可以改变每个独立元素的值：

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

可以进行切片操作，甚至还可以改变链表的大小：

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a     # Insert (a copy of) itself at the beginning
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
```

内置函数 len()也同样可以用于链表：

```
>>> len(a)
8
```

它也可以嵌套链表（在链表中创建其它链表），例如：

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')      # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

注意最后一个例子，p[1]和 q 实际上指向同一个对象！我们在后面会讲到对象语法。

3.2 开始编程

当然，我们可以用 Python 做比 2 加 2 更复杂的事。例如，我们可以用以下的方法输出*菲波那契*（*Fibonacci*）序列的子序列：

```
>>> # 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
```

示例中介绍了一些新功能：

- 第一行包括了*复合参数*：变量 a 和 b 同时被赋值为 0 和 1。最后一行又一次使用了这种技术，证明了在赋值之前表达式右边先进行了运算。右边的表达式从左到右运算。
- while 循环运行在条件为真时执行。在 Python 中，类似于 C 任何非零值为真，零为假。 这个条件也可以用于字符串或链表，事实上于对任何序列类型，长度非零时

为真，空序列为假。示例所用的是一个简单的比较。标准的比较运算符写法和 C 相同： < （小于），> （大于），== （等于），<= （小于等于），>=（大于等于）和!= （不等于）。

- 循环体是缩进的：缩进是 Python 对语句分组的方法。 Python 仍没有提供一个智能编辑功能所以你要在每一个缩进行输入一个 tab 或（一个或多个）空格。 实际上你可能会准备更为复杂的文本编辑器来编写你的 Python 程序，大多数文本编辑器都提供了自动缩进功能。交互式的输入一个复杂语句时，需要用一个空行表示完成（因为解释器没办法猜出你什么时候输入最后一行）。需要注意的是每一行都要有相同的空字符来标识这是同一个语句块。
- print 语句打印给定表达式的值。它与你仅仅输入你需要的表达式（就像前面的计算器示例）不同，它可以同时输出多个表达式。字符串输出时没有引号，各项之间用一个空格分开，你可以很容易区分它们，如下所示：

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

print 语句末尾的逗号避免了输出中的换行：

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

需要注意的是，如果最后一行仍没有写完，解释器会在它打印下一个命令时插入一个新行。

4. 其它流程控制工具

除了前面介绍的 while 语句，Python 还从别的语言中借鉴了一些流程控制功能，并有所改变。

4.1 if 语句

也许最有句的语句类型是 if 语句。例如：

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

可能会有 0 或很多个 elif 部分，else 是可选的。关键字“elif ” 是“ else if ”的缩写，这个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的 switch 或 case 语句。

4.2 for 语句

Python 中的 for 语句和你在 C 或 Pascal 中使用的略有不同。 通常的循环可能会依据一个等差数值步进过程（如 Pascal）或由用户来定义迭代步骤和中止条件（如 C），Python 的 for 语句依据任意序列（链表或字符串）中的子项，按它们在序列中的顺序来进行迭代。例如（没有暗指）：

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
```

```
cat 3
window 6
defenestrate 12
```

在迭代过程中修改迭代序列不安全（只有在使用链表这样的可变序列时才会有这样的情况）。如果你想要修改你迭代的序列（例如，复制选择项），你可以迭代它的复本。通常使用切片标识就可以很方便的做到这一点：

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

4.3 range() 函数

如果你需要一个数值序列，内置函数 range()可能会很有用，它生成一个等差级数链表。

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range(10)生成了一个包含 10 个值的链表，它准确的用链表的索引值填充了这个长度为 10 的列表，所生成的链表中不包括范围中的结束值。也可以让 range 操作从另一个数值开始，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, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

需要迭代链表索引的话，如下所示结合使用 range()和 len()：

```
>>> 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
```

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

break 语句和 C 中的类似，用于跳出最近的一级 for 或 while 循环。

continue 语句是从 C 中借鉴来的，它表示循环继续执行下一次迭代。

循环可以有一个 else 子句;它在循环迭代完整个列表（对于 for）或执行条件为 false（对于 while）时执行，但循环被 break 中止的情况下不会执行。以下搜索素数的示例程序演示了这个子句：

```
>>> 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
```

4.5 pass 语句

pass 语句什么也不做。它用于那些语法上必须要有什么语句，但程序上什么也不要做的场合，例如：

```
>>> while True:
...       pass # Busy-wait for keyboard interrupt
...
```

4.6 定义函数

我们可以编写一个函数来生成有给定上界的菲波那契数列:

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

关键字 def 引入了一个函数*定义*。在其后必须跟有函数名和包括形式参数的圆括号。函数体语句从下一行开始，必须是缩进的。函数体的第一行可以是一个字符串值，这个字符串是该函数的文档字符串，也可称作 *docstring*。

有些文档字符串工具可以在线处理或打印文档，或让用户交互的浏览代码;在你的代码中加入文档字符串是一个好的作法，应该养成习惯。

调用函数时会为局部变量引入一个新的符号表。所有的局部变量都存储在这个局部符号表中。引用参数时，会先从局部符号表中查找，然后是全局符号表，然后是内置命名表。因此，全局参数虽然可以被引用，但它们不能在函数中直接赋值（除非它们用 global 语句命名）。

函数引用的实际参数在函数调用时引入局部符号表，因此，实参总是传值调用（这里的*值*总是一个对象*引用*，而不是该对象的值）。[4.1] 一个函数被另一个函数调用时，一个新的局部符号表在调用过程中被创建。

函数定义在当前符号表中引入函数名。作为用户定义函数，函数名有一个为解释器认可的类型值。这个值可以赋给其它命名，使其能句做为一个函数来使用。这就像一个重命名机制:

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

你可能认为 fib 不是一个函数（function），而是一个过程（procedure）。Python 和 C 一样，过程只是一个没有返回值的函数。实际上，从技术上讲，过程也有一个返回值，虽然是一个不讨人喜欢的。这个值被称为 None （这是一个内置命名）。如果一个值只是 None

的话，通常解释器不会写一个 None 出来，如果你真想要看它的话，可以这样做：

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

以下示列演示了如何从函数中返回一个包含菲波那契数列的数值链表，而不是打印它：

```
>>> 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 b < n:
...         result.append(b)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

和以前一样，这个例子演示了一些新的 Python 功能：

- return 语句从函数中返回一个值，不带表达式的 return 返回 None。过程结束后也会返回 None。
- 语句 result.append(b)称为链表对象 result 的一个*方法*（method）。方法是一个“属于”某个对象的函数，它被命名为 obj.methodename，这里的 obj 是某个对象（可能是一个表达式），methodename 是某个在该对象类型定义中的方法的命名。 不同的类型定义不同的方法。不同类型可能有同样名字的方法，但不会混淆。（当你定义自己的对象类型和方法时，可能会出现这种情况，本指南后面的章节会介绍如何使用*类*）。示例中演示的 append()方法由链表对象定义，它向链表中加入一个新元素。在示例中它等同于"result = result + [b]"，不过效率更高。

4.7 深入函 数定义

有时需要定义参数个数可变的函数。有三个方法可以做到，我们可以组合使用它们。

4.7.1 参数 默认值

最有用的形式是给一个或多个参数指定默认值。这样创建的函数可以在调用时使用更少的参数。

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return 1
        if ok in ('n', 'no', 'nop', 'nope'): return 0
        retries = retries - 1
        if retries < 0: raise IOError, 'refusenik user'
        print complaint
```

这个函数还可以用以下的方式调用：ask_ok('Do you really want to quit?')，或者像这样：ask_ok('OK to overwrite the file?', 2)。

默认值在函数*定义*段被解析，如下所示：

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

will print 5.

重要警告： 默认值只会解析一次。当默认值是一个可变对象，诸如链表、字典或大部分类实例时，会产生一些差异。例如，以下函数在后继的调用中会积累它的参数值：

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

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

这会打印出：

```
[1]
[1, 2]
[1, 2, 3]
```

如果你不想在不同的函数调用之间共享参数默认值，可以如下面的实例一样编写函数：

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

4.7.2 参数关键字

函数可以通过参数关键字的形式来调用，形如“*keyword = value*”。*例如，以下的函数：*

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

可以用以下的任一方法调用：

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

不过以下几种调用是无效的：

```
parrot()                     # required argument missing (缺少必要参数)
parrot(voltage=5.0, 'dead')  # non-keyword argument following keyword (在关键字后面有非关键字参数)
parrot(110, voltage=220)     # duplicate value for argument (对参数进行了重复赋值)
parrot(actor='John Cleese')  # unknown keyword (未知关键字)
```

通常，参数列表中的每一个关键字都必须来自于形式参数，每个参数都有对应的关键字。形式参数有没有默认值并不重要。实际参数不能一次赋多个值－－形式参数不能在同一次

调用中同时使用位置和关键字绑定值。这里有一个例子演示了在这种约束下所出现的失败情况:

```
>>> 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'
```

引入一个形如 **name 的参数时，它接收一个字典，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。这里可能还会组合使用一个形如 *name 的形式参数，它接收一个拓扑（下一节中会详细介绍），包含了所有没有出现在形式参数列表中的参数值。（*name 必须在 **name 之前出现） 例如，我们这样定义一个函数:

```
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 = keywords.keys()
    keys.sort()
    for kw in keys: print kw, ':', keywords[kw]
```

它可以像这样调用:

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

当然它会按如下内容打印:

```
-- 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 sort()方法在关键字字典内容打印前被调用，否则的话，打印参数时的顺序是未定义的。

4.7.3 可变参数表

最后，一个最不常用的选择是可以让函数调用可变个数的参数。这些参数被包装进一个拓扑。在这些可变个数的参数之前，可以有零到多个普通的参数：

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

4.7.4 Lambda 形式

出于适当的需要，有几种通常在功能性语言和 Lisp 中出现的功能加入到了 Python。通过 lambda 关键字，可以创建很小的匿名函数。这里有一个函数返回它的两个参数的和："lambda a, b: a+b"。 Lambda 形式可以用于任何需要的函数对象。出于语法限制，它们只能有一个单独的表达式。语义上讲，它们只是普通函数定义中的一个语法技巧。类似于嵌套函数定义，lambda 形式可以从包含范围内引用变量：

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

4.7.5 文档字符串

这里介绍文档字符串的概念和格式。

第一行应该是关于对象用途的简介。简短起见，不用明确的陈述对象名或类型，因为它们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，以句号结尾。

如果文档字符串有多行，第二行应该空出来，与接下来的详细描述明确分隔。接下来的文档应该有一或多段描述对象的调用约定、边界效应等。

Python 的解释器不会从多行的文档字符串中去除缩进，所以必要的时候应当自己清除缩进。这符合通常的习惯。第一行*之后*的第一个非空行决定了整个文档的缩进格式。（我们不用第一行是因为它通常紧靠着起始的引号，缩进格式显示的不清楚。）留白"相当于"是字

符串的起始缩进。 每一行都不应该有缩进，如果有缩进的话，所有的留白都应该清除掉。相当于留白就是验证后的制表符扩展（通常是 8 个空格）。（这一段译得不通，有疑问的读者请参见原文－－译者）

以下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.
```

脚注

... 对象）。[4.1]

实际上，说是*调用对象引用*更合适，因为如果传入一个可变对象，调用者可以得到被调用者产生的任何改变（如链表中插入了子项）。

5. 数据结 构

本章节深入讲述一些你已经学习过的东西，并且还加入了新的内容。

5.1 深入链表

链表类型有很多方法，这里是链表类型的所有方法：

append（*x*）

把一个元素添加到链表的结尾，相当于 a[len(a):] = [*x*]。

extend（*L*）

通过添加指定链表的所有元素来扩充链表，相当于 a[len(a):] = *L* 。

insert（*i*, *x*）

在指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，例如 a.insert(0, *x*) 会插入到整个链表之前，而 a.insert(len(a), *x*) 相当于 a.append(*x*)。

remove（*x*）

删除链表中值为 x 的第一个元素。如果没有这样的元素，就会返回一个错误。

pop([*i*])

从链表的指定位置删除元素，并将其返回。如果没有指定索引，a.pop() 返回最后一个元素。元素随即从链表中被删除。（方法中 *i* 两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号，你会经常在 *Python 库参考手册*中遇到这样的标记。）

ind ex(*x*)

返回链表中第一个值为 *x* 的元素的索引。如果没有匹配的元素就会返回一个错误。

count(*x*)

返回 *x* 在链表中出现的次数。

sort()

对链表中的元素进行适当的排序。

reverse ()

倒排链表中的元素。

下面这个示例演示了链表的大部分方法:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 把链表 当作堆 栈使用

链表方法使得链表可以很方便的做为一个堆栈来使用，堆栈是这样的数据结构，最先进入的元素最后一个被释放（后进先出）。用 append() 方法可以把一个元素添加到堆栈顶。用不指定索引的 pop() 方法可以把一个元素从堆栈顶释放出来。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 把链表 当作队 列使用

你也可以把链表当做队列使用，队列是这样的数据结构，最先进入的元素最先释放（先进先出）。使用 append()方法可以把元素添加到队列最后，以 0 为参数调用 pop() 方法可以把最先进入的元素释放出来。例如：

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

5.1.3 函数 化编程 工具

对于链表来讲，有三个内置函数非常有用：filter()， map()， 和 reduce()。

“filter(*function*, *sequence*)” 返回一个序列（sequence），包括了给定序列中所有调用 function(item)后返回值为 true 的元素。（如果可能的话，会返回相同的类型）。例如，以下程序可以计算部分素数：

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
…
>>> filter(f, range(2, 25))
```

```
[5, 7, 11, 13, 17, 19, 23]
```

“map(*function*, *sequence*)” 为每一个元素依次调用 *function*(*item*) 并将返回值组成一个链表返回。例如，以下程序计算立方：

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

可以传入多个序列，函数也必须要有对应数量的参数，执行时会依次用各序列上对应的元素来调用函数（如果某些序列比其它的短，就用 None 来代替）。如果把 None 做为一个函数传入，则直接返回参数做为替代。

组合这两种情况，我们会发现“map(None, *list1*, *list2*)”是把一对序列变成元素对序列的便捷方式。例如：

```
>>> seq = range(8)
>>> def square(x): return x*x
...
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

"reduce(*func*, *sequence*)" 返回一个单值，它是这样构造的：首先以序列的前两个元素调用函数，再以返回值和第三个参数调用，依次执行下去。例如，以下程序计算 1 到 10 的整数之和：

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

如果序列中只有一个元素，就返回它，如果序列是空的，就抛出一个异常。

可以传入第三个参数做为初始值。如果序列是空的，就返回初始值，否则函数会先接收初始值和序列的第一个元素，然后是返回值和下一个元素，依此类推。例如：

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

不要像示例中这样定义 sum()：因为合计数值是一个通用的需求，在新的 2.3 版中，提供了

内置的 sum(*sequence*) 函数。

5.1.4 链表推导式

链表推导式提供了一个创建链表的简单途径，无需使用 map()， filter() 以及 lambda。返回链表的定义通常要比创建这些链表更清晰。每一个链表推导式包括在一个 for 语句之后的表达式，零或多个 for 或 if 语句。返回值是由 for 或 if 子句之后的表达式得到的元素组成的链表。如果想要得到一个元组，必须要加上括号。

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]      # error - parens required for tuples
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
             ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

为使链表推导式匹配 for 循环的行为，可以在推导之外保留循环变量：

```
>>> x = 100                     # this gets overwritten
>>> [x**3 for x in range(5)]
[0, 1, 8, 27, 64]
>>> x                           # the final value for range(5)
4
```

5.2 del 语句

有一个方法可从链表中删除指定索引的元素：del 语句。这个方法也可以从链表中删除切片（之前我们是把一个空链表赋给切片）。例如：

```
>>> a = [-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

del 也可以用于删除整个变量：

```
>>> del a
```

此后再引用这个名字会发生错误（至少要到给它赋另一个值为止）。后面我们还会发现 del 的其它用法。

5.3 元组（Tuples） 和序列（Sequences ）

我们知道链表和字符串有很多通用的属性，例如索引和切片操作。它们是*序列*类型中的两种。因为 Python 是一个在不停进化的语言，也可以加入其它的序列类型，这里有另一种标准序列类型：元组。

一个元组由数个逗号分隔的值组成，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))
```

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。在输入时可能有或没有括号都可以，不过经常括号都是必须的（如果元组是一个更大的表达式的一部分）。

元组有很多用途。例如（x, y）坐标点，数据库中的员工记录等等。元组就像字符串，不

可改变：不能给元组的一个独立的元素赋值（尽管你可以通过联接和切片来模仿）。也可以通过包含可变对象来创建元组，例如链表。

一个特殊的问题是构造包含零个或一个元素的元组：为了适应这种情况，语法上有一些额外的改变。一对空的括号可以创建空元组；要创建一个单元素元组可以在值后面跟一个逗号（在括号中放入一个单值是不够的）。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*元组封装（sequence packing）*的一个例子：值 12345， 54321 和 'hello!' 被封装进元组。其逆操作可能是这样：

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

这个调用被称为*序列拆封*非常合适。序列拆封要求左侧的变量数目与序列的元素个数相同。要注意的是可变参数（multiple assignment

）其实只是元组封装和序列拆封的一个结合！

这里有一点不对称：封装多重参数通常会创建一个元组，而拆封操作可以作用于任何序列。

5.4 字典（Dictionaries）

另一个非常有用的 Python 内建数据类型是*字典（Dictionaries）*。字典在某些语言中可能称为“联合内存”（``associative memories''）或“联合数组”（``associative arrays''）。序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。如果元组中只包含字符串和数字，它可以做为关键字，如果它直接或间接的包含了可变对象，就不能当做关键字。不能用链表做关键字，因为链表可以用它们的 append() 和 extend() 方法，或者用切片、或者通过检索变量来即时改变。

理解字典的最佳方式是把它看做无序的*关键字：值*对（ *key:value* pairs ）集合，关键字必须是互不相同的（在同一个字典之内）。一对大括号创建一个空的字典：{}。初始化链表时，在大括号内放置一组逗号分隔的关键字：值对，这也是字典输出的方式。

字典的主要操作是依据关键字来存储和析取值。也可以用 del 来删除关键字：值对。如果你用一个已经存在的关键字存储值，以前为该关键字分配的值就会被遗忘。试图析取从一个不存在的关键字中读取值会导致错误。

字典的 keys() 方法返回由所有关键字组成的链表，该链表的顺序不定（如果你需要它有序，只能调用关键字链表的 sort()方法）。使用字典的 has_key() 方法可以检查字典中是否存在某一关键字。

这是一个关于字典应用的小示例：

```
>>> 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}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
```

链表中存储关键字-值对元组的话，字典可以从中直接构造。关键字-值对来自一个模式时，可以用链表推导式简单的表达关键字-值链表。

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in vec])     # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

5.5 循环技巧

在字典中循环时，关键字和对应的值可以使用 items() 方法同时解读出来。

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

在序列中循环时，索引位置和对应值可以使用 enumerate() 函数同时得到。

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

同时循环两个或更多的序列，可以使用 zip() 整体解读。

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

5.6 深入条件控制

用于 while 和 if 语句的条件包括了比较之外的操作符。

in 和 not 比较操作符审核值是否在一个区间之内。操作符 is 和 is not 比较两个对象是否相同； 这只和诸如链表这样的可变对象有关。所有的比较操作符具有相同的优先级，低于所有的数值操作。

比较操作可以传递。例如 a < b == c 审核是否 a 小于 b 并 b 等于 c。

比较操作可以通过逻辑操作符 and 和 or 组合，比较的结果可以用 not 来取反义。这些操作符的优先级又低于比较操作符，在它们之中，not 具有最高的优先级，or 的优先组最低，所以 A and not B or C 等于 (A and (not B)) or C。当然，表达式可以用期望的方式表示。

逻辑操作符 and 和 or 也称作*短路操作符*：它们的参数从左向右解析，一旦结果可以确定就停止。例如，如果 A 和 C 为真而 B 为假，A and B and C 不会解析 C。作用于一个普通的非逻辑值时，短路操作符的返回值通常是最后一个变量。

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_nu11
'Trondheim'
```

需要注意的是 Python 与 C 不同，内部表达式不能分配值。C 程序员经常对此抱怨，不过它避免了一类在 C 程序中司空见惯的错误：想要在解析式中使==时误用了＝操作符。

5.7 比较序列和其它类型

序列对象可以与相同类型的其它对象比较。 比较操作按*字典*序进行：首先比较前两个元素，如果不同，就决定了比较的结果；如果相同，就 比较后两个元素，依此类推，直到所有序列都完成比较。如果两个元素本身就是同样类型的序列，就递归字典序比较。如果两个序列的所有子项都相等，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，较短的一个序列就小于另一个。字符串的字典序按照单字符的 ASCII 顺序。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' < 'Pasca1' < '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)
```

需要注意的是不同类型的对象比较是合法的。输出结果是确定而非任意的：类型按它们的名字排序。因而，一个链表（1ist）总是小于一个字符串（string），一个字符串（string）总是小于一个元组（tup1e）等等。数值类型比较时会统一它们的数据类型，所以 0 等于 0.0，等等。[5.1]

注释

... etc.[5.1]
不同类型对象的比较规则不依赖于此，它们有可能会在 Python 语言的后继版本中改变。

6. 模块

如果你退出 Python 解释器重新进入，以前创建的一切定义（变量和函数）就全部丢失了。因此，如果你想写一些长久保存的程序，最好使用一个文本编辑器来编写程序，把保存好的文件输入解释器。 我们称之为创建一个*脚本*。程序变得更长一些了，你可能为了方便维护而把它分离成几个文件。你也可能想要在几个程序中都使用一个常用的函数，但是不想把它的定义复制到每一个程序里。

为了支持这些需要，Python 提供了一个方法可以从文件中获取定义，在脚本或者解释器的一个交互式实例中使用。这样的文件被称为*实例*；模块中的定义可以导入到另一个模块或*主*模块中（在脚本执行时可以调用的变量集位于最高级，并且处于计算器模式）

模块是包括 Python 定义和声明的文件。文件名就是模块名加上.py 后缀。模块的模块名（做为一个字符串）可以由全局变量__name__得到。例如，你可以用自己惯用的文件编辑器在当前目录下创建一个叫 fibo.py 的文件，录入如下内容：

```
# Fibonacci numbers module

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

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 解释器用如下命令导入这个模块：

```
>>> import fibo
```

这样做不会直接把 fibo 中的函数导入当前的语义表；，它只是引入了模块名 fibo。你可以通过模块名按如下方式访问这个函数：

```
>>> 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'
```

如果你想要直接调用函数，通常可以给它赋一个本地名称:

```
>>> 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

模块可以像函数定义一样包含执行语句。这些语句通常用于初始化模块。它们只在模块第一次导入时执行一次。

对应于定义模块中所有函数的全局语义表，每一个模块有自己的私有语义表。因此，模块作者可以在模块中使用一些全局变量，不会因为与用户的全局变量冲突而引发错误。另一方面，如果你确定你需要这个，可以像引用模块中的函数一样获取模块中的全局变量，形如: modname.itemname。

模块可以导入（import）其它模块。习惯上所有的 import 语句都放在模块（或脚本，等等）的开头，但这并不是必须的。被导入的模块名入在本模块的全局语义表中。

import 语句的一个变体直接从被导入的模块中导入命名到本模块的语义表中。例如:

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

这样不会从局域语义表中导入模块名（例如，fibo 没有定义）。

这里还有一个变体从模块定义中导入所有命名:

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

这样可以导入所有除了以下划线（_）开头的命名。

6.1.1 模块搜索路径

导入一个叫 spam 的模块时，解释器先在当前目录中搜索名为 spam.py 的文件，然后在环境变量 PYTHONPATH 指琮的目录列表中搜索，然后是环境变量 PATH 中的路径列表。如果 PYTHONPATH 没有设置，或者文件没有找到，接下来搜索安装目录，在 UNIX 中，通常是 .:

/usr/local/lib/python。

实际上，解释器由 sys.path 变量指定的路径目录搜索模块，该变量初始化时默认包含了输入脚本（或者当前目录），PATHPATH 和安装目录。这样就允许 Python 程序（原文如此，programs；我猜想应该是“programer”，程序员——译者）了解如何修改或替换模块搜索目录。需要注意的是由于这些目录中包含有搜索路径中运行的脚本，所以这些脚本不应该和标准模块重名，否则在导入模块时 Python 会尝试把这些脚本当作模块来加载。这通常会引发一个错误。请参见 6.2 节“标准模块”以了解更多的信息。

6.1.2 “编译”Python 文件

对于引用了大量标准模块的短程序，有一个提高启动速度有重要方法，如果在 spam.py 目录下存在一个名为 spam.pyc 的文件，它会被视为 spam 模块的预“编译”（``byte-compiled"，二进制编译）版本。用于创建 spam.pyc 的这一版 spam.py 的修改时间记录在 spam.pyc 文件中，如果两者不匹配，.pyc 文件就被忽略。

通常你不需要为创建 spam.pyc 文件做任何工作。一旦 spam.py 成功编译，就会试图编译对应版本的 spam.pyc。如果有任何原因导致写入不成功，返回的 spam.pyc 文件就会视为无效，随后即被忽略。spam.pyc 文件的内容是平台独立的，所以 Python 模块目录可以在不同架构的机器之间共享。

部分高级技巧：

- 以-O 参数调用 Python 解释器时，会生成优化代码并保存在.pyo 文件中。通用的优化器没有太多帮助；它只是删除了断言（assert）语句。使用-O 参数，所有的代码都会被优化；.pyc 文件被忽略，.py 文件被编译为优化代码。
- 向 Python 解释器传递两个 -O 参数（-OO）会执行完全优化的二进制优化编译，这偶尔会生成错误的程序。当前，压缩的.pyo 文件只是从二进制代码中删除了__doc__字符串。因为某些程序依赖于这些变量的可用性，你应该只在确定无误的场合使用这一选项。
- 来自.pyc 文件或.pyo 文件中的程序不会比来自.py 文件的运行更快；.pyc 或.pyo 文件只是在它们加载的时候更快一些。
- 通过脚本名在命令行运行脚本时，不会为该脚本向创建.pyc 或.pyo 文件的二进制代码。当然，把脚本的主要代码移进一个模块里，然后用一个小的解构脚本导入这个模块，就可以提高脚本的启动速度。也可以直接在命令行中指定一个.pyc 或乾.pyo 文件。
- 对于同一个模块（这里指例程 spam——译者），可以只有 spam.pyc 文件（或者 spam.pyo，在使用 -O 参数时）而没有 spam.py 文件。这样可以打包发布比较于逆向工程的 Python 代码库。

- compileall 模块可以为指定目录中的所有模块创建.pyc 文件（或者使用-O 参数创建.pyo 文件）。

6.2 标准模块

Python 带有一个标准模块库，并发布有独立的文档，名为 *Python 库参考手册*（此后称其为“库参考手册”）。有一些模块内置于解释器之中，这些操作的访问接口不是语言内核的一部分，但是已经内置于解释器了。这既是为了提高效率，也是为了给系统调用等操作系统原生访问提供接口。这类模块集合是一个依赖于底层平台的配置选项。例如，amoeba 模块只提供对 Amoeba 原生系统的支持。有一个具体的模块值得注意：sys，这个模块内置于所有的 Python 解释器。变量 sys.ps1 和 sys.ps2 定义了主提示符和副助提示符字符串：

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

这两个变量只在解释器的交互模式下有意义（此处原文为：These two variables are only defined if the interpreter is in interactive mode.）。

变量 sys.path 是解释器模块搜索路径的字符串列表。它由环境变量 PYTHONPATH 初始化，如果 PYTHONPATH 没有内定，就由内置的默认值初始化。你可以用标准和字符串操作修改它：

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

6.3 dir() 函数

内置函数 dir()用于按模块名搜索模块定义，它返回一个字符串类型的存储列表：

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

无参数调用时，dir()函数返回你当前定义的名称：

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

应该该列表列出了所有类型的名称：变量，模块，函数，等等：

dir()不会列出内置函数和变量名。如果你想列出这此内容，它们在标准模块__buildin__中定义：

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'False', 'FloatingPointError', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError', 'OverflowWarning',
 'PendingDeprecationWarning', 'ReferenceError',
 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
 'True', 'TypeError', 'UnboundLocalError', 'UnicodeError', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '__debug__', '__doc__',
 '__import__', '__name__', 'abs', 'apply', 'bool', 'buffer',
```

```
'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'min',
'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit',
'range', 'raw_input', 'reduce', 'reload', 'repr', 'round',
'setattr', 'slice', 'staticmethod', 'str', 'string', 'sum', 'super',
'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 包

包通常是使用用“圆点模块名”的结构化模块命名空间。例如，名为 A.B 的模块表示了名为“A”的包中名为“B”的子模块。正如同用模块来保存不同的模块架构可以避免全局变量之间的相互冲突，使用圆点模块名保存像 NunPy 或 Python Imaging Library 之类的不同类库架构可以避免模块之间的命名冲突。

假设你现在想要设计一个模块集（一个“包”）来统一处理声音文件和声音数据。存在几种不同的声音格式（通常由它们的扩展名来标识，例如：.wav，.aiff，.au），于是，为了在不同类型的文件格式之间转换，你需要维护一个不断增长的包集合。 可能你还想要对声音数据做很多不同的操作（例如混音，添加回声，应用平衡功能，创建一个人造效果），所以你要加入一个无限流模块来执行这些操作。你的包可能会是这个样子（通过分级的文件体系来进行分组）：

```
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
        …
```

导入模块时，Python 通过 sys.path 中的目录列表来搜索存放包的子目录。

必须要有一个__init__.py 文件的存在，才能使 Python 视该目录为一个包；这是为了防止某些目录使用了“string”这样的通用名而无意中在随后的模块搜索路径中覆盖了正确的模块。最简单的情况下，__init__.py 可以只是一个空文件，不过它也可能包含了包的初始化代码，或者设置了 __all__ 变量，后面会有相关介绍。

包用户可以从包中导入合法的模块，例如：

```
import Sound.Effects.echo
```

这样就导入了 Sound.Effects.echo 子模块。它必需通过完整的名称来引用。

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

导入包时有一个可以选择的方式：

```
from Sound.Effects import echo
```

这样就加载了 echo 子模块，并且使得它在没有包前缀的情况下也可以使用，所以它可以如下方式调用：

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

还有另一种变体用于直接导入函数或变量：

```
from Sound.Effects.echo import echofilter
```

这样就又一次加载了 echo 子模块，但这样就可以直接调用它的 echofilter() 函数：

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

需要注意的是使用 from *package* import *item* 方式导入包时，这个子项（item）既可以是包中的一个子模块（或一个子包），也可以是包中定义的其它命名，像函数、类或变量。import 语句首先核对是否包中有这个子项，如果没有，它假定这是一个模块，并尝试加载它。如果没有找到它，会引发一个 ImportError 异常。

相反，使用类似 import *item.subitem.subsubitem* 这样的语法时，这些子项必须是包，最后的子项可以是包或模块，但不能是前面子项中定义的类、函数或变量。

6.4.1 从包中导入全部信息（Importing * From a Package）

那么当用户写下 from Sound.Effects import *时会发生什么事？理想中，总是希望在文件系统中找出包中所有的子模块，然后导入它们。不幸的是，这个操作在 Mac 和 Windows 平台上工作的并不太好，这些文件系统的文件大小写并不敏感！在这些平台上没有什么方法可以确保一个叫 ECHO.PY 的文件应该导入为模块 echo、Echo 或 ECHO。（例如，Windows 95 有一个讨厌的习惯，它会把所有的文件名都显示为首字母大写的风格。） DOS 8+3 文件名限制又给长文件名模块带来了另一个有趣的问题。

对于包的作者来说唯一的解决方案就是给提供一个明确的包索引。import 语句按如下条件进行转换：执行 from packae import * 时，如果包中的__init__.py 代码定义了一个名为__all__的链表，就会按照链表中给出的模块名进行导入。新版本的包发布时作者可以任意更新这个链表。如果包作者不想 import * 的时候导入他们的包中所有模块，那么也可能会决定不支持它（import *）。例如，Sounds/Effects/__init__.py 这个文件可能包括如下代码：

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

这意味着 from Sound.Effects import * 语句会从 Sound 包中导入以上三个已命名的子模块。

如果没有定义 __all__ ，from Sound.Effects import * 语句*不会*从 Sound.Effects 包中导入所有的子模块。Effects 导入到当前的命名空间，只能确定的是导入了 Sound.Effects 包（可能会运行 __init__.py 中的初始化代码）以及包中定义的所有命名会随之导入。这样就从__init__.py 中导入了每一个命名（以及明确导入的子模块）。同样也包括了前述的 import 语句从包中明确导入的子模块，考虑以下代码：

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

在这个例子中，echo 和 surround 模块导入了当前的命名空间，这是因为执行 from … import 语句时它们已经定义在 Sound.Effects 包中了（定义了__all__时也会同样工作）。

需要注意的是习惯上不主张从一个包或模块中用 import * 导入所有模块，因为这样的通常意味着可读性会很差。然而，在交互会话中这样做可以减少输入，相对来说确定的模块被设计成只导出确定的模式中命名的那一部分。

记住，from Package import specific_submodule 没有错误！事实上，除非导入的模块需要使用其它包中的同名子模块，否则这是受到推荐的写法。

6.4.2 内置包 （Intra-package）参考

子模块之间经常需要互相引用。例如，surround 模块可能会引用 echo 模块。事实上，这样的引用如此普遍，以致于 import 语句会先搜索包内部，然后才是标准模块搜索路径。因此 surround module 可以简单的调用 import echo 或者 from echo import echofilter。如果没有在当前的包中发现要导入的模块，import 语句会依据指定名寻找一个顶级模块。

如果包中使用了子包结构（就像示例中的 Sound 包），不存在什么从邻近的包中引用子模块的便捷方法－－必须使用子包的全名。例 如，如果 Sound.Filters.vocoder 包需要使用 Sound.Effects 包中的 echosa 模块，它可以使用 from Sound.Effects import echo。

6.4.3 多重路径中 的包

包支持一个另为特殊的变量， __path__。 在包的__init__.py 文件代码执行之前，该变量初始化一个目录名列表。该变量可以修改，它作用于包中的子包和模块的搜索功能。

这个功能可以用于扩展包中的模块集，不过它不常用。

Foo tnotes

... somewhere.[6.1]

事实上函数定义既是“声明”又是“可执行体”；执行体由函数在模块全局语义表中的命名导入。（In fact function definitions are also `statements' that are `executed'; the execution enters the function name in the module's global symbol table.）

7. 输入和 输出

有几种方法可以表现程序的输出结果；数据可以用可读的结构打印，也可以写入文件供以后使用。本章将会讨论几种可行的做法。

7.1 设计输出格 式

我们有两种大相径庭的输出值方法：表达式语句和 print 语句。（第三种访求是使用文件对象的 wite()方法，标准文件输出可以参考 sys.stdout。详细内容参见库参考手册。）

可能你经常想要对输出格式做一些比简单的打印空格分隔符更为复杂的控制。有两种方法可以格式化输出。第一种是由你来控制整个字符串，使用字符切片和联接操作就可以创建出任何你想要的输出形式。标准模块 string 包括了一些操作，将字符串填充入给定列时，这些操作很有用。随后我们会讨论这部分内容。第二种方法是使用 % 操作符，以某个字符串做为其左参数。 % 操作符将左参数解释为类似于 sprintf()风格的格式字符串，并作用于右参数，从该操作中返回格式化的字符串。

当然，还有一个问题，如何将（不同的）值转化为字符串？很幸运，Python 总是把任意值传入 repr() 或 str() 函数，转为字符串。相对而言引号（"）等价于 repr （），不过不提倡这样用。

函数 str() 用于将值转化为适于人阅读的形式，而 repr()转化为供解释器读取的形式（如果没有等价的语法，则会发生 SyntaxError 异常） 某对象没有适于人阅读的解释形式的话，str()会返回与 repr()等同的值。很多类型，诸如数值或链表、字典这样的结构，针对各函数都有着统一的解读方式。字符串和浮点数，有着独特的解读方式。

以下是一些示例：

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> 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'))"
>>> # reverse quotes are convenient in interactive sessions:
… `x, y, ('spam', 'eggs')`
"(32.5, 40000, ('spam', 'eggs'))"
```

以下两种方法可以输出平方和立方表:

```
>>> import string
>>> for x in range(1, 11):
…     print string.rjust(repr(x), 2), string.rjust(repr(x*x), 3),
…     # Note trailing comma on previous line
…     print string.rjust(repr(x*x*x), 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 '%2d %3d %4d' % (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
```

（需要注意的是使用 print 方法时每两列之间有一个空格：它总是在参数之间加一个空格。）

以上是一个 string.rjust()函数的演示，这个函数把字符串输出到一列，并通过向左侧填充空格来使其右对齐。类似的函数还有 string.ljust() 和 string.center()。这些函数只是输出新的字符串，并不改变什么。如果输出的字符串太长，它们也不会截断它，而是原样输出，这会使你的输出格式变得混乱，不过总强过另一种选择（截断字符串），因为那样会产生错误的输出值。（如果你确实需要截断它，可以使用切片操作，例如："string.ljust(x,n)[0:n]"。）

还有一个函数，string.zfill()它用于向数值的字符串表达左侧填充 0。该函数可以正确理解正负号：

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

可以如下这样使用 % 操作符：

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

如果有超过一个的字符串要格式化为一体，就需要将它们传入一个元组做为右值，如下所示：

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}

>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

大多数类 C 的格式化操作都需要你传入适当的类型，不过如果你没有定义异常，也不会有什么从内核中主动的弹出来。（however, if you don't you get an exception, not a core dump） 使用 %s 格式会更轻松些：如果对应的参数不是字符串，它会通过内置的 str() 函数转化为字符串。Python 支持用 * 作为一个隔离（整型的）参数来传递宽度或精度。Python 不支持 C 的 %n 和 %p 操作符。

如果可以逐点引用要格式化的变量名，就可以产生符合真实长度的格式化字符串，不会产生间隔。这一效果可以通过使用 form %(name)结构来实现：

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

这个技巧在与新的内置函数 vars() 组合使用时非常有用，该函数返回一个包含所有局部变量的字典。

7.2 读写文件

open() 返回一个文件对象，通常的用法需要两个参数：“open(*filename*, *mode*)”。

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

第一个参数是一个标识文件名的字符串。第二个参数是由有限的字母组成的字符串，描述了文件将会被如何使用。可选的模式有：‘r’，此选项使文件只读；‘w’，此选项使文件只写（对于同名文件，该操作使原有文件被覆盖）；‘a’，此选项以追加方式打开文件；‘r+’，此选项以读写方式打开文件；如果没有指定，默认为‘r’模式。

在 Windows 和 Macintosh 平台上，‘b’模式以二进制方式打开文件，所以可能会有类似于‘rb’，‘wb’，‘r+b’等等模式组合。 Windows 平台上文本文件与二进制文件是有区别的，读写文本文件时，行尾会自动添加行结束符。 这种后台操作方式对文本文件没有什么问题，但是操作 JPEG 或 EXE 这样的二进制文件时就会产生破坏。在操作这些文件时一定要记得以二进制模式打开。（需要注意的是 Mactiontosh 平台上的文本模式依赖于其使用的底层 C 库）。

7.2.1 文件对象（file object）的方法

本节中的示例都默认文件对象 f 已经创建。

要读取文件内容，需要调用 f.read(*size*)，该方法读取若干数量的数据并以字符串形式返回其内容，字符串长度为数值 *size* 所指定的大小。如果没有指定 size 或者指定为负数，就会读取并返回整个文件。当文件大小为当前机器内存两倍时，就会产生问题。正常情况下，会按 size 尽可能大的读取和返回数据。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() 从文件中读取单独一行，字符串结尾会自动加上一个换行符，只有当文件最后一行没有以换行符结尾时，这一操作才会被忽略。这样返回值就不会有什么混淆不清，如果 if f.readline() 返回一个空字符串，那就表示到达了文件末尾，如果是一个空行，就会描述为‘\n’，一个只包含换行符的字符串。

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

f.readlines() 返回一个列表，其中包含了文件中所有的数据行。如果给定了 *sizehint* 参数，就会读入多于一行的比特数，从中返回行列表。这个功能通常用于高效读取大型行文件，避免了将整个文件读入内存。这种操作只返回完整的行。

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

f.write(*string*) 将 *string* 的内容写入文件，返回 None。

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

f.tell() 返回一个整数，代表文件对象在文件中的指针位置，该数值计量了自文件开头到指针处的比特数。需要改变文件对象指针话话，使用“f.seek(*offset*, *from_what*)” 。指针在该操作中从指定的引用位置移动 *offset* 比特，引用位置由 *from_what* 参数指定。. *from_what* 值为 0 表示自文件起初处开始，1 表示自当前文件指针位置开始，2 表示自文件末尾开始。 *from_what* 可以乎略，其默认值为零，此时从文件头开始。

```
>>> f=open('/tmp/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'
```

文件使用完后，调用 f.close() 可以关闭文件，释放打开文件后占用的系统资源。调用 f.close()之后，再调用文件对象会自动引发错误。

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

文件对象还有一些不太常用的附加方法，比如 isatty() 和 truncate() 在库参考手册中有文件对象的完整指南。

7.2.2 pickle 模块

我们可以很容易的读写文件中的字符串。数值就要多费点儿周折，因为 read() 方法只会返回字符串，应该将其传入 string.atoi()方法中，就可以将'123' 这样的字符转为相应的值。不过，当你需要保存更为复杂的数据类型，例如链表、字典，类的实例，事情就会变得更复杂了。

好在用户不必要非得自己编写和调试保存复杂数据类型的代码。Python 提供了一个名为 Pickle 的标准模块。这是一个令人赞叹的模块，几乎可以把任何 Python 对象（甚至是一些 Python 代码块（form）！）表达为为字符串，这一过程称之为*封装*（*pickling*）。从字符串表达出重新构造对象称之为*拆封*（*unpickling*）。封装状态中的对象可以存储在文件或对象中，也可以通过网络在远程的机器之间传输。

如果你有一个对象 x，一个以写模式打开的文件对象 f，封装对像的最简单的方法只需要一行代码：

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

如果 f 是一个以读模式打开的文件对象，就可以重装拆封这个对象：

```
x = pickle.load(f)
```

（如果不想把封装的数据写入文件，这里还有一些其它的变化可用。完整的 pickle 文档请见库参考手册）。

pickle 是存储 Python 对象以供其它程序或其本身以后调用的标准方法。提供这一组技术的是一个持久化对象（ *persistent* object ）。因为 pickle 的用途很广泛，很多 Python 扩展的作者都非常注意类似矩阵这样的新数据类型是否适合封装和拆封。

8. 错误和 异常

至今为止还没有进一步的谈论过错误信息，不过在你已经试验过的那些例子中，可能已经遇到过一些。Python 中（至少）有两种错误：语法错误和异常（ *syntax errors* and *exceptions*）。

8.1 语法错 误

语法错误，也称作解析错误，可能是学习 Python 的过程中最容易犯的：

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

解析器会重复出错的行，并在行中最早发现的错误位置上显示一个小箭头。错误（至少是被检测到）就发生在箭头指向的位置。示例中的错误表现在关键字 print 上，因为在它之前少了一个冒号（:）。同时也会显示文件名和行号，这样你就可以知道错误来自哪个脚本，什么位置。

8.2 异常

即使是在语法上完全正确的语句，尝试执行它的时候，也有可能会发生错误。在程序运行中检测出的错误称之为异常，它通常不会导致致命的问题，你很快就会学到如何在 Python 程序中控制它们。大多数异常不会由程序处理，而是显示一个错误信息：

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo 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: cannot concatenate 'str' and 'int' objects
```

错误信息的最后一行指出发生了什么错误。异常也有不同的类型，异常类型做为错误信息的一部分显示出来：示例中的异常分别为零除错误（ ZeroDivisionError ），命名错误（ NameError）和类型错误（ TypeError ）。打印错误信息时，异常的类型作为异常的内置名显示。对于所有的内置异常都是如此，不过用户自定义异常就不一定了（尽管这是一个很有用的约定）。标准异常名是内置的标识（没有保留关键字）。

这一行后一部分是关于该异常类型的详细说明，这意味着它的内容依赖于异常类型。

错误信息的前半部分以堆栈的形式列出异常发生的位置。通常在堆栈中列出了源代码行，然而，来自标准输入的源码不会显示出来。

Python 库参考手册 列出了内置异常和它们的含义。

8.3 处理异常

通过编程可以处理指定的异常。以下的例子重复要求用户输入一个值，直到用户输入的是一个合法的整数为止。不过这个程序允许用户中断程序（使用 Control-C 或者其它操作系统支持的方法）。需要注意的是用户发出的中断会引发一个 KeyboardInterrupt 异常。

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

try 语句按如下方式工作：

- 首先，执行 *try 子句*（在 *try* 和 *except* 关键字之间的部分）。
- 如果没有异常发生，*except 子句* 在 try 语句执行完毕后就被忽略了。
- 如果在 try 子句执行过程中发生了异常，那么该子句其余的部分就会被忽略。如果异常匹配于 except 关键字后面指定的异常类型，就执行对应的 except 子句，忽略 try 子句的其它部分。然后继续执行 try 语句之后的代码。
- 如果发生了一个异常，在 except 子句中没有与之匹配的分支，它就会传递到上一级 try 语句中。如果最终仍找不到对应的处理语句，它就成为一个*未处理异常*，终止程序运行，显示提示信息。

一个 try 语句可能包含多个 except 子句，分别指定处理不同的异常。至多只会有一个分支被执行。 异常处理程序只会处理对应的 try 子句中发生的异常，在同一个 try 语句中，其他子句中发生的异常则不作处理。一个 except 子句可以在括号中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，把它当做一个通配项使用。一定要慎用这种方法，因为它很可能会屏蔽掉真正的程序错误，使人无法发现！它也可以用于打印一行错误信息，然后重新抛出异常（可以使调用者更好的处理异常）。

```
import string, sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

try … except 语句可以带有一个 *else 子句*， 该子句只能出现在所有 except 子句之后。当 try 语句没有抛出异常时，需要执行一些代码，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 子句比在 try 子句中附加代码要好，因为这样可以避免 try … except 意外的截获本来不属于它们保护的那些代码抛出的异常。

发生异常时，可能会有一个附属值，作为异常的参数存在。这个参数是否存在、是什么类型，依赖于异常的类型。

在异常名（列表）之后，也可以为 except 子句指定一个变量。这个变量绑定于一个异常实例，它存储在 instance.args 的参数中。为了方便起见，异常实例定义了 __getitem__

和 __str__ ，这样就可以直接访问过打印参数而不必引用 .args。

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception, inst:
...     print type(inst)     # the exception instance
...     print inst.args      # arguments stored in .args
...     print inst           # __str__ allows args to printed directly
...     x, y = inst          # __getitem__ allows args to be unpacked directly
...     print 'x =', x
...     print 'y =', y
...
<type 'instance'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

对于未处理的异常，如果它有一个参数，那做就会作为错误信息的最后一部分（“明细”）打印出来。

异常处理句柄不止可以处理直接发生在 try 子句中的异常，即使是其中（甚至是间接）调用的函数，发生了异常，也一样可以处理。例如：

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError, detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo
```

8.4 抛出异常

在发生了特定的异常时，程序员可以用 raise 语句强制抛出异常。例如：

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

第一个参数指定了所抛出异常的名称，第二个指定了异常的参数。

如果你决定抛出一个异常而不处理它，raise 语句可以让你很简单的重新抛出该异常。

```
>>> 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 用户自定义 异常

在程序中可以通过创建新的异常类型来命名自己的异常。异常类通常应该直接或间接的从 Exception 类派生，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, 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!'
```

异常类中可以定义任何其它类中可以定义的东西，但是通常为了保持简单，只在其中加入几个属性信息，以供异常处理句柄提取。如果一个新创建的模块中需要抛出几种不同的错误时，一个通常的作法是为该模块定义一个异常基类，然后针对不同的错误类型派生出对应的异常子类。

```
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
```

与标准异常相似，大多数异常的命名都以“Error”结尾。

很多标准模块中都定义了自己的异常，用以报告在他们所定义的函数中可能发生的错误。关于类的进一步信息请参见第 9章，“类”。

8.6 定义清 理行为

try 语句还有另一个可选的子句，目的在于定义在任何情况下都一定要执行的功能。例如：

```
>>> try:
...     raise KeyboardInterrupt
... finally:
```

```
...     print 'Goodbye, world!'
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

不管 try 子句中有没有发生异常， *finally 子句*都一定会被执行。如果发生异常，在 finally 子句执行完后它会被重新抛出。 try 子句经由 break 或 return 退出也一样会执行 finally 子句。

在 finally 子句中的代码用于释放外部资源（例如文件或网络连接），不管这些资源是否已经成功利用。

在 try 语句中可以使用若干个 except 子句或一个 finally 子句，但两者不能共存。

9. 类

Python 在尽可能不增加新的语法和语义的情况下加入了类机制。这种机制是 C++和 Python's Modula-3 的混合。Python 中的类没有在用户和定义之间建立一个绝对的屏障，而是依赖于用户自觉的不去“破坏定义”。然而，类机制最重要的功能都完整的保留下来。类继承机制允许多继承，派生类可以覆盖（override）基类中的任何方法，方法中可以调用基类中的同名方法。对象可以包含任意数量的私有成员。

用 C++术语来讲，所有的类成员（包括数据成员）都是*公有*（*public*）的，所有的成员函数都是*虚拟*（*virtual*）的。用 Modula-3 的术语来讲，在成员方法中没有什么简便的方式（shorthands）可以引用对象的成员：方法函数在定义时需要以引用的对象做为第一个参数，以调用时则会隐式引用对象。这样就形成了语义上的引入和重命名。（This provides semantics for importing and renaming.）但是，像 C++而非 Modula-3 中那样，大多数带有特殊语法的内置操作符（算法运算符、下标等）都可以针对类的需要重新定义。

9.1 有关术 语的话题

由于没有什么关于类的通用术语，我从 Smalltalk 和 C++中借用一些（我更希望用 Modula-3 的，因为它的面向对象机制比 C++更接近 Python，不过我想没多少读者听说过它）。

我要提醒读者，这里有一个面向对象方面的术语陷阱，在 Python 中“对象”这个词不一定指类实例。Python 中并非所有的类型都是类：例如整型、链表这些内置数据类型就不是，甚至某些像文件这样的外部类型也不是，这一点类似于 C++和 Modula-3，而不像 Smalltalk。然而，*所有*的 Python 类型在语义上都有一点相同之处：描述它们的最贴切词语是“对象”。

对象是被特化的，多个名字（在多个作用域中）可以绑定同一个对象。这相当于其它语言中的别名。通常对 Python 的第一印象中会忽略这一点，使用那些不可变的基本类型（数值、字符串、元组）时也可以很放心的忽视它。然而，在 Python 代 码调用字典、链表之类可变对象，以及大多数涉及程序外部实体（文件、窗体等等）的类型时，这一语义就会有影响。这通用有助于优化程序，因为别名的行为在某 些方面类似于指针。例如，很容易传递一个对象，因为在行为上只是传递了一个指针。如果函数修改了一个通过参数传递的对象，调用者可以接收到变化－－在 Pascal 中这需要两个不同的参数传递机制。

9.2 Python 作用域和命名空间

在介绍类之前，我首先介绍一些有关 Python 作用域的规则：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。

我们从一些定义开始。

命名空间是从命名到对象的映射。当前命名空间主要是通过 Python 字典实现的，不过通常不关心具体的实现方式（除非出于性能考虑），以后也有可能会改变其实现方式。以下有一些命名空间的例子：内置命名（像 abs() 这样的函数，以及内置异常名）集，模块中的全局命名，函数调用中的局部命名。某种意义上讲对象的属性集也是一个命名空间。关于命名空间需要了解的一件很重要的事就是不同命名空间中的命名没有任何联系，例如两个不同的模块可能都会定义一个名为“maximize”的函数而不会发生混淆——用户必须以模块名为前缀来引用它们。

顺便提一句，我称 Python 中任何一个“.”之后的命名为*属性*——例如，表达式 z.real 中的 real 是对象 z 的一个属性。严格来讲，从模块中引用命名是引用属性：表达式 modname.funcname 中，modname 是一个模块对象， funcname 是它的一个属性。因此，模块的属性和模块中的全局命名有直接的映射关系：它们共享同一命名空间！ [9.1]

属性可以是只读过或写的。后一种情况下，可以对属性赋值。你可以这样作：“modname.the_answer = 42”。可写的属性也可以用 del 语句删除。例如：“del modname.the_answer”会从 modname 对象中删除 the_answer 属性。

不同的命名空间在不同的时刻创建，有不同的生存期。包含内置命名的命名空间在 Python 解释器启动时创建，会一直保留，不被删除。模块的全局命名空间在模块定义被读入时创建，通常，模块命名空间也会一直保存到解释器退出。由解释器在最高层调用执行的语句，不管它是从脚本文件中读入还是来自交互式输入，都是__main__模块的一部分，所以它们也拥有自己的命名空间。（内置命名也同样被包含在一个模块中，它被称作__builtin__。）

当函数被调用时创建一个局部命名空间，函数反正返回过抛出一个未在函数内处理的异常时删除。（实际上，说是遗忘更为贴切）。当然，每一个递归调用拥有自己的命名空间。

作用域是指 Python 程序可以直接访问到的命名空间。“直接访问”在这里意味着访问命名空间中的命名时无需加入附加的修饰符。

尽管作用域是静态定义，在使用时他们都是动态的。每次执行时，至少有三个命名空间可以直接访问的作用域嵌套在一起：包含局部命名的使用域在最里面，首先被搜索；其次搜索的是中层的作用域，这里包含了同级的函数；最后搜索最外面的作用域，它包含内置命名。

如果一个命名声明为全局的，那么所有的赋值和引用都直接针对包含模全局命名的中级作用域。另外，从外部访问到的所有内层作用域的变量都是只读的。

从文本意义上讲，局部作用域引用当前函数的命名。在函数之外，局部作用域与全局使用域引用同一命名空间：模块命名空间。类定义也是局部作用域中的另一个命名空间。

作用域决定于源程序的文本：一个定义于某模块中的函数的全局作用域是该模块的命名空间，而不是该函数的别名被定义或调用的位置，了解这一点非常重要。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，所以不要依赖于动态解析！（事实上，局部变量已经是静态确定了。）

Python 的一个特别之处在于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。）

9.3 初识类

类引入了一点新的语法，三种新的对象类型，以及一些新的语义。

9.3.1 类定义 语法

最简单的类定义形式如下：

```
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。）

习惯上，类定义语句的内容通常是函数定义，不过其它语句也可以，有时会很有用——后面我们再回过头来讨论。类中的函数定义通常包括了一个特殊形式的参数列表，用于方法调用约定——同样我们在后面讨论这些。

定义一个类的时候，会创建一个新的命名空间，将其作为局部作用域使用——因此，所以对局部变量的赋值都引入新的命名空间。特别是函数定义将新函数的命名绑定于此。

类定义完成时（正常退出），就创建了一个*类对象*。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*使用和 Python 中所有的属性引用一样的标准语法： obj.name。类对象创建后，类命名空间中所有的命名都是有效属性名。所以如果类定义是这样：

```
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*使用函数符号。只要将类对象看作是一个返回新的类实例的无参数函数即可。例如（假设沿用前面的类）：

```
x = MyClass()
```

以上创建了一个新的类实例并将该对象赋给局部变量 x。

这个实例化操作（“调用”一个类对象）来创建一个空的对象。很多类都倾向于将对象创建为有初始状态的。因此类可能会定义一个名为__init__() 的特殊方法，像下面这样：

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

类定义了 __init__() 方法的话，类的实例化操作会自动为新创建的类实例调用 __init__() 方法。所以在下例中，可以这样创建一个新的实例：

```
x = MyClass()
```

当然，出于弹性的需要， __init__() 方法可以有参数。事实上，参数通过 __init__()传递到类的实例化操作上。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作什么？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。

第一种称作*数据属性*。这相当于 Smalltalk 中的“实例变量”或 C++中的“数据成员”。和局部变量一样，数据属性不需要声明，第一次使用时它们就会生成。例如，如果 x 是前面创建的 MyClass 实例，下面这段代码会打印出 16 而不会有任何多余的残留：

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

第二种为实例对象所接受的引用属性是*方法*。方法是*属于*一个对象的函数。（在 Python 中，方法不止是类实例所独有：其它类型的对象也可有方法。例如，链表对象有 append，insert，remove，sort 等等方法。然而，在这里，除非特别说明，我们提到的方法特指类方法）

实例对象的有效名称依赖于它的类。按照定义，类中所有（用户定义）的函数对象对应它的实例中的方法。所以在我们的例子中，x..f 是一个有效的方法引用，因为 MyClass.f 是一个函数。但 x.i 不是，因为 MyClass.i 是不是函数。不过 x.f 和 MyClass.f 不同－－它是一个*方法对象*，不是一个函数对象。

9.3.4 方法对象

通常方法是直接调用的：

```
x.f()
```

在我们的例子中，这会返回字符串‘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，它可以存储起来以后调用。例如：

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

会不断的打印“Hello world”。

调用方法时发生了什么？你可能注意到调用 x.f() 时没有引用前面标出的变量，尽管在 f 的函数定义中指明了一个参数。这个参数怎么了？事实上如果函数调用中缺少参数，Python 会抛出异常——甚至这个参数实际上没什么用……

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作为函数的第一个参数传给了函数。在我们的例子中，调用 x.f() 相当于 MyClass.f(x)。通常，以 n 个参数的列表去调用一个方法就相当于将方法的对象插入到参数列表的最前面后，以这个列表去调用相应的函数。

如果你还是不理解方法的工作原理，了解一下它的实现也许有帮助。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。

9.4 一些说明

〔有些内容可能需要明确一下……〕

同名的数据属性会覆盖方法属性，为了避免可能的命名冲突——这在大型程序中可能会导致难以发现的 bug——最好以某种命名约定来避免冲突。可选的约定包括方法的首字母大写，数据属性名前缀小写（可能只是一个下划线），或者方法使用动词而数据属性使用名词。

数据属性可以由方法引用，也可以由普通用户（客户）调用。换句话说，类不能实现纯的数据类型。事实上，Python 中没有什么办法可以强制隐藏数据——一切都基本约定的惯例。

（另一方法讲，Python 的实现是用 C 写成的，如果有必要，可以用 C 来编写 Python 扩展，完全隐藏实现的细节，控制对象的访问。）

客户应该小心使用数据属性－－客户可能会因为随意修改数据属性而破坏了本来由方法维护的数据一致性。需要注意的是，客户只要注意避免命名冲突，就可以随意向实例中添加数据属性而不会影响方法的有效性－－再次强调，命名约定可以省去很多麻烦。

从方法内部引用数据属性（以及其它方法！）没有什么快捷的方式。我认为这事实上增加了方法的可读性：即使粗略的浏览一个方法，也不会有混淆局部变量和实例变量的机会。

习惯上，方法的第一个参数命名为 self。这仅仅是一个约定：对 Python 而言，self 绝对没有任何特殊含义。（然而要注意的是，如果不遵守这个约定，别的 Python 程序员阅读你的代码时会有不便，而且有些类浏览程序也是遵循此约定开发的。）

类属性中的任何函数对象在类实例中都定义为方法。不是必须要将函数定义代码写进类定义中，也可以将一个函数对象赋给类中的一个变量。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 的属性，引用的都是函数对象，因此它们都是 C 实例的方法－－h 严格等于 g。要注意的是这种习惯通常只会迷惑程序的读者。

通过 self 参数的方法属性，方法可以调用其它的方法：

```
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)
```

方法可以像引用普通的函数那样引用全局命名。与方法关联的全局作用域是包含类定义的模块。（类本身永远不会做为全局作用域使用！）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！

9.5 继承

当然，如果一种语言不支持继承就，“类”就没有什么意义。派生类的定义如下所示：

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

命名 BaseClassName （示例中的基类名）必须与派生类定义在一个作用域内。除了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用：

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

派生类定义的执行过程和基类是一样的。构造派生类对象时，就记住了基类。这在解析属性引用的时候尤其有用：如果在类中找不到请求调用的属性，就搜索基类。如果基类是由别的类派生而来，这个规则会递归的应用上去。

派生类的实例化没有什么特殊之处：DerivedClassName() （示列中的派生类）创建一个新的类实例。方法引用按如下规则解析：搜索对应的类属性，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。

派生类可能会覆盖其基类的方法。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++程序员来说，Python 中的所有方法本质上都是虚方法。）

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)”。有时这对于客户也很有用。（要注意的中只有基类在同一全局作用域定义或导入时才能这样用。）

9.5.1 多继承

Python 同样有限的支持多继承形式。多继承的类定义形如下例：

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

这里唯一需要解释的语义是解析类属性的规则。顺序是深度优先，从左到右。因此，如果在 DerivedClassName （示例中的派生类）中没有找到某个属性，就会搜索 Base1 ，然后（递归的）搜索其基类，如果最终没有找到，就搜索 Base2，以此类推。

（有些人认为广度优先——在搜索 Base1 的基类之前搜索 Base2 和 Base3——看起来更为自然。然而，如果 Base1 和 Base2 之间发生了命名冲突，你需要了解这个属性是定义于 Base1 还是 Base1 的基类中。而深度优先不区分属性继承自基类还是直接定义。）

显然不加限制的使用多继承会带来维护上的噩梦，因为 Python 中只依靠约定来避免命名冲突。多继承一个很有名的问题是派生继承的两个基类都是从同一个基类继承而来。目前还不清楚这在语义上有什么意义，然而很容易想到这会造成什么后果（实例会有一个独立的“实例变量”或数据属性复本作用于公共基类。）

9.6 私有变量

Python 对类的私有成员提供了有限的支持。任何形如__spam （以至少双下划线开头，至多单下划线结尾）随即都被替代为 _classname__spam， 去掉前导下划线的 classname 即当前的类名。这种混淆不关心标识符的语法位置，所以可用来定义私有类实例和类变量、方法，以及全局变量，甚至于将*其它*类的实例保存为私有变量。混淆名长度超过 255 个字符的时候可能会发生截断。在类的外部，或类名只包含下划线时，不会发生截断。

命名混淆意在给出一个在类中定义“私有”实例变量和方法的简单途径，避免派生类的实例变量定义产生问题，或者与外界代码中的变量搞混。要注意的是混 淆规则主要目的在于避免意外错误，被认作为私有的变量仍然有可能被访问或修改。在特定的场合它也是有用的，比如调试的时候，这也是一直没有堵上这个漏洞的 原因之一（小漏洞：派生类和基类取相同的名字就可以使用基类的私有变量。）

要注意的是传入 exec，eval()或 evalfile()的代码不会将调用它们的类视作当前类，这与 global 语句的情况类似，global 的作用局限于“同一批”进行字节编译的代码。同样的限制也适用于 getattr() ，setattr()和 delattr()，以及直接引用__dict__的时候。

9.7 补充

有时类似于 Pascal 中“记录（record）”或 C 中“结构（struct）”的数据类型很有用，它将一组已命名的数据项绑定在一起。一个空的类定义可以很好的实现这它：

```
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，事实上这模仿了该类的方法。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。

实例方法对象也有属性： m.im_self 是一个实例方法所属的对象，而 m.im_func 是这个方法对应的函数对象。

9.8 异常也 是类

用户自定义异常也可以是类。利用这个机制可以创建可扩展的异常体系。

以下是两种新的有效（语义上的）异常抛出形式：

```
raise Class, instance

raise instance
```

第一种形式中，instance 必须是 Class 或其派生类的一个实例。第二种形式是以下形式的简写：

```
raise instance.__class__, instance
```

发生的异常其类型如果是异常子句中列出的类，或者是其派生类，那么它们就是相符的（反过来说－－发生的异常其类型如果是异常子句中列出的类的基类，它们就不相符）。例如，以下代码会按顺序打印 B，C，D：

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

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

要注意的是如果异常子句的顺序颠倒过来（“execpt B”在最前），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
```

这种形式的访问清晰、简洁、方便。这种迭代器的用法在 Python 中普遍而且统一。在后台，for 语句在容器对象中调用 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() 抛出一个 StopIteration 异常通知 for 语句循环结束。以下是其工作原理的示例：

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

```
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    it.next()
StopIteration
```

了解了迭代器协议的后台机制，就可以很容易的给自己的类添加迭代器行为。定义一个 __iter__() 方法，使其返回一个带有 next() 方法的对象。如果这个类已经定义了 next()，那么 __iter__() 只需要返回 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]

>>> for char in Reverse('spam'):
        print char

m
a
p
s
```

9.10 发生器

发生器是创建迭代器的简单而强大的工具。它们写起来就像是正则函数，需要返回数据的时候使用 yield 语句。每次 next() 被调用时，生成器回复它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）。以下示例演示了发生器可以很简单的创建出来：

```
>>> 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
```

前一节中描述了基于类的迭代器，它能作的每一件事发生器也能作到。因为自动创建了 __iter__() 和 next() 方法，发生器显得如此简洁。

另外一个关键的功能是两次调用之间的局部变量和执行情况都自动保存了下来。这样函数编写起来就比手动调用 self.index 和 self.data 这样的类变量容易的多。

除了创建和保存程序状态的自动方法，当发生器终结时，还会自动抛出 StopIteration 异常。综上所述，这些功能使得编写一个正则函数成为创建迭代器的最简单方法。

Foo tnotes

... 命名空间![9.1]

有一个例外。模块对象有一个隐秘的只读对象，名为 __dict__ ，它返回用于实现模块命名空间的字典，命名 __dict__ 是一个属性而非全局命名。显然，使用它违反了命名空间实现的抽象原则，应该被严格限制于调试中。

10. What Now?

Reading this tutorial has probably reinforced your interest in using Python -- you should be eager to apply Python to solve your real-world problems. Now what should you do?

You should read, or at least page through, the *Python Library Reference*, which gives complete (though terse) reference material about types, functions, and modules that can save you a lot of time when writing Python programs. The standard Python distribution includes a *lot* of code in both C and Python; there are modules to read UNIX mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, write CGI programs, compress data, and a lot more; skimming through the Library Reference will give you an idea of what's available.

The major Python Web site is http://www.python.org/; it contains code, documentation, and pointers to Python-related pages around the Web. This Web site is mirrored in various places around the world, such as Europe, Japan, and Australia; a mirror may be faster than the main site, depending on your geographical location. A more informal site is http://starship.python.net/, which contains a bunch of Python-related personal home pages; many people have downloadable software there. Many more user-created Python modules can be found in a third-party repository at http://www.vex.net/parnassus.

For Python-related questions and problem reports, you can post to the newsgroup comp.lang.python, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are around 120 postings a day (with peaks up to several hundred), asking (and answering) questions, suggesting new features, and announcing new modules. Before posting, be sure to check the list of Frequently Asked Questions (also called the FAQ), at http://www.python.org/doc/FAQ.html, or look for it in the Misc/ directory of the Python source distribution. Mailing list archives are available at http://www.python.org/pipermail/. The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.

A. Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the *GNU Readline* library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the UNIX and CygWin versions of the interpreter.

This chapter does *not* document the editing facilities of Mark Hammond's PythonWin package or the Tk-based environment, IDLE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.

A.1 Line Editing

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: C-A (Control-A) moves the cursor to the beginning of the line, C-E to the end, C-B moves it one position to the left, C-F to the right. Backspace erases the character to the left of the cursor, C-D the character to its right. C-K kills (erases) the rest of the line to the right of the cursor, C-Y yanks back the last killed string. C-underscore undoes the last change you made; it can be repeated for cumulative effect.

A.2 History Substitution

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. C-P moves one line up (back) in the history

buffer, C-N moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the Return key passes the current line to the interpreter. C-R starts an incremental reverse search; C-S starts a forward search.

A.3 Key Bindings

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called ~/.inputrc. Key bindings have the form

```
key-name: function-name
```

or

```
"string": function-name
```

and options can be set with

```
set option-name value
```

For example:

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Note that the default binding for Tab in Python is to insert a Tab character instead of Readline's default filename completion function. If you insist, you can override this by putting

```
Tab: complete
```

in your ~/.inputrc. (Of course, this makes it harder to type indented continuation lines.)

Automatic completion of variable and module names is optionally available. To enable it in the interpreter's interactive mode, add the following to your startup file:

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

This binds the Tab key to the completion function, so hitting the Tab key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as string.a, it will evaluate the the expression up to the final "." and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a __getattr__() method is part of the expression.

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environments. You may find it convenient to keep some of the imported modules, such as os, which turn out to be needed in most sessions with the interpreter.

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it:  "export PYTHONSTARTUP=/max/home/itamar/.pystartup" in bash.
#
# Note that PYTHONSTARTUP does *not* expand "~", so you have to put in the
# full path to your home directory.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
```

```
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

A.4 Commentary

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

Footnotes

… file:[A.1]

Python will execute the contents of a file identified by the PYTHONSTARTUP environment variable when you start an interactive interpreter.

Subsections

- B.1 Representation Error

B. Floating Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction

```
0.125
```

has value 1/10 + 2/100 + 5/1000, and in the same way the binary fraction

0.001

has value 0/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction 1/3. You can approximate that as a base 10 fraction:

0.3

or, better,

0.33

or, better,

0.333

and so on. No matter how many digits you're willing to write down, the result will never be exactly 1/3, but will be an increasingly better approximation to 1/3.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, 1/10 is the infinitely repeating fraction

0.0001100110011001100110011001100110011001100110011…

Stop at any finite number of bits, and you get an approximation. This is why you see things like:

```
>>> 0.1
0.10000000000000001
```

On most machines today, that is what you'll see if you enter 0.1 at a Python prompt. You may not, though, because the number of bits used by the hardware to store floating-point values can vary across machines, and Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal

value of the binary approximation stored for 0.1, it would have to display

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

instead! The Python prompt (implicitly) uses the builtin repr() function to obtain a string version of everything it displays. For floats, repr(*float*) rounds the true decimal value to 17 significant digits, giving

```
0.10000000000000001
```

repr(*float*) produces 17 significant digits because it turns out that's enough (on most machines) so that eval(repr(*x*)) == *x* exactly for all finite floats *x*, but rounding to 16 digits is not enough to make that true.

Note that this is in the very nature of binary floating-point: this is not a bug in Python, it is not a bug in your code either, and you'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

Python's builtin str() function produces only 12 significant digits, and you may wish to use that instead. It's unusual for eval(str(*x*)) to reproduce *x*, but the output may be more pleasant to look at:

```
>>> print str(0.1)
0.1
```

It's important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly 1/10, you're simply rounding the *display* of the true machine value.

Other surprises follow from this one. For example, after seeing

```
>>> 0.1
0.10000000000000001
```

you may be tempted to use the round() function to chop it back to the single digit you expect. But that makes no difference:

```
>>> round(0.1, 1)
0.10000000000000001
```

The problem is that the binary floating-point value stored for "0.1" was already the best possible binary approximation to 1/10, so trying to round it again can't make it better: it was already as good as it gets.

Another consequence is that since 0.1 is not exactly 1/10, adding 0.1 to itself 10 times may not yield exactly 1.0, either:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.99999999999999989
```

Binary floating-point arithmetic holds many surprises like this. The problem with "0.1" is explained in precise detail below, in the "Representation Error" section. See *The Perils of Floating Point* for a more complete account of other common surprises.

As that says near the end, ``there are no easy answers." Still, don't be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic, and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. str() usually suffices, and for finer control see the discussion of Pythons's % format operator: the %g, %f and %e format codes supply flexible and easy ways to round float results for display.

B.1 Representation Error

This section explains the ``0.1" example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect:

```
>>> 0.1
0.10000000000000001
```

Why is that? 1/10 is not exactly representable as a binary fraction. Almost all machines today (November 2000) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 "double precision". 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form *J*/2**_N_ where *J* is an integer containing exactly 53 bits. Rewriting

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

as

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

and recalling that *J* has exactly 53 bits (is >= 2**52 but < 2**53), the best value for *N* is 56:

```
>>> 2L**52
4503599627370496L
>>> 2L**53
9007199254740992L
>>> 2L**56/10
7205759403792793L
```

That is, 56 is the only value for *N* that leaves *J* with exactly 53 bits. The best possible value for *J* is then that quotient rounded:

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

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

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

Therefore the best possible approximation to 1/10 in 754 double precision is that over 2**56, or

```
7205759403792794 / 72057594037927936
```

Note that since we rounded up, this is actually a little bit larger than 1/10; if we had not rounded up, the quotient would have been a little bit smaller than

1/10. But in no case can it be *exactly* 1/10!

So the computer never ``sees'' 1/10: what it sees is the exact fraction given above, the best 754 double approximation it can get:

```
>>> .1 * 2L**56
7205759403792794.0
```

If we multiply that fraction by 10**30, we can see the (truncated) value of its 30 most significant decimal digits:

```
>>> 7205759403792794L * 10L**30 / 2L**56
100000000000000005551115123125L
```

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.100000000000000005551115123125. Rounding that to 17 significant digits gives the 0.10000000000000001 that Python displays (well, will display on any 754-conforming platform that does best-possible input and output conversions in its C library -- yours may not!).

Subsections

C. History and License

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs

team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AG REEMENT FOR PYTHON 2. 3

1. This LICENSE AGREEMENT is between the Python Software Foundation (``PSF''), and the Individua1 or Organization (``Licensee'') accessing and otherwise using Python 2.3 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.3 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., ``Copyright © 2001-2003 Python Software Foundation; A11 Rights Reserved'' are retained in Python 2.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.3.

4. PSF is making Python 2.3 available to Licensee on an ``AS IS'' basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement wi11 automatica11y terminate upon a materia1 breach of its terms and conditions.

7. Nothing in this License Agreement sha11 be deemed to create any

relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPE N.COM LICENSE AGREEMENT FOR PYTH ON 2.0

BEOPE N PYTHO N OPEN SO UR CE LICENSE AG REEMENT VERSIO N 1

1. This LICENSE AGREEMENT is between BeOpen.com (``BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (``Licensee") accessing and otherwise using this software in source or binary form and its associated documentation (``the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an ``AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and

Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the ``BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LI CENSE AG REEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (``CNRI"), and the Individual or Organization (``Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., ``Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): ``Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an ``AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY

REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the ``ACCEPT'' button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or

publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

About this document ...

Python Tutorial , July 29, 2003, Release 2.3

This document was generated using the LaTeX2HTML translator.

LaTeX2HTML is Copyright © 1993, 1994, 1995, 1996, 1997, Nikos Drakos, Computer Based Learning Unit, University of Leeds, and Copyright © 1997, 1998, Ross Moore, Mathematics Department, Macquarie University, Sydney.

The application of LaTeX2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.

Comments and Questions

General comments and questions regarding this document should be sent by email to python-docs@python.org. If you find specific errors in this document, either in the content or the presentation, please report the bug at the Python Bug Tracker at SourceForge.

Questions regarding how to use the information in this document should be sent to the Python news group, comp.lang.python, or the Python mailing list (which is gated to the newsgroup and carries the same content).

For any of these channels, please be sure not to send HTML email. Thanks.