Python 3.3
入门指南

Release:	3.3
Date:	January 19, 2013
来源:	www.pythondoc.com

Python 是一门简单易学且功能强大的编程语言。 它拥有高效的高级数据结构，并且能够用简单而又高效的方式进行面向对象编程。 Python 优雅的语法和动态类型，再结合它的解释性，使其在大多数平台的许多领域成为编写脚本或开发应用程序的理想语言。

你可以自由的从 Python 官方点，http://www.python.org，以源代码或二进制形式获取 Python 解释器及其标准扩展库，并可以自由的分发。 此站点同时也提供了大量的第三方 Python 模块、程序和工具，及其附加文档。

你可以很容易的使用 C 或 C++（其他可以通过 C 调用的语言）为 Python 解释器扩展新函数和数据类型。 Python 还可以被用作定制应用程式的一门扩展语言。

本手册非正式的向读者介绍了 Python 语言及其体系相关的基本知识与概念。 在学习实践中结合使用 Python 解释器是很有帮助的，不过所有的例子都是完整的，所以本手册亦可离线阅读。

如果需要了解相关标准库或对象的详细介绍，请查阅 Python 参考文档 。 Python 参考手册 提供了更多语言相关的正式说明。 如果想要使用 C 或 C++编写扩展，请查阅 Python 解释器扩展和集成章节 和 C API 参考手册 。 当然也可阅读一些深入介绍 Python 知识的图书。

本手册不会尝试涵盖 Python 的全部知识和每个特性，甚至不会涵盖所有常用的特性。 相反的，它介绍了 Python 中许多最引人瞩目的特性，并且会给你一个关于语言特色和风格的认识。 读完之后，你将能够阅读和编写 Python 模块或程序，并为以后使用 Python 参考手册 继续学习诸多 Python 模块库做好准备。

1. 开胃菜

如果你要用计算机做很多工作，最后你会发现有一些任务你更希望用自动化的方式进行处理。 比如，你想要在大量的文本文件中执行查找/替换，或者以复杂的方式对大量的图片进行重命名和整理。 也许你想要编写一个小型的自定义数据库、一个特殊的 GUI 应用程式或一个简单的小游戏。

如果你是一名专业的软件开发者，可能你必须使用几种 C/C++/JAVA 类库，并且发现通常编写/编译/测试/重新编译的周期是如此漫长。 也许你正在为这些类库编写测试用例，但是发现这是一个让人烦躁的工作。 又或者你已经完成了一个可以使用扩展语言的程式，但你并不想为此重新设计并实现一套全新的语言。

那么 Python 正是你所需要的语言。

虽然你能够通过编写 Unix shell 脚本或 Windows 批处理文件来处理其中的某些任务，但 Shell 脚本更适合移动文件或修改文本数据，并不适合编写 GUI 应用程序或游戏； 虽然你能够使用 C/C++/JAVA 编写程序，但即使编写一个简单的 first-draft 程序也有可能耗费大量的开发时间。 相比之下，Python 更易于使用，无论在 Windows、Mac OS X 或 Unix 操作系统上它都会帮助你更快的完成任务。

虽然 Python 易于使用，但它却是一门完整的编程语言；与 Shell 脚本或批处理文件相比，它为编写大型程序提供了更多的结构和支持。 另一方面，Python 提供了比 C 更多的错误检查，并且作为一门 * 高级语言* ，它内置支持高级的数据结构类型，例如：灵活的数组和字典。 因其更多的通用数据类型，Python 比 Awk 甚至 Perl 都适用于更的多问题领域，至少大多数事情在 Python 中与其他语言同样简单。

Python 允许你将程序分割为不同的模块，以便在其他的 Python 程序中重用。 Python 内置提供了大量的标准模块，你可以将其用作程序的基础，或者作为学习 Python 编程的示例。 这些模块提供了诸如文件 I/O、系统调用、sockets 支持，甚至类似 Tk 的用户图形界面（GUI）工具包接口。

Python 是一门解释型语言，因为无需编译和链接，你可以在程式开发中节省宝贵的时间。 Python 解释器可以交互的使用，这使得试验语言的特性、编写临时程序或在自底向上的程序开发中测试方法非常容易。 你甚至还可以把它当做一个桌面计算器。

Python 让程序编写的紧凑和可读。 用 Python 编写的程式通常比同样的 C、C++或 Java 程式更短小，这是因为以下几个原因：

- 高级数据结构使你可以在一条语句中表达复杂的操作；
- 语句组使用缩进代替开始和结束大括号来组织；
- 变量或参数无需声明。

Python 是 *可扩展* 的：如果你会 C 语言编程便可以轻易的为解释器添加内置函数或模块，或者为了对性能瓶颈作优化，或者将 Python 程序与只有二进制形式的库（比如某个专业的商业图形库）连接起来。 一旦你真正掌握了它，你可以将 Python 解释器集成进某个 C 应用程序，并把它当做那个程序的扩展或命令行语言。

顺便说一句，这个语言的名字来自于 BBC 的“Monty Python’s Flying Circus”节目，和爬行类动物没有任何关系。 在文档中引用 Monty Python 的典故不仅可行，而且值得鼓励！

现在你已经为 Python 兴奋不已了吧，大概想要领略一些更多的细节！ 学习一门语言最好的方法就是使用它，本指南推荐你边读边使用 Python 解释器练习。

下一节中，我们将解释 Python 解释器的用法。 这是很简单的一件事情，但它有助于试验后面的例子。

本手册剩下的部分将通过示例介绍 Python 语言及系统的诸多特性，开始是简单的语法、数据类型和表达式，接着介绍函数与模块，最后涉及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 Python 解释器

Python 解释器通常被安装在目标机器的 /usr/local/bin/python3.3 目录下。 将 /usr/local/bin 目录包含进 Unix shell 的搜索路径里，以确保可以通过输入

```
python3.3
```

命令来启动他。[1] 由于 Python 解释器的安装路径是可选的，这也可能是其他路径，你可以联系安装 Python 的用户或系统管理员确认。 （例如， /usr/local/python 就是一个常见的选择）
在 Windows 机器上，Python 通常安装在 C:\Python33 位置，当然你可以在运行安装向导时修改此值。 要想把此目录添加到你的 PATH 环境变量中，你可以在 DOS 窗口中输入以下命令

```
set path=%path%;C:\python33
```

通常你可以在主窗口输入一个文件结束符（Unix 系统是 Control-D ，Windows 系统是 Control-Z ）让解释器以 0 状态码退出。 如果那没有作用，你可以通过输入 quit() 命令退出解释器。
Python 解释器具有简单的行编辑功能。 在 Unix 系统上，任何 Python 解释器都可能已经添加了 GNU readline 库支持，这样就具备了精巧的交互编辑和历史记录等功能。 在 Python 主窗口中输入 Control-P 可能是检查是否支持命令行编辑的最简单的方法。 如果发出嘟嘟声（计算机扬声器），则说明你可以使用命令行编辑功能；更多快捷键的介绍请参考 交互的输入编辑和历史记录 。如果没有任何声音，或者显示 ^P 字符，则说明命令行编辑功能不可用；你只能通过退格键从当前行删除已键入的字符并重新输入。
Python 解释器有些操作类似 Unix shell： 当使用终端设备（tty）作为标准输入调用时，它交互的解释并执行命令； 当使用文件名参数或以文件作为标准输入调用时，它读取文件并将文件作为 脚本 执行。
第二种启动 Python 解释器的方法是 python -c command [arg] ... ，这种方法可以在 命令行 执行 Python 语句，类似于 shell 中的 -c 选项。 由于 Python 语句通常会包含空格或其他特殊 shell 字符，一般建议将 命令 用单引号包裹起来。
有一些 Python 模块也可以当作脚本使用。 你可以使用 python -m module [arg] ... 命令调用它们，这类似在命令行中键入完整的路径名执行 模块 源文件一样。
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

2.1.1. 参数传递

调用解释器时，脚本名和附加参数传入一个名为 sys.argv 的字符串列表。你能够获取这个列表通过执行 importsys，列表的长度大于等于 1；没有给定脚本和参数时，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] 被设定为 '-' ，使用 -c 指令 时， sys.argv[0] 被设定为'-c' 。 使用 -m 模块 参数时， sys.agv[0] 被设定为指定模块

的全名。-c 指令 或者 -m 模块 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:

```
$ python3.3
Python 3.3 (py3k, Sep 12 2007, 12:21:02)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on linux2
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 或者 DEL）就会取消当前输入，回到主命令行。 [2]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 类的 Unix 系统中，Python 脚本可以像 Shell 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 python3.3
```

(要确认 Python 解释器在用户的 **PATH** 中) #! 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 Windows （ 'rn' ） 的结束符。注意， '#' 是 Python 中是行注释的起始符。
脚本可以通过 **chmod** 命令指定执行模式和权限

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

Windows 系统上没有“执行模式”。 Python 安装程序自动将 .py 文件关联到 python.exe ，所以在

Python 文件图标上双击，它就会作为脚本执行。同样 .pyw 也作了这样的关联，通常它执行时不会显示控制台窗口。

2.2.3. 源程序编码

默认情况下，Python 源文件是 UTF-8 编码。 在此编码下，全世界大多数语言的字符可以同时用在字符串、标识符和注释中 — 尽管 Python 标准库仅使用 ASCII 字符做为标识符，这只是任何可移植代码应该遵守的约定。 如果要正确的显示所有的字符，你的编辑器必须能识别出文件是 UTF-8 编码，并且它使用的字体能支持文件中所有的字符。
你也可以为源文件指定不同的字符编码。 为此，在 #! 行（首行）后插入至少一行特殊的注释行来定义源文件的编码。:

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

通过此声明，源文件中所有的东西都会被当做用 encoding 指代的 UTF-8 编码对待。 在 Python 库参考手册 **codecs** 一节中你可以找到一张可用的编码列表。
例如，如果你的编辑器不支持 UTF-8 编码的文件，但支持像 Windows-1252 的其他一些编码，你可以定义:

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

这样就可以在源文件中使用 Windows-1252 字符集中的所有字符了。 这个特殊的编码注释必须在文件中的 第一或第二 行定义。

2.2.4. 交互执行文件

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 /dev/tty 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，可以在全局启动文件中加入类似以下的代码： ifos.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):
    exec(open(filename).read())
```

2.2.5. 本地化模块

Python 提供了两个钩子（方法）来本地化: **sitecustomize** 和 **usercustomize**. 为了见识它们, 你首先需要找到你的 site-packages 的目录. 启动 python 执行下面的代码:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.2/site-packages'
```

现在你可以在 site-packages 的目录下创建 usercustomize.py 文件，内容就悉听尊便了。 这个文

件将会影响 python 的每次调用，除非启动的时候加入 -s 选项禁止自动导入。
sitecustomize 的工作方式一样，但是是由电脑的管理账户创建以及在 **usercustomize** 之前导入。 具体可以参见 **site** 。

Footnotes

[1] 在 Unix 系统上，Python 3.1 解释器默认未被安装成名为 python 的命令，所以它不会与同时安装在系统中的 Python 2.x 命令冲突。

[2] GNU Readline 包的一个问题可能禁止此功能。

3. Python 简介

下面的例子中，输入和输出分别由大于号和句号提示符（ >>> 和 ... ）标注：如果想重现这些例子，就要在解释器的提示符后，输入（提示符后面的）那些不包含提示符的代码行。需要注意的是在练习中遇到的从属提示符表示你需要在最后多输入一个空行，解释器才能知道这是一个多行命令的结束。
本手册中的很多示例——包括那些带有交互提示符的——都含有注释。Python 中的注释以 # 字符起始，直至实际的行尾（译注——这里原作者用了 physical line 以表示实际的换行而非编辑器的自动换行）。注释可以从行首开始，也可以在空白或代码之后，但是不出现在字符串中。文本字符串中的 # 字符仅仅表示 # 。代码中的注释不会被 Python 解释，录入示例的时候可以忽略它们。
如下示例:

```
# 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. 将 Python 当做计算器

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

3.1.1. 数字

解释器的表示就像一个简单的计算器：可以向其录入一些表达式，它会给出返回值。表达式语法很直白：运算符+ ， - ， * 和 / 与其它语言一样（例如： Pascal 或 C）；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
5.0
>>> 8/5 # Fractions aren't lost when dividing integers
1.6
```

注意：有时你可能会得到不同的结果；浮点数在不同机器上的运算结果可能是不同的。 后面我们将对控制浮点数输出的显示结果做更多说明；这里我们看到的仅是有效的显示，并非我们能得到的可读性更好的结果。

对整数做除法运算并想去除小数部分取得整数结果时，可以使用另外一个运算符， `//`

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

等号（ `'='` ）用于给变量赋值:

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

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
```

变量在使用前必须“定义”（赋值），否则会出错:

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

浮点数有完整的支持；与整型混合计算时会自动转为浮点数:

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

复数也得到支持；带有后缀 `j` 或 `J` 就被视为虚数。带有非零实部的复数写为 `(real+imagj)` ，或者可以用 `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)
```

复数的实部和虚部总是记为两个浮点数。要从复数 z 中提取实部和虚部，使用 z.real 和 z.imag 。

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

浮点数和整数之间的转换函数（ **float()** 和 **int()** 以及 **long()** ） 不能用于复数。没有什么正确方法可以把一个复数转成一个实数。函数 abs(z) 用于获取其模（浮点数）或 z.real 获取其实部:

```
>>> 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 abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
```

交互模式中，最近一个表达式的值赋给变量 _ 。这样我们就可以把它当作一个桌面计算器，很方便的用于连续计算，例如:

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

此变量对于用户是只读的。不要尝试给它赋值 —— 你只会创建一个独立的同名局部变量，它屏蔽了系统内置变量的魔术效果.

3.1.2. 字符串

相比数值，Python 也提供了可以通过几种不同方式传递的字符串。它们可以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.'
```

Python 解释器按照字符串被输入的方式打印字符串结果：为了显示准确的值，字符串包含在成对的引号中，引号和其他特殊字符要用反斜线（ ）转译。 如果字符串只包含单引号（ ‘ ）而没有双引号（ ” ）就可以用双引号（ ” ）包围，反之用单引号（ ‘ ）包围。 再强调一下， **print** 函数可以生成可读性更好的输出。
字符串文本有几种方法分行。可以使用反斜杠为行结尾的连续字符串，它表示下一行在逻辑上是本行的后续内容:

```
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)
```

需要注意的是，还是需要在字符串中写入 `n` ——结尾的反斜杠会被忽略。前例会打印为如下形式:

```
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.
```

另外，字符串可以标识在一对儿三引号中： `"""` 或 `'''` 。三引号中，不需要行属转义，它们已经包含在字符串中。

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

得到如下输出:

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

如果我们生成一个“原始”字符串， `\n` 序列不会被转义，而且行尾的反斜杠，源码中的换行符，都成为字符串中的一部分数据，因此下例:

```
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:

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

字符串可以由 `+` 操作符连接（粘到一起），可以由 `*` 重复:

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

相邻的两个字符串文本自动连接在一起，前面那行代码也可以写为 word ='Help' 'A'

;它只用于两个字符串文本，不能用于字符串表达式:

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

字符串也可以被截取（检索）。类似于 C ，字符串的第一个字符索引为 0 。没有独立的字符类型，字符就是长度为 1 的字符串。类似 Icon ，可以用 切片标注 法截取字符串：由两个索引分割的复本。

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

索引切片可以有默认值，切片时，忽略第一个索引的话，默认为 0，忽略第二个索引，默认为字符串的长度。

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

不同于 C 字符串，Python 字符串不可变。向字符串文本的某一个索引赋值会引发错误:

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

不过，组合文本内容生成一个新文本简单而高效:

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

切片操作有个有用的不变性： s[:i] + s[i:] 等于 s。

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

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

Python 能够优雅的处理那些没有意义的切片索引：一个过大的索引值（即下标值大于字符串实际长度）将被字符串实际长度所代替，当上边界比下边界大时（即切片左值大于右值）就返回空字符串。

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

索引也可以是负数，这将导致从右边开始计算。 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]    # Everything except the last two characters
'Hel'
```

请注意 -0 实际上就是 0 ，所以它不会导致从右边开始计算！

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

负索引切片越界会被截断，不要尝试将它用于单元素（非切片）检索:

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

有个办法可以很容易的记住切片的工作方式：切片时的索引是在两个字符 之间 。左边第一个字符的索引为 0,，而长度为 n 的字符串其最后一个字符的右界索引为 n 。例如:

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

文本中的第一行数字给出字符串中的索引点 0...5 。第二行给出相应的负索引。切片是从 i 到 j 两个数值标示的边界之间的所有字符。
对于非负索引，如果上下都在边界内，切片长度与索引不同。例如， `word[1:3]` 是 2 。
内置函数 **len()** 返回字符串长度:

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

3.1.3. 关于 Unicode

从 Python 3.0 开始所有的字符串都支持 Unicode（参考 http://www.unicode.org ）。
Unicode 的先进之处在于为每一种现代或古代使用的文字系统中出现的每一个字符都提供了统一的序列号。之前，文字系统中的字符只能有 256 种可能的顺序。通过代码页分界映射。文本绑定到映射文字系统的代码页。这在软件国际化的时候尤其麻烦 （通常写作 i18n —— 'i' + 18 个字符 + 'n' ）。Unicode 解决了为所有的文字系统设置一个独立代码页的难题。
如果想在字符串中包含特殊字符，你可以使用 Python 的 Unicode_Escape 编码方式。 下面的列子展示了如何这样做:

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

转码序列 \u0020 表示在指定位置插入编码为 0x0020 的 Unicode 字符（空格）。
其他字符就像 Unicode 编码一样被直接解释为对应的编码值。 如果你有在许多西方国家使用的标准 Latin-1 编码的字符串，你会发现编码小于 256 的 Unicode 字符和在 Latin-1 编码中的一样。
除了这些标准编码，Python 还提供了一整套基于其他已知编码创建 Unicode 字符串的方法。字符串对象提供了一个 **encode()** 方法用以将字符串转换成特定编码的字节序列，它接收一个小写的编码名称作为参数。:

```
>>> "Äpfel".encode('utf-8')
b'\xc3\x84pfel'
```

3.1.4. 列表

Python 有几个 复合 数据类型，用于分线其它的值。最通用的是 list (列表) ，它可以写作中括号之间的一列逗号分隔的值。列表的元素不必是同一类型。

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

就像字符串索引，列表从 0 开始检索。列表可以被切片和连接:

```
>>> 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] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

所有的切片操作都会返回新的列表，包含求得的元素。这意味着以下的切片操作返回列表 a 的一个浅拷贝的副本:

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

```
['spam', 'eggs', 100, 1234]
```

不像 不可变的 字符串，列表允许修改元素

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

也可以对切片赋值，此操作可以改变列表的尺寸，或清空它:

```
>>> # 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]
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
[]
```

内置函数 **len()** 同样适用于列表:

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

允许嵌套列表（创建一个包含其它列表的列表），例如:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
```

你可以在列表末尾添加内容:

```
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
```

```
>>> q
[2, 3, 'xtra']
```

注意最后一个例子中，`p[1]` 和 `q` 实际上指向同一个对象！ 我们会在后面的 object semantics 中继续讨论。

3.2. 编程的第一步

当然，我们可以使用 Python 完成比二加二更复杂的任务。例如，我们可以写一个生成 菲波那契 子序列的程序，如下所示:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

这个例子介绍了几个新功能。

- 第一行包括了一个 多重赋值 ：变量 `a` 和 `b` 同时获得了新的值 0 和 1 最后一行又使用了一次。在这个演示中，变量赋值前，右边首先完成计算。右边的表达式从左到右计算。
- 条件（这里是 `b < 10`）为 true 时，`while` 循环执行。在 Python 中，类似于 C ，任何非零整数都是 true；0 是 false 条件也可以是字符串或列表，实际上可以是任何序列；所有长度不为零的是 true ，空序列是 false。示例中的测试是一个简单的比较。标准比较操作符与 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 是 Python 组织語句的方法。 Python (还) 不提供集成的行编辑功能，所以你要为每一个缩进行输入 TAB 或空格。实践中建议你找个文本编辑来录入复杂的 Python 程序，大多数文本编辑器提供自动缩进。交互式录入复合语句时，必须在最后输入一个空行来标识结束（因为解释器没办法猜测你输入的哪一行是最后一行），需要 注意的是同一个语句块中的语句块必须缩进同样数量的空白。
- 关键字 `print` 语句输出给定表达式的值。它控制多个表达式和字符串输出为你想要字符串（就像我们在前面计算器的例子中那样）。字符串打印时不用引号包围，每两个子项之间插入空间，所以你可以把格式弄得很漂亮，像这样

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

用一个逗号结尾就可以禁止输出换行:

```
>>> a, b = 0, 1
```

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

4. 深入 Python 流程控制

除了前面介绍的 **while** 语句，Python 还从其它语言借鉴了一些流程控制功能，并有所改变。

4.1. if 语句

也许最有名的是 **if** 语句。例如:

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

可能会有零到多个 **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 C 或 Pascal 中的略有不同。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。使用切割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. The range() 函数

如果你需要一个数值序列，内置函数 **range()** 会很方便，它生成一个等差级数链表:

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

``range(10)`` 生成了一个包含 10 个值的链表，它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 through 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
```

不过，这种场合可以方便的使用 **enumerate()** ， 请参见 循环技巧 。

A strange thing happens if you just print a range:

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

在不同方面 **range()** 函数返回的对象表现为它是一个列表，但事实上它并不是。 当你迭代它时，它是一个能够像期望的序列返回连续项的对象；但为了节省空间，它并不真正构造列表。

我们称此类对象是 可迭代的 ，即适合作为那些期望从某些东西中获得连续项直到结束的函数或

结构的一个目标（参数）。 我们已经见过的 **for** 语句就是这样一个 迭代器 。 **list()** 函数是另外一个（ 迭代器），它从可迭代（对象）中创建列表:

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

稍后我们会看到更多返回可迭代（对象）和以可迭代（对象）作为参数的函数。

4.4. break 和 continue 语句，以及循环中的 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。
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
```

(Yes, 这是正确的代码。看仔细: `else` 语句是属于 **for** 循环之中, **不是 if** 语句.)
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:

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

4.5. pass 语句

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

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

这通常用于创建最小结构的类:

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

另一方面， **pass** 可以在创建新代码时用来做函数或控制体的占位符。可以让你在更抽象的级别上思考。 **pass** 可以默默的被忽视

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

4.6. 定义函数

我们可以创建一个用以生成指定边界的斐波那契数列的函数:

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

关键字 **def** 引入了一个函数 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。

函数体的第一行语句可以是可选的字符串文本，这个字符串是函数的文档字符串，或者称为 docstring 。(更多关于 docstrings 的信息请参考 Documentation Strings 文档字符串 。) 有些工具通过 docstrings 自动生成在线的或可打印的文档，或者让用户通过代码交互浏览；在你的代码中包含 docstrings 是一个好的实践，让它成为习惯吧。

函数 调用 会为函数局部变量生成一个新的符号表。 确切的说，所有函数中的变量赋值都是将值存储在局部符号表。 变量引用首先在局部符号表中查找，然后是包含函数的局部符号表，然后是全局符号表，最后是内置名字表。 因此，全局变量不能在函数中直接赋值（除非用 **global** 语句命名），尽管他们可以被引用。

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）。 [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。

一个函数定义会在当前符号表内引入函数名。 函数名指代的值（即函数体）有一个被 Python 解释器认定为 用户自定义函数 的类型。 这个值可以赋予其他的名字（即变量名），然后它也可以

被当做函数使用。 这可以作为通用的重命名机制:

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

如果你使用过其他语言，你可能会反对说： fib 不是一个函数，而是一个方法，因为它并不返回任何值。 事实上，没有 **return** 语句的函数确实会返回一个值，虽然是一个相当令人厌烦的值（指 None ）。 这个值被称为 None （这是一个内建名称）。 如果 None 值是唯一被书写的值，那么在写的时候通常会被解释器忽略（即不输出任何内容）。 如果你确实想看到这个值的输出内容，请使用 **print()** 函数:

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

定义一个返回斐波那契数列数字列表的函数，而不是打印它，是很简单的:

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

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， class 的定义方法详见 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 Python 函数定义

在 Python 中，你也可以定义包含若干参数的函数。 这里有三种可用的形式，也可以混合使用。

4.7.1. 默认参数值

最常用的一种形式是为一个或多个参数指定默认值。 这会创建一个可以使用比定义时允许的参

数更少的参数调用的函数，例如:

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

这个函数可以通过几种不同的方式调用:

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

这个例子还介绍了 **in** 关键字。它测定序列中是否包含某个确定的值。
默认值在函数 定义 作用域被解析，如下所示

```
i = 5

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

i = 6
f()
```

将会输出 `5`.
重要警告: 默认值只被赋值一次。这使得当默认值是可变对象时会有所不同，比如列表、字典或者大多数类的实例。例如，下面的函数在后续调用过程中会累积（前面）传给它的参数:

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

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

这将输出

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

如果你不想让默认值在后续调用中累积，你可以像下面一样定义函数:

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

```
    return L
```

4.7.2. 关键字参数

函数可以通过 关键字参数 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, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

接受一个必选参数 (voltage) 以及三个可选参数 (state, action, and type). 可以用以下的任一方法调用:

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

不过以下几种调用是无效的:

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

通常，参数列表必须（先书写）位置参数然后才是关键字参数，这里关键字必须来自于形参名字。形参是否有一个默认值并不重要。 任何参数都不能被多次赋值——在同一个调用中，与位置参数相同的形参名字不能用作关键字。 这里有一个违反此限制而出错的例子:

```
>>> 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（参见 typesmapping ） ，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 （下一小节詳細介绍） 的形式参数，它接收一个元组（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 = sorted(keywords.keys())
    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.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           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
```

注意在打印 关系字 参数字典的内容前先调用 sort() 方法。否则的话，打印参数时的顺序是未定义的。

4.7.3. 可变参数列表

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元组（参见 元组和序列）。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 write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

通常，这些 可变 参数是参数列表中的最后一个， 因为它们将把所有的剩余输入参数传递给函数。任何出现在*args 后的参数是关键字参数，这意味着，他们只能被用作关键字，而不是位置参数。:

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

4.7.4. 参数列表的分拆

另有一种相反的情况: 当你要传递的参数已经是一个列表，但要调用的函数却接受分开一个个的参数值. 这时候你要把已有的列表拆开来. 例如内建函数 range() 需要要独立的 start , stop 参数. 你可以在调用函数时加一个 * 操作符来自动把参数列表拆开:

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

```
[3, 4, 5]
```

以同样的方式，可以使用 ** 操作符分拆关键字参数为字典:

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

4.7.5. Lambda 形式

出于实际需要，有几种通常在函数式编程语言例如 Lisp 中出现的功能加入到了 Python 。通过 **lambda** 关键字，可以创建短小的匿名函数。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 形式可以从外部作用域引用变量:

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

4.7.6. 文档字符串

这里介绍的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。留白的长度应当等于扩展制表符的宽度（通常是 8 个空格）。
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.
```

4.8. 插曲：编码风格

此时你已经可以写一此更长更复杂的 Python 程序，是时候讨论一下 编码风格 了。大多数语言可以写（或者更明白的说， 格式化 ）作几种不同的风格。有些比其它的更好读。让你的代码对别人更易读是个好想法，养成良好的编码风格对此很有帮助。
对于 Python， **PEP 8** 引入了大多数项目遵循的风格指导。它给出了一个高度可读，视觉友好的编码风格。每个 Python 开发者都应该读一下，大多数要点都会对你有帮助：

- 使用 4 空格缩进，而非 TAB。

  在小缩进（可以嵌套更深）和大缩进（更易读）之间，4 空格是一个很好的折中。TAB 引发了一些混乱，最好弃用。

- 折行以确保其不会超过 79 个字符。

  这有助于小显示器用户阅读，也可以让大显示器能并排显示几个代码文件。

- 使用空行分隔函数和类，以及函数中的大块代码。
- 可能的话，注释独占一行
- 使用文档字符串
- 把空格放到操作符两边，以及逗号后面，但是括号里侧不加空格：`a = f(1, 2) + g(3, 4)`。
- 统一函数和类命名。

  推荐类名用 `驼峰命名`， 函数和方法名用 `小写_和_下划线`。总是用 `self` 作为方法的第一个参数（关于类和方法的知识详见 初识类 ）。

- 不要使用花哨的编码，如果你的代码的目的是要在国际化 环境。 Python 的默认情况下，UTF-8，甚至普通的 ASCII 总是工作的最好。
- 同样，也不要使用非 ASCII 字符的标识符，除非是不同语种的会阅读或者维护代码。

Footnotes

[1] 实际上， 引用对象 调用描述的更为准确。如果传入一个可变对像，调用者会看到调用操作带来的任何变化（如子项插入到列表中）。

5. 数据结构

本章详细讨论了你已经学过的一些知识，同样也添加了一些新内容。

5.1. 深入列表

Python 的列表数据类型包含更多的方法。 这里是所有的列表对象方法：

list.**append**(*x*)

把一个元素添加到链表的结尾，相当于 `a[len(a):] = [x]` 。

list.**extend**(*L*)

将一个给定列表中的所有元素都添加到另一个列表中，相当于 `a[len(a):] = L` 。

list.**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)` 。

list.**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。

list.**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。）

list.**index**(*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。

list.**count**(*x*)

返回 *x* 在链表中出现的次数。

list.**sort**()

对链表中的元素就地进行排序。

list.**reverse**()

就地倒排链表中的元素。

下面这个示例演示了链表的大部分方法 :::

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

```
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

也许大家会发现像 `insert`, `remove` 或者 `sort` 这些修改列表的方法没有打印返回值–它们返回 `None` 。 [1]_ 在 python 中对所有可变的数据类型这是统一的设计原则。

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，堆栈作为特定的数据结构，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，队列作为特定的数据结构，最先进入的元素最先释放（先进先出）。不过，列表这样用效率不高。相对来说从列表末尾添加和弹出很快；在头部插入和弹出很慢（因为，为了一个元素，要移动整个列表中的所有元素）。

要实现队列，使用 **collections.deque** ，它为在首尾两端快速插入和删除而设计。例如:

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

5.1.3. 列表推导式

列表推导式为从序列中创建列表提供了一个简单的方法。 普通的应用程式通过将一些操作应用于序列的每个成员并通过返回的元素创建列表，或者通过满足特定条件的元素创建子序列。

例如，假设我们创建一个 squares 列表，可以像下面方式:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

我们同样能够达到目的采用下面的方式:

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

这也相当于 `squares = map(lambda x: x**2, range(10))`，但是上面的方式显得简洁以及具有可读性.

列表推导式由包含一个表达式的括号组成，表达式后面跟随一个 for 子句，之后可以有零或多个 for 或 if 子句。 结果是一个列表，由表达式依据其后面的 for 和 if 子句上下文计算而来的结果构成。

例如，如下的列表推导式结合两个列表的元素，如果元素之间不相等的话

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

等同于:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
```

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

值得注意的是在上面两个方法中的 **for** 和 **if** 语句的顺序。

如果想要得到一个元组，必须要加上括号。

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
File "<stdin>", line 1, in ?
[x, x**2 for x in range(6)]
^

SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

列表推导式可使用复杂的表达式和嵌套函数:

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

5.1.4. 嵌套的列表推导式

列表推导式可以嵌套。

考虑以下的 3x4 矩阵， 一个列表中包含三个长度为 4 的列表:

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

现在，如果你想交换行和列，可以用嵌套的列表推导式:

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

像前面看到的，嵌套的列表推导式是对 for 后面的内容进行求值，所以上例就等价于:

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

反过来说，如下也是一样的:

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

在实际中，你应该更喜欢使用内置函数组成复杂流程语句。 对此种情况 zip() 函数将会做的更好:

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

更多关于本行中使用的星号（*）的说明，参考 *参数列表的分拆* 。

5.2. del 语句

有个方法可以从列表中按给定的索引而不是值来删除一个子项： **del** 语句。它不同于有返回值的 pop() 方法。语句 **del** 还可以从列表中删除切片或清空整个列表（我们以前介绍过一个方法是将空列表赋值给列表的切片）。例如:

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

del 也可以删除整个变量:

```
>>> del a
```

此后再引用命名 `a` 会引发错误(直到另一个值赋给它为止)。我们在后面的内容中可以看到 **del** 的其它用法。

5.3. 元组和序列

我们知道链表和字符串有很多通用的属性，例如索引和切割操作。它们是 序列 类型（参见 *typesseq*)中的两种。因为 Python 是一个在不停进化的语言，也可能会加入其它的序列类型，这里介绍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))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

```
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可以有或没有括号，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）。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割来模拟）。还可以创建包含可变对象的元组，例如链表。

虽然元组和列表很类似，它们经常被用来在不同的情况和不同的用途。元组有很多用途。例如 (x, y) 坐标对，数据库中的员工记录等等。元组就像字符串，不可改变。

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不够明确）。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* （tuple 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
```

这个调用等号右边可以是任何线性序列，称之为 *序列拆封* 非常恰当。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。

5.4. 集合

Python 还包含了一个数据类型—— *set* （集合） 。集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。集合对象还支持 union（联合），intersection（交），difference（差）和 sysmmetric difference（对称差集）等数学运算。

大括号或 `set()` 函数可以用来创建集合。注意：想要创建空集合，你必须使用 `set()` 而不是 `{}` 。后者用于创建空字典，我们在下一节中介绍的一种数据结构。

以下是一个简单的演示:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
```

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

```
>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

类似 *for lists* ，这里有一种集合推导式语法:

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

5.5. 字典

另一个非常有用的 Python 内建数据类型是 *字典* （参见 *typesmapping* ）。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，因为链表可以用索引、切割或者 append() 和 extend() 等方法改变。

理解字典的最佳方式是把它看做无序的键： *值对* （key:value pairs）集合，键必须是互不相同的（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键：值对，这也是字典输出的方式。

字典的主要操作是依据键来存储和析取值。也可以用 `del` 来删除键：值对（key:value）。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。试图从一个不存在的键中取值会导致错误。

对一个字典执行 `list(d.keys())` 将返回一个字典中所有关键字组成的无序列表（如果你想要排序，只需使用 `sorted(d.keys())` ）。[2]_ 使用 `in` 关键字（指 Python 语法）可以检查字典中是否存在某个关键字（指字典）。

这里是使用字典的一个小示例:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

`dict()` 构造函数可以直接从 key-value 对中创建字典:

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

此外，字典推导式可以从任意的键值表达式中创建字典:

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

如果关键字都是简单的字符串，有时通过关键字参数指定 key-value 对更为方便:

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

5.6. 循环技巧

在字典中循环时，关键字和对应的值可以使用 `iteritems()` 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() 整体打包。:

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

需要逆向循环序列的话，先正向定位序列，然后调用 reversed() 函数。

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

要按排序后的顺序循环序列的话，使用 sorted() 函数，它不改动原序列，而是生成一个新的已排序的序列。

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

5.7. 深入条件控制

while 和 if 语句中使用的条件不仅可以使用比较，而且可以包含任意的操作。

比较操作符 in 和 not in 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 并且 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 优先级最低，所以 A and not B or C 等于 (A and(notB)) or C 。当然，括号也可以用于比较表达式。

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_null
'Trondheim'
```

需要注意的是 Python 与 C 不同，在表达式内部不能赋值。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。

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

需要注意的是如果通过 < 或者 > 比较的对象只要具有合适的比较方法就是合法的。 比如，混合数值类型是通过它们的数值就行比较的，所以 0 是等于 0.0。 否则解释器将会触发一个 TypeError 异常，而不是提供一个随意的结果。

6. 模块

如果你退出 Python 解释器并重新进入，你做的任何定义（变量和方法）都会丢失。 因此，如果你想要编写一些更大的程序，为准备解释器输入使用一个文本编辑器会更好，并以那个文件替代作为输入执行。 这就是传说中的 *脚本* 。 随着你的程序变得越来越长，你可能想要将它分割成几个更易于维护的文件。 你也可能想在不同的程序中使用顺手的函数，而不是把代码在它们之间中拷来拷去。

为了满足这些需要，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 *模块* ；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, end=' ')
        a, b = b, a+b
    print()

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

现在进入 Python 解释器并使用以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'
```

如果打算频繁使用一个函数，你可以将它赋予一个本地变量:

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

6.1. 深入模块

除了包含函数定义外，模块也可以包含可执行语句。这些语句一般用来初始化模块。他们仅在 *第一次* 被导入的地方执行一次。 [1]

每个模块都有自己私有的符号表，被模块内所有的函数定义作为全局符号表使用。 因此，模块的作者可以在模块内部使用全局变量，而无需担心它与某个用户的全局变量意外冲突。 从另一个方面讲，如果你确切的知道自己在做什么，你可以使用引用模块函数的表示法访问模块的全局变量， `modname.itemname` 。

模块可以导入其他的模块。 一个（好的）习惯是将所有的 **import** 语句放在模块的开始（或者是脚本），这并非强制。 被导入的模块名会放入当前模块的全局符号表中。

import 语句的一个变体直接从被导入的模块中导入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（如上所示， `fibo` 没有定义）。

甚至有种方式可以导入模块中的所有定义:

```
>>> 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。

需要注意的是在实践中往往不鼓励从一个模块或包中使用 `*` 导入所有，因为这样会让代码变得很难读。不过，在交互式会话中这样用很方便省力。

Note

出于性能考虑，每个模块在每个解释器会话中只导入一遍。因此，如果你修改了你的模块，需要重启解释器——或者，如果你就是想交互式的测试这么一个模块，可以用 `reload()` 重新加载，例如 `reload(modulename)` 。

6.1.1. 作为脚本来执行模块

当你使用以下方式运行 Python 模块时，模块中的代码便会被执行:

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

模块中的代码会被执行，就像导入它一样，不过此时 `__name__` 被设置为 `"__main__"` 。这相当于，如果你在模块后加入如下代码:

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

就可以让此文件像作为模块导入时一样作为脚本执行。此代码只有在模块作为 “main” 文件执行时才被调用:

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

如果模块被导入，不会执行这段代码:

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

这通常用来为模块提供一个便于测试的用户接口（将模块作为脚本执行测试需求）。

6.1.2. 模块的搜索路径

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。如果没有找到的话，接着会到 `sys.path` 变量中给出的目录列表中查找。 `sys.path` 变量的初始值来自如下：

- 输入脚本的目录（当前目录。
- 环境变量 **PYTHONPATH** 表示的目录列表中搜索 (这和 shell 变量 envvar:*PATH* 具有一样的语法，即一系列目录名的列表)。
- Python 默认安装路径中搜索。

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）， PYTHONPATH 和安装目录。这样就允许 Python 程序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。这通常会引发错误。请参见 *标准模块* 以了解更多的信息。

6.1.3. “编译的” 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的重要方法，如果在 spam.py 所在的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，就会尝试生成对应版本的 spam.pyc 。如果有任何原因导致写入不成功，生成的 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。现在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* 的字节码（ *bytecode* ）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。现在的优化器，只是从字节码中删除了 __doc__ 符串，生成更为紧凑的 .pyo 文件。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，不会将为该脚本创建的二进制代码写入 .pyc 或 .pyo 文件。当然，把脚本的主要代码移进一个模块里，然后用一个小的启动脚本导入这个模块，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 .pyo 文件。
- 对于同一个模块（这里指例程 spam.py ——译者），可以只有 spam.pyc 文件（或者 spam.pyc ，在使用 *-O* 参数时）而没有 spam.py 文件。这样可以打包发布比较难于逆向工程的 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，**winreg** 模块只提供在 Windows 系统上才有。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。

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，如果没有设定 PYTHONPATH，就由内置的默认值初始化。你可以用标准的字符串操作修改它:

```
>>> 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_info', '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() 函数返回当前定义的命名:

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

注意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。如果你想列出这些内容，它们在标准模块 __builtin__ 中定义:

```
>>> import builtins
>>> dir(builtins)
```

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

6.4. 包

包通常是使用用“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 `B` 的包中名为 `A` 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 NumPy 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
              ...
```

当导入这个包时，Python 通过 `sys.path` 搜索路径查找包含这个包的子目录。

为了让 Python 将目录当做内容包，目录中必须包含 `__init__.py` 文件。 这是为了避免一个含有烂俗名字的目录无意中隐藏了稍后在模块搜索路径中出现的有效模块，比如 string 。 最简单的情况下，只需要一个空的`__init__.py` 文件即可。 当然它也可以执行包的初始化代码，或者定义稍后介绍的 `__all__` 变量。

用户可以每次只导入包里的特定模块，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 * 导入包

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。这可能会花掉委有长时间，并且出现期待之外的边界效应，导出了希望只能显式导入的包。

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 package import *` 时，如果包中的 `__init__.py` 代码定义了一个名为 `__all__` 的列表，就会按照列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列表。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。无论包中定义多少命名，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）。

尽管某些模块设计为使用 `import *` 时它只导出符全某种模式的命名，仍然不建议在生产代码中使用这种写法。

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，否则这是推荐的写法。

6.4.2. 包内引用

如果包中使用了子包结构（就像示例中的 sound 包），可以按绝对位置从相邻的包中引入子模块。例如， 如果 sound.filters.vocoder 包需要使用 sound.effects 包中的 echo 模块， 它可以 `from Sound.Effects importecho` 。

你可以用这样的形式 `from module import name` 来写显式的相对位置导入。那些显式相对导入用点号标明关联导入当前和上级包。以 **surround** 模块为例，你可以这样用:

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

需要注意的是显式或隐式相对位置导入都基于当前模块的命名。因为主模块的名字总是 `"__main__"`，Python 应用程序的主模块应该总是用绝对导入。

6.4.3. 多重目录中的包

包支持一个更为特殊的特性， **__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。

Footnotes

[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。

7. 输入和输出

一个程序可以有几种输出方式：以人类可读的方式打印数据，或者写入一个文件供以后使用。 本章将讨论几种可能性。

7.1. 格式化输出

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 **write()** 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。）

通常，你想要对输出做更多的格式控制，而不是简单的打印使用空格分隔的值。 有两种方法可以格式化你的输出： 第一种方法是由你自己处理整个字符串，通过使用字符串切割和连接操作可以创建任何你想要的输出形式。string 类型包含一些将字符串填充到指定列宽度的有用操作，随后就会讨论这些。 第二种方法是使用 **str.format()** 方法。

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 **Template** 方法。

当然，还有一个问题，如何将值转化为字符串？很幸运，Python 有办法将任意值转为字符串：将它传入 **repr()**或 **str()** 函数。

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。

下面有些例子:

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

有两种方式可以写平方和立方表:

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

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
```

```
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(注意第一个例子， print 在每列之间加了一个空格，它总是在参数间加入空格。)

以上是一个 str.rjust() 方法的演示，它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。类似的方法还有 str.ljust() 和 str.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，可以使用切割操作，例如： x.ljust(n)[:n]。）

还有另一个方法， str.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:

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

方法 str.format() 的基本用法如下:

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

大括号和其中的字符会被替换成传入 str.format() 的参数。大括号中的数值指明使用传入 str.format() 方法的对象中的哪一个。:

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

如果在 str.format() 调用时使用关键字参数，可以通过参数名来引用值。:

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

定位和关键字参数可以组合使用

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

'!a' (应用 ascii()), '!s' （应用 str() ） 和 '!r' （应用 repr() ） 可以在格式化之前转换值:

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

字段名后允许可选的 ':' 和格式指令。这允许对值的格式化加以更深入的控制。下例将 Pi 转为三位精度。

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

在字段后的 ':' 后面加一个整数会限定该字段的最小宽度，这在美化表格时很有用。

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

如果你有个实在是很长的格式化字符串，不想分割它。如果你可以用命名来引用被格式化的变量而不是位置就好了。有个简单的方法，可以传入一个字典，用中括号访问它的键

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

也可以用 '**' 标志将这个字典以关键字参数的方式传入。

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

这种方式与新的内置函数 vars() 组合使用非常有效。该函数返回包含所有局部变量的字典。

要进一步了解字符串格式化方法 str.format() ， 参见 *formatstrings* 。

7.1.1. 旧式的字符串格式化

操作符 % 也可以用于字符串格式化。它以类似 sprintf()-style 的方式解析左参数，将右参数应用于此，得到格式化操作生成的字符串，例如:

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

因为 str.format() 还很新，大量 Python 代码还在使用 % 操作符。然而，因为旧式的格式化方法最终将从语言中去掉，应该尽量使用 str.format() 。

进一步的信息可以参见 *string-formatting* 一节。

7.2. 文件读写

函数 open() 返回文件对象，通常的用法需要两个参数： open(filename, mode)。

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

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； 模式 参数是可选的。如果没有指定，默认为 'r' 模式。

在 Windows 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 ASCII 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。在 Unix 上，加一个 'b' 模式也一样是无害的，所以你可以一切二进制文件处理中平台无关的使用它。

7.2.1. 文件对象方法

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， *size* 是可选的数值，指定字符串长度。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。反之，会尽可能按比较大的 *size* 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（`\n`），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混淆，如果如果 `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，从中返回多行文本。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']
```

一种替代的方法是通过遍历文件对象来读取文件行。 这是一种内存高效、快速，并且代码简介的方式:

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

虽然这种替代方法更简单，但并不具备细节控制能力。 因为这两种方法处理行缓存的方式不同，千万不能搞混。

`f.write(string)` 方法将 string 的内容写入文件，并返回写入字符的长度。

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

想要写入其他非字符串内容，首先要将它转换为字符串:

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

`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

件起始处开始，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* 可以忽略，其默认值为零，此时从文件头开始。

```
>>> f = open('/tmp/workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

在文本文件中（那些没有使用 `b` 模式选项打开的文件），只允许从文件头开始计算相对位置（使用 `seek(0, 2)`从文件尾计算时就会引发异常）。

当你使用完一个文件时，调用 `f.close()` 方法就可以关闭它并释放其占用的所有系统资源。 在调用 `f.close()`方法后，试图再次使用文件对象将会自动失败。:

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

用关键字 **with** 处理文件对象是个好习惯。它的先进之处在于文件用完后会自动关闭，就算发生异常也没关系。它是 **try-finally** 块的简写:

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

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 **int()** 这样的方法中，就可以将 '123' 这样的字符转为对应的数值 123 。不过，当你需要保存更为复杂的数据类型，例如列表、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 代码段！）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`，封装对象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 Python 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* 和 *exceptions* ）。

8.1. 语法错误

语法错误，也被称作解析错误，也许是你学习 Python 过程中最常见抱怨:

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

语法分析器指出错误行，并且在检测到错误的位置前面显示一个小“箭头”。错误是由箭头 *前面* 的标记引起的（或者至少是这么检测的）： 这个例子中，函数 `print()` 被发现存在错误，因为它前面少了一个冒号（`':'`）。 错误会输出文件名和行号，所以如果是从脚本输入的你就知道去哪里检查错误了。

8.2. 异常

即使一条语句或表达式在语法上是正确的，当试图执行它时也可能会引发错误。 运行期检测到的错误称为 *异常*，并且程序不会无条件的崩溃：很快，你将学到如何在 Python 程序中处理它们。然而，大多数异常都不会被程序处理，像这里展示的一样最终会产生一个错误信息:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: int 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: Can't convert 'int' object to str implicitly
```

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。

bltin-exceptions 列出了内置异常和它们的含义。

8.3. 异常处理

通过编程处理选择的异常是可行的。 看一下下面的例子：它会一直要求用户输入，直到输入一个合法的整数为止，但允许用户终端这个程序（使用 `Control-C` 或系统支持的任何方法）。 注意：用户产生的终端会引发一个 **KeyboardInterrupt** 异常。

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

try 语句按如下方式工作。

- 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。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称，以作为通配符使用。 你需要慎用此法，因为它会轻易隐藏一个实际的程序错误！ 可以使用这种方法打印一条错误信息，然后重新抛出异常（允许调用者处理这个异常):

```
import sys

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

try ... except 语句可以带有一个 *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 __str__() ，这样就可以直接访问过打印参数而不必引用 .args 。 这种做法不受鼓励。相反，更好的做法是给异常传递一个参数（如果要传递多个参数，可以传递一个元组），把它绑定到 message 属性。一旦异常发生，它会在抛出前绑定所有指定的属性。

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

对于那些未处理的异常，如果一个它们带有参数，那么就会被作为异常信息的最后部分（“详情”）打印出来。

异常处理器不仅仅处理那些在 try 子句中立刻发生的异常，也会处理那些 try 子句中调用的函数内部发生的异常。 例如:

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

8.4. 抛出异常

raise 语句允许程序员强制抛出一个指定的异常。例如:

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

要抛出的异常由 raise 的唯一参数标识。它必需是一个异常实例或异常类（继承自 Exception 的类）。

如果你需要明确一个异常是否抛出，但不想处理它， 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（Python 类的内容请参见 *类*）。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 as e:
...     print('My exception occurred, value:', e.value)
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

在这个例子中，Exception 默认的 __init__() 被覆盖。新的方式简单的创建 *value* 属性。这就替换了原来创建 *args* 属性的方式。

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 *类* 一章。

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!
KeyboardInterrupt
```

不管有没有发生异常， *finally* 子句在程序离开 try 后都一定会被执行。当 try 语句中发生了未被 except 捕获的异常（或者它发生在 except 或 else 子句中），在 finally 子句执行完后它会被重新抛出。 try 语句经由 break ， continue 或 return 语句退 出也一样会执行 finally 子句。以下是一个更复杂些的例子（在同 一个 try 语句中的 except 和 finally 子句的工作方式与 Python 2.5 一样）

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

如你所见， finally 子句在任何情况下都会执 行。 TypeError 在两个字符串相除的时候抛出，未被 except 子句捕获，因此在 finally 子句执行完毕后重新抛出。

在真实场景的应用程序中， finally 子句用于释放外部资源（文件 或网络连接之类的），无论它们的使用过程中是否出错。

8.7. 预定义清理行为

有些对象定义了标准的清理行为，无论对象操作是否成功，不再需要该对象的时 候就会起作用。以下示例尝试打开文件并把内容打印到屏幕上。：

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

这段代码的问题在于在代码执行完后没有立即关闭打开的文件。这在简单的脚本 里没什么，但是大型应用程序就会出问题。 with 语句使得文件之类的对象可以 确保总能及时准确地进行清理。：

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

语句执行后，文件 *f* 总会被关闭，即使是在处理文件中的数据时出错也一样。 其它对象是否提供了预定义的清理行为要查看它们的文档。

9. 类

Python 的类机制通过最小的新语法和语义在语言中实现了类。 它是 C++何 Modula-3 语言中类机制的混合。 就像模块一样，Python 的类并没有在用户和定义之间设立绝对的屏障，而是依赖于用户不去“强行闯入定义”的优雅。 另一方面，类的大多数重要特性都被完整的保留下来：类继承机制允许多重继承，派生类可以覆盖（override）基类中的任何方法或类，可以使用相同的方法名称调用基类的方法。 对象可以包含任意数量的私有数据。

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 （*virtual* ）的。用 Modula-3 的术语来讲，在成员方法中没有简便的方式引用对象的成员：方法函数在定义时需要以引用的对象做为第一个参数，调用时则会隐式引用对象。像在 Smalltalk 中一个，类也是对象。这就提供了导入和重命名语义。不像 C++ 和 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。

（在讨论类时，没有足够的得到共识的术语，我会偶尔从 Smalltalk 和 C++ 借用一些。我比较喜欢用 Modula-3 的用语，因为比起 C++， Python 的面向对象语法更像它，但是我想很少有读者听过这个。）

9.1. 术语相关

对象具有特性，并且多个名称（在多个作用于中）可以绑定在同一个对象上。 这在其它语言中被称为别名。 在对 Python 的第一印象中这通常会被忽略，并且当处理不可变基础类型（数字，字符串，元组）时可以被放心的忽略。 但是，在调用列表、字典这类可变对象，或者大多数程序外部类型（文件，窗体等）描述实体时，别名对 Python 代码的语义便具有（有意而为！）影响。 这通常有助于程序的优化，因为在某些方面别名表现的就像是指针。 例如，你可以轻易的传递一个对象，因为通过继承只是传递一个指针。 并且如果一个方法修改了一个作为参数传递的对象，调用者可以接收这一变化——这消除了两种不同的参数传递机制的需要，像 Pascal 语言。

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。

让我们从一些定义说起。

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！[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__ 。）

当调用函数时，就会为它创建一个局部命名空间，并且在函数返回或抛出一个并没有在函数内部处理的异常时被删除。 （实际上，用遗忘来形容到底发生了什么更为贴切。） 当然，每个递归调用都有自己的局部命名空间。

作用域 就是一个 Python 程序可以直接访问命名空间的正文区域。这里的 直接访问 意思是一个对名称的错误引用会尝试在命名空间内查找。

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。
- 首先搜索最内层的作用域，它包含局部命名任意函数包含的作用域，是内层嵌套作用域搜索起点，包含非局部，但是也非全局的命名
- 接下来的作用域包含当前模块的全局命名
- 最外层的作用域（最后搜索）是包含内置命名的命名空间。

如果一个命名声明为全局的，那么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。（试图写这样的变量只会在内部作用域创建一个 *新* 局部变量，外部标示命名的那个变量不会改变）。

通常，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。

重要的是作用域决定于源程序的意义：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——如果没有使用 global 语法——其赋值操作总是在最里层的作用域。赋值不会复制数据——只是将命名绑定到对象。删除也是如此： `del x` 只是从局部作用域的命名空间中删除命名 `x` 。事实上，所有引入新命名的操作都作用于局部作用域。特别是 import 语句和函数定将模块名或函数绑定于局部作用域。（可以使用 global 语句将变量引入到全局作用域。）

global 语句用以指明某个特定的变量为全局作用域，并重新绑定它。 nonlocal 语句用以指明某个特定的变量为封闭作用域，并重新绑定它。

9.2.1. 作用域和命名空间示例

以下是一个示例，演示了如何引用不同作用域和命名空间，以及 global 和 nonlocal 如何影响变量绑定:

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"

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

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

以上示例代码的输出为:

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

注意： *local* 赋值语句是无法改变 *scope_test* 的 *spam* 绑定。 nonlocal 赋值语句改变了 *scope_test* 的 *spam* 绑定，并且 global 赋值语句从模块级改变了 spam 绑定。

你也可以看到在 global 赋值语句之前对 spam 是没有预先绑定的。

9.3. 初识类

类引入了一些新语法：三种新的对象类型和一些新的语义。

9.3.1. 类定义语法

类定义最简单的形式如下:

```
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。

进入类定义部分后，会创建出一个新的命名空间，作为局部作用域——因此，所有的赋值成为这个新命名空间的局部变量。特别是函数定义在此绑定了新的命名。

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 simpleexample 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。

数据属性 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 而在堆栈中留下多余的东西:

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

另一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，在后面的介绍中，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` 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，方法通过右绑定方式调用:

```
x.f()
```

在 **MyClass** 示例中，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

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

数据属性可以被方法引用，也可以由一个对象的普通用户（客户）使用。 换句话说，类不能用来实现纯净的数据类型。 事实上，Python 中不可能强制隐藏数据——一切基于约定。 （如果需

要，使用 C 编写的 Python 实现可以完全隐藏实现细节并控制对象的访问。这可以用来通过 C 语言扩展 Python。）

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

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

一般，方法的第一个参数被命名为 self 。 这仅仅是一个约定：对 Python 而言，名称 self 绝对没有任何特殊含义。 （但是请注意：如果不遵循这个约定，对其他的 Python 程序员而言你的代码可读性就会变差，而且有些 类查看器 程序也可能是遵循此约定编写的。）

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

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

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

现在 f, g 和 h 都是类 C 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。

每个值都是一个对象，因此每个值都有一个 类(*class*) （也称为它的 类型(*type*) ），它存储为 object.__class__ 。

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。（要注意只有 `BaseClassName` 在同一全局作用域定义或导入时才能这样用。）

Python 有两个用于继承的函数：

- 函数 **isinstance()** 用于检查实例类型：`isinstance(obj, int)` 只有在 `obj.__class__` 是 **int** 或其它从 **int** 继承的类型
- 函数 **issubclass()** 用于检查类继承： `issubclass(bool, int)` 为 `True` ，因为 **bool** 是 int 的子类。但是， `issubclass(unicode, str)` 是 `False` ，因为 **unicode** 不是 **str** 的子类（它们只是共享一个通用祖先类 **basestring** ）。

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>
```

在大多数情况下，在最简单的情况下，你能想到的搜索属性从父类继承的深度优先，左到右，而不是搜索两次在同一个类层次结构中，其中有一个重叠。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。

实际上，super() 可以动态的改变解析顺序。这个方式可见于其它的一些多继承语言，类似 call-next-method，比单继承语言中的 super 更强大 。

动态调整顺序十分必要的，因为所有的多继承会有一到多个菱形关系（指有至少一个祖先类可以从子类经由多个继承路径到达）。例如，所有的 new-style 类继承自 object ，所以任意的多继承总是会有多于一条继承路径到达 object 。为了防止重复访问基类，通过动态的线性化算法，每个类都按从左到右的顺序特别指定了顺序，每个祖先类只调用一次，这是单调的（意味着一个类被继承时不会影响它祖先的次序）。总算可以通过这种方式使得设计一个可靠并且可扩展的多继承类成为可能。进一步的内容请参见 http://www.python.org/download/releases/2.3/mro/ 。

9.6. 私有变量

只能从对像内部访问的“私有”实例变量，在 Python 中不存在。然而，也有一个变通的访问用于大多数 Python 代码：以一个下划线开头的命名（例如 _spam ）会被处理为 API 的非公开部分（无论它是一个函数、方法或数据成员）。它会被视为一个实现细节，无需公开。

因为有一个正当的类私有成员用途（即避免子类里定义的命名与之冲突），Python 提供了对这种结构的有限支持，称为 *name mangling*`（命名编码） 。任何形如 ``__spam`` 的标识（前面至少两个下划线，后面至多一个），被替代为 _classname__spam ，去掉前导下划线的 classname 即当前的类名。此语法不关注标识的位置，只要求在类定义内。

名称重整是有助于子类重写方法，而不会打破组内的方法调用。 例如:

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

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

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

class MappingSubclass(Mapping):

    def update(self, keys, values):
```

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

需要注意的是编码规则设计为尽可能的避免冲突，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。

要注意的是代码传入 `exec`， `eval()` 或 `execfile()` 时不考虑所调用的类的类名，视其为当前类，这类似于 `global` 语句的效应，已经按字节编译的部分也有同样的限制。这也同样作用于 `getattr()`， `setattr()` 和 `delattr()`，像直接引用 `__dict__` 一样。

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，有效的（语义上的）异常抛出形式，使用 `raise` 语句:

```
raise Class

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 Class()
```

发生的异常其类型如果是 except 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(Exception):
    pass
class C(B):
    pass
class D(C):
    pass

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

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。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>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    next(it)
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]
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.10. 生成器

Generators 是创建迭代器的简单而强大的工具。它们写起来就像是正规的函数，需要返回数据的时候使用 **yield** 语句。每次 **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）。以下示例演示了生成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，它能作的每一件事生成器也能作到。因为自动创建了 **__iter__()** 和 **next()** 方法，生成器显得如此简洁。

另一个关键的功能在于两次执行之间，局部变量和执行状态都自动的保存下来。这使函数很容易写，而且比使用 `self.index` 和 `self.data` 之类的方式更清晰。

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规函数成为创建迭代器的最简单方法。

9.11. 生成器表达式

有时简单的生成器可以用简洁的方式调用，就像不带中括号的链表推导式。这些表达式是为函数调用生成器而设计的。生成器表达式比完整的生成器定义更简洁，但是没有那么多变，而且通常比等价的链表推导式更容易记。

例如:

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

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

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

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

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

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

Footnotes

[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. Python 标准库概览

10.1. 操作系统接口

os 模块提供了很多与操作系统交互的函数:

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python33'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

应该用 `import os` 风格而非 `from os import *` 。这样可以保证随操作系统不同而有所变化的 os.open() 不会覆盖内置函数 open()

在使用一些像 os 这样的大型模块时内置的 dir() 和 help() 函数非常有用:

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

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

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

10.2. 文件通配符

glob 模块提供了一个函数用于从目录通配符搜索中生成文件列表:

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

10.3. 命令行参数

通用工具脚本经常调用命令行参数。这些命令行参数以链表形式存储于 sys 模块的 *argv* 变量。例如在命令行中执行 `python demo.py one two three` 后可以得到以下输出结果

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

getopt 模块使用 Unix getopt() 函数处理 *sys.argv* 。更多的复杂命令行处理由 argparse 模块提供。

10.4. 错误输出重定向和程序终止

sys 还有 *stdin* ， *stdout* 和 *stderr* 属性，即使在 *stdout* 被重定向时，后者也可以用于显示警告和错误信息

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

大多脚本的定向终止都使用 `sys.exit()` 。

10.5. 字符串正则匹配

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

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

只需简单的操作时，字符串方法最好用，因为它们易读，又容易调试

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

10.6. 数学

math 模块为浮点运算提供了对底层 C 函数库的访问

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

random 提供了生成随机数的工具

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

SciPy <http://scipy.org> 项目提供了许多数值计算的模块。

10.7. 互联网访问

有几个模块用于访问互联网以及处理网络通信协议。其中最简单的两个是用于处理从 urls 接收的数据的 urllib.request 以及用于发送电子邮件的 smtplib

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

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

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
```

```
...
...     Beware the Ides of March.
...     """)
>>> server.quit()
```

(注意第二个例子需要在 localhost 运行一个邮件服务器。)

10.8. 日期和时间

datetime 模块为日期和时间处理同时提供了简单和复杂的方法。支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。该模块还支持时区处理。:

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y.  %d %b %Y is a %A on the  %d day of %B.")
'12-02-03.  02 Dec 2003 is a Tuesday on the 02 day of December.'

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

10.9. 数据压缩

以下模块直接支持通用的数据打包和压缩格式： zlib, gzip, bz2, zipfile 以及 tarfile。:

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

10.10. 性能度量

有些用户对了解解决同一问题的不同方法之间的性能差异很感兴趣。Python 提供了一个度量工具，为这些问题提供了直接答案。

例如，使用元组封装和拆封来交换元素看起来要比使用传统的方法要诱人的多。 **timeit** 证明了后者更快一些

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

相对于 **timeit** 的细粒度，**profile** 和 **pstats** 模块提供了针对更大代码块的时间度量工具。

10.11. 质量控制

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试。

doctest 模块提供了一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。测试构造如同简单的将它的输出结果剪切并粘贴到文档字符串中。通过用户提供的例子，它发展了文档，允许 doctest 模块确认代码的结果是否与文档一致

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

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

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

unittest 模块不像 **doctest** 模块那么容易使用，不过它可以在一个独立的文件里提供一个更全面的测试集

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

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

```
        self.assertRaises(TypeError, average, 20, 30, 70)

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

10.12. “瑞士军刀”

Python 展现了“瑞士军刀”的哲学。 这可以通过它更大的包的高级和健壮的功能来得到最好的展现。 列如:

- **xmlrpc.client** 和 **xmlrpc.server** 模块让远程过程调用变得轻而易举。 尽管模块有这样的名字，用户无需拥有 XML 的知识或处理 XML。
- **email** 包是一个管理邮件信息的库，包括 MIME 和其它基于 RFC 2822 的信息文档。 不同于实际发送和接收信息的 **smtplib** 和 **poplib** 模块， email 包包含一个构造或解析复杂消息结构（包括附件）及实现互联网编码和头协议的完整工具集。
- **xml.dom** 和 **xml.sax** 包为流行的信息交换格式提供了强大的支持。同样， **csv** 模块支持在通用数据库格式中直接读写。综合起来，这些模块和包大大简化了 Python 应用程序和其它工具之间的数据交换。
- 国际化由 **gettext** ， **locale** 和 **codecs** 包支持。

11. 标准库浏览 - Part II

第二部分包含了支持专业编程工作所需的更高级的模块，这些模块很少出现在小脚本中。

11.1. 输出格式

reprlib 模块为大型的或深度嵌套的容器缩写显示提供了 **repr()** 函数的一个定制版本:

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

pprint 模块给老手提供了一种解释器可读的方式深入控制内置和用户自定义对象的打印。当输出超过一行的时候，“美化打印（pretty printer）”添加断行和标识符，使得数据结构显示的更清晰:

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

textwrap 模块格式化文本段落以适应设定的屏宽:

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

locale 模块按访问预定好的国家信息数据库。locale 的格式化函数属性集提供了一个直接方式以分组标示格式化数字:

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

11.2. 模板

string 提供了一个灵活多变的模版类 Template，使用它最终用户可以用简单的进行编辑。这使用户可以在不进行改变的情况下定制他们的应用程序。

格式使用 $ 为开头的 Python 合法标识（数字、字母和下划线）作为占位符。占位符外面的大括号使它可以和其它的字符不加空格混在一起。 $$ 创建一个单独的 $:

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

当一个占位符在字典或关键字参数中没有被提供时， substitute() 方法就会抛出一个 KeyError 异常。 对于邮件合并风格的应用程序，用户提供的数据可能并不完整，这时使用 safe_substitute() 方法可能更适合 — 如果数据不完整，它就不会改变占位符:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
```

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

模板子类可以指定一个自定义分隔符。 例如，图像查看器的批量重命名工具可能选择使用百分号作为占位符，像当前日期，图片序列号或文件格式:

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

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

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

模板的另一个应用是把多样的输出格式细节从程序逻辑中分类出来。这便使得 XML 文件，纯文本报表和 HTML WEB 报表定制模板成为可能。

11.3. 使用二进制数据记录布局

struct 模块为使用变长的二进制记录格式提供了 **pack()** 和 **unpack()** 函数。 下面的示例演示了在不使用 **zipfile** 模块的情况下如何迭代一个 ZIP 文件的头信息。 压缩码 `"H"` 和 `"I"` 分别表示 2 和 4 字节无符号数字， `"<"` 表明它们都是标准大小并且按照 little-endian 字节排序。:

```
import struct

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

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

```
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

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

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

11.4. 多线程

线程是一个分离无顺序依赖关系任务的技术。在某些任务运行于后台的时候应用程序会变得迟缓，线程可以提升其速度。一个有关的用途是在 I/O 的同时其它线程可以并行计算。

下面的代码显示了高级模块 threading 如何在主程序运行的同时运行任务:

```
import threading, zipfile

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

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

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

多线程应用程序的主要挑战是协调线程，诸如线程间共享数据或其它资源。 为了达到那个目的，线程模块提供了许多同步化的原生支持，包括：锁，事件，条件变量和信号灯。

尽管这些工具很强大，微小的设计错误也可能造成难以挽回的故障。 因此，任务协调的首选方法是把对一个资源的所有访问集中在一个单独的线程中，然后使用 queue 模块用那个线程服务其他线程的请求。 为内部线程通信和协调而使用 Queue 对象的应用程序更易于设计，更可读，并且更可靠。

11.5. 日志

logging 模块提供了完整和灵活的日志系统。它最简单的用法是记录信息并发送到一个文件或 sys.stderr:

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

输出如下:

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

默认情况下捕获信息和调试消息并将输出发送到标准错误流。其它可选的路由信息方式通过 email，数据报文，socket 或者 HTTP Server。基于消息属性，新的过滤器可以选择不同的路由： DEBUG, INFO, WARNING, ERROR, 和 CRITICAL 。

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

11.6. 弱引用

Python 自动进行内存管理（对大多数的对象进行引用计数和垃圾回收——*garbage collection*——以循环利用）在最后一个引用消失后，内存会很快释放。

这个工作方式对大多数应用程序工作良好，但是偶尔会需要跟踪对象来做一些事。不幸的是，仅仅为跟踪它们创建引用也会使其长期存在。 weakref 模块提供了不用创建引用的跟踪对象工具，一旦对象不再存在，它自动从弱引用表上删除并触发回调。典型的应用包括捕获难以构造的对象:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...             self.value = value
...     def __repr__(self):
...             return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
```

```
>>> del a                           # remove the one reference
>>> gc.collect()                    # run garbage collection right away
0
>>> d['primary']                    # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                    # entry was automatically removed
  File "C:/python33/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7. 列表工具

很多数据结构可能会用到内置列表类型。然而，有时可能需要不同性能代价的实现。

array 模块提供了一个类似列表的 **array()** 对象，它仅仅是存储数据，更为紧凑。以下的示例演示了一个存储双字节无符号整数的数组（类型编码 `"H"` ）而非存储 16 字节 Python 整数对象的普通正规列表

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

collections 模块提供了类似列表的 **deque()** 对象，它从左边添加（append）和弹出（pop）更快，但是在内部查询更慢。这些对象更适用于队列实现和广度优先的树搜索:

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

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

除了链表的替代实现，该库还提供了 **bisect** 这样的模块以操作存储链表:

```
>>> import bisect
```

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

heapq 提供了基于正规链表的堆实现。最小的值总是保持在 0 点。这在希望循环访问最小元素但是不想执行完整堆排序的时候非常有用:

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

11.8. 十进制浮点数算法

decimal 模块提供了一个 Decimal 数据类型用于浮点数计算。相比内置的二进制浮点数实现 float ，这个类型有助于

- 金融应用和其它需要精确十进制表达的场合，
- 控制精度，
- 控制舍入以适应法律或者规定要求，
- 确保十进制数位精度，或者
- 用户希望计算结果与手算相符的场合。

例如，计算 70 分电话费的 5% 税计算，十进制浮点数和二进制浮点数计算结果的差别如下。如果在分值上舍入，这个差别就很重要了:

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

Decimal 的结果总是保有结尾的 0，自动从两位精度延伸到 4 位。 Decimal 重现了手工的数学运算，这就确保了二进制浮点数无法精确保有的数据精度。

高精度使 Decimal 可以执行二进制浮点数无法进行的模运算和等值测试:

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

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

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

decimal 提供了必须的高精度算法:

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

12. 接下来？

读过这本指南应该会让你有兴趣使用 Python —— 可能你已经期待着用 Python 解决你的实际问题了。可以在哪里进行一步学习？

入门指南是 Python 文档集的一部分。其中的另一些文档包括：

- *library-index*:

  应该浏览一下这份文档，它为标准库中的类型、函数和模块提供了完整（尽管很简略）的参考资料。标准的 Python 发布版包括了 大量 的附加模块。其中有针对读取 Unix 邮箱、接收 HTTP 文档、生成随机数、解析命令行选项、写 CGI 程序、压缩数据以及很多其它任务的模块。略读一下库参考会给你很多解决问题的思路。

- *install-index* 展示了如何安装其他 Python 用户编写的附加模块。
- *reference-index*: 詳細说明了 Python 语法和语义。它读起来很累，不过对于语言本身，有份完整的手册很有用。

其它 Python 资源：

- http://www.python.org:Python 官方网站。它包含代码、文档和 Web 上与 Python 有关的页面链接该网站镜 像于全世界的几处其它问题，类似欧洲、日本和澳大利亚。镜像可能会比主站快，这取决于你的地理位置。
- http://docs.python.org: 快速访问 Python 的文档。
- http://pypi.python.org: Python 包索引，以前昵称为奶酪店，索引了可供下载的，用户创建的 Python 模块。如果你发布了代码，可以注册到这里，这样别人可以找到它。
- http://aspn.activestate.com/ASPN/Python/Cookbook/: Python 食谱是大量的示例代码、大型的集合，和有用的脚本。值得关注的是这次资源已经结集成书，名为《Python 食谱》（O'Reilly & Associates, ISBN 0-596-00797-3。）

与 Python 有关的问题，以及问题报告，可以发到新闻组 *comp.lang.python* ，或者发送到邮件组 python-list@python.org 。新闻组和邮件组是开放的，所以发送的消息可以自动的跟到另一个之后。每天有超过 120 个投递（高峰时有数百）,提问（以及回答）问题，为新功能提建议，发布新模块。在发信之前，请查阅 常见问题 (亦称 FAQ)，或者在 Python 源码发布包的 Misc/ 目录中查阅。邮件组也可以在 http://mail.python.org/pipermail/ 访问。FAQ 回答了很多被反复提到的问题，很可能已经解答了你的问题。

13. 交互式输入行编辑历史回溯

有些版本的 Python 解释器支持输入行编辑和历史回溯，类似 Korn shell 和 GNU bash shell 的功能。这是通过 GNU Readline 库实现的。它支持 Emacs 风格和 vi 风格的编辑。这个库有它自己的文档，在此不重复了。不过，基本的东西很容易演示。交互式编辑和历史查阅在 Unix 和 Cygwin 版中是可选项。

本章 *不是* 马克 哈密尔顿的 PythonWin 包和随 Python 发布的基于 TK 的 IDLE 环境的文档。NT 系统和其它 DOS、Windows 系统上的 DOS 窗中的命令行历史回调，属于另一个话题。

13.1. 行编辑

如果支持，无论解释器打印主提示符或从属提示符，行编辑都会激活。当前行可以用 Emacs 风格的快捷键编辑。其中最重要的是： `C-A` （Control-A）将光标移动到行首， `C-E` 移动到行尾， `C-B` 向左移一个字符， `C-F` 向右移一位。退格向左删除一个符串， `C-D` 向右删除一个字符。 `C-K` 删掉光标右边直到行尾的所有字符， `C-Y` 将最后一次删除的字符串粘贴到光标位置。`C-underscore`（underscores 即下划线，译注）撤销最后一次修改，它可以因积累作用重复。

13.2. 历史回溯

历史代替可以工作。所有非空的输入行都被保存在历史缓存中，获得一个新的提示符的时候，你处于这个缓存的最底的空行。 `C-P` 在历史缓存中上溯一行， `C-N` 向下移一行。历史缓存中的任一行都可以编辑；按下 `Return` 键时将当前行传入解释器。 `C-R` 开始一个增量向前搜索；`C-S` 开始一个向后搜索。

13.3. 快捷键绑定

Readline 库的快捷键绑定和其它一些参数可以通过名为 `~/.inputrc` 的初始化文件的替换命名来定制。快捷键绑定如下形式

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

或者

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

选项可以如下设置

```
set option-name value
```

例如

```
# 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
```

需要注意的是 Python 中默认 Tab 绑定为插入一个 Tab 字符而不是 Readline 库的默认文件名完成函数，如果你想用这个，可以将以下内容插入

```
Tab: complete
```

到你的 ~/.inputrc 中来覆盖它。（当然，如果你真的把 Tab 设置成这样，就很难在后继行中插入缩进。）

```
.. index::
```

module: rlcompleter module: readline

自动完成变量和模块名也可以激活生效。要使之在解释器交互模式中可用，在你的启动文件中加入下面内容:[1]

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

这个操作将 Tab 绑定到完成函数，故按 Tab 键两次会给出建议的完成内容；它查找 Python 命名、当前的局部变量、有效的模块名。对于类似 string.a 这样的文件名，它会解析 '.' 相关的表达式，从返回的结果对象中获取属性，以提供完成建议。需要注意的是，如果对象的 __getattr__() 方法是此表达式的一部分，这可能会执行应用程序定义代码。

更有用的初始化文件可能是下面这个例子这样的。要注意一旦创建的名字没用了，它会删掉它们；因为初始化文件作为解释命令与之在同一个命名空间执行，在交互环境中删除命名带来了边际效应。可能你发现了它体贴的保留了一些导入模块，类似 os ，在解释器的大多数使用场合中都会用到它们。:

```
# 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=~/.pystartup" in bash.

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
```

13.4. 其它交互式解释器

跟早先版本的解释器比，现在已经有了很大的进步。不过，还是有些期待没有完成：它应该在后继行中优美的提供缩进（解释器知道下一行是否需要缩进）建议。完成机制可以使用解释器的符号表。命名检查（或进一步建议）匹配括号、引号等等。

另有一个强化交互式解释器已经存在一段时间了，它就是 IPython，它支持 tab 完成，对象浏览和高级历史管理。它也可以完全定制或嵌入到其它应用程序中。另一个类似的强化交互环境是 bpython 。

Footnotes

[1] 启动交互解释器时，Python 可以执行 **PYTHONSTARTUP** 环境变量所指定的文件内容。

14. 浮点数算法：争议和限制

浮点数在计算机中表达为二进制（binary）小数。例如：十进制小数

```
0.125
```

是 1/10 + 2/100 + 5/1000 的值，同样二进制小数

```
0.001
```

是 0/2 + 0/4 + 1/8。这两个数值相同。唯一的实质区别是第一个写为十进制小数记法，第二个是二进制。

遗憾的是，大多数十进制小数不能精确的表达二进制小数。

这个问题更早的时候首先在十进制中发现。考虑小数形式的 1/3 ，你可以来个十进制的近似值。:

```
0.3
```

或者更进一步的,

```
0.33
```

或者更进一步的,

```
0.333
```

诸如此类。如果你写多少位，这个结果永远不是精确的 1/3 ，但是可以无限接近 1/3 。

同样，无论在二进制中写多少位，十进制数 0.1 都不能精确表达为二进制小数。二进制来表达 1/10 是一个无限循环小数

```
0.0001100110011001100110011001100110011001100110011...
```

在任意无限位数值中中止，你可以得到一个近似。

在一个典型的机器上运行 Python，一共有 53 位的精度来表示一个浮点数，所以当你输入十进制的 `0.1` 的时候，看到是一个二进制的小数

```
0.0001100110011001100110011001100110011001100110011010
```

非常接近，但是不完全等于, 1/10.

这是很容易忘记，存储的值是一个近似的原小数，由于浮体的方式，显示在提示符的解释。Python 中只打印一个小数近似的真实机器所存储的二进制近似的十进制值。如果 Python 要打印存储的二进制近似真实的十进制值 0.1，那就要显示

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

认识到这个幻觉的真相很重要：机器不能精确表达 1/10，你可以简单的截断 显示 真正的机器值。 这里还有另一个惊奇之处。例如，下面

```
>>> 0.1 + 0.2
0.30000000000000004
```

需要注意的是这在二进制浮点数是非常自然的：它不是 Python 的 bug，也不是你的代码的 bug。你会看到只要你的硬件支持浮点数算法，所有的语言都会有这个现象（尽管有些语言可能默认或完全不 *显示* 这个差异）

由于小数 2.675 是 2.67 和 2.68 的正中间，你可能期望的结果（二进制近似）2.68。这不是，因为当十进制字符串“2.675”转换为二进制浮点数，再换成一个二进制近似，其精确值

```
2.674999999999999822364316059974953532218933105468750
```

这个问题在于存储“0.1”的浮点值已经达到 1/10 的最佳精度了，所以尝试截断它不能改善：它已经尽可能的好了。 另一个影响是因为 0.1 不能精确的表达 1/10，对 10 个 0.1 的值求和不能精确的得到 1.0，即

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

浮点数据算法产生了很多诸如此类的惊奇。在“表现错误”一节中，这个 “0.1”问题详细表达了精度问问题。更完整的其它常见的惊奇请参见 浮点数危害 。 最后我要说，“没有简单的答案”。还是不要过度的敌视浮点数！

Python 浮点数操作的错误来自于浮点数硬件，大多数机器上同类的问题每次计算误差不超过 2**53 分之一。对于大多数任务这已经足够让人满意了。但是你要在心中记住这不是十进制算法，每个浮点数计算可能会带来一个新的精度错误。

问题已经存在了，对于大多数偶发的浮点数错误，你应该比对你期待的最终显示结果是否符合你的期待。`str()`通常够用了，完全的控制参见 formatstrings 中 `str.format()` 方法的格式化方式。

14.1. 表达错误

这一节详细说明“0.1”示例，教你怎样自己去精确的分析此类案例。假设这里你已经对浮点数表示有基本的了解。

Representation error 提及事实上有些（实际是大多数）十进制小数不能精确的表示为二进制小数。这是 Python （或 Perl，C，C++，Java，Fortran 以及其它很多）语言往往不能按你期待的样子显示十进制数值的根本原因

```
>>> 0.1 + 0.2
0.30000000000000004
```

这 是为什么？ 1/10 不能精确的表示为二进制小数。大多数今天的机器（2000 年十一月）使用 IEEE-754 浮点数算法，大多数平台上 Python 将浮点数映射为 IEEE-754 “双精度浮点数”。754 双精度包含 53 位精度，所以计算机努力将输入的 0.1 转为 J/2**N 最接近的二进制小数。*J* 是一个 53 位的整数。改写

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

为

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

J 重现时正是 53 位（是 >= 2**52 而非 < 2**53 ）， N 的最佳值是 56

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

因此，56 是保持 J 精度的唯一 N 值。 J 最好的近似值是整除的商

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

因为余数大于 10 的一半，最好的近似是取上界

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

因此在 754 双精度中 1/10 最好的近似值是是 2**56，或

```
7205759403792794 / 72057594037927936
```

要注意因为我们向上舍入，它其实比 1/10 稍大一点点。如果我们没有向上舍入，它会比 1/10 稍小一点。但是没办法让它 恰好 是 1/10！

所以计算机永远也不 “知道” 1/10：它遇到上面这个小数，给出它所能得到的最佳的 754 双精度实数

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

如果我们用 10**30 除这个小数，会看到它最大 30 位（截断后的）的十进制值

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

这表示存储在计算机中的实际值近似等于十进制值 0.100000000000000005551115123125。Python 显示时取 17 位精度为 0.10000000000000001（是的，在任何符合 754 的平台上，都会由其 C 库转换为这个最佳近似——你的可能不一样！）。