Python 3.8 官方教程

Python 是一种易于学习又功能强大的编程语言。它提供了高效的高级数据结构，还能简单有效地面向对象编程。Python 优雅的语法和动态类型，以及解释型语言的本质，使它成为多数平台上写脚本和快速开发应用的理想…

目　录

致谢

当前文档 《Python 3.8 官方教程》 由 进击的皇虫 使用 书栈网 (BookStack.CN) 进行构建，生成于 2019-11-26。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：Python官网 https://docs.python.org/zh-cn/3/tutorial/index.html

文档地址：http://www.bookstack.cn/books/python-3.8-tutorial-zh

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Python 教程

Python 是一种易于学习又功能强大的编程语言。它提供了高效的高级数据结构，还能简单有效地面向对象编程。Python 优雅的语法和动态类型，以及解释型语言的本质，使它成为多数平台上写脚本和快速开发应用的理想语言。

Python 解释器及丰富的标准库，提供了适用于各个主要系统平台的源码或机器码，这些可以到 Python 官网 https://www.python.org/ 免费获取，并可自由地分发。许多免费的第三方 Python 模块、程序、工具和它们的文档，也能在这个网站上找到对应内容或链接。

Python 解释器易于扩展，可以使用 C 或 C++（或者其他可以通过 C 调用的语言）扩展新的功能和数据类型。Python 也可用于可定制化软件中的扩展程序语言。

这个教程非正式地介绍了 Python 语言和系统的基本概念和功能。最好在阅读的时候准备一个 Python 解释器进行练习。所有的例子都是相互独立的，所以这个教程也可以离线阅读。

有关标准的对象和模块，请参阅 Python 标准库。Python 语言参考 提供了更正式的语言定义。想要编写 C 或者 C++ 扩展可以参考 扩展和嵌入 Python 解释器 和 Python/C API 参考手册。另外还有不少书籍深入讲解 Python。

这个教程并没有完整地介绍每一个功能，甚至可能没有涉及全部的常用功能。这个教程只介绍 Python 中最值得注意的功能，也会让你体会到这个语言的风格特色。学习完这个教程，你将可以阅读和编写 Python 模块和程序，也可以开始学习 Python 标准库。

术语对照表 也很值得一读。

1. 课前甜点

如果你经常在电脑上工作，总会有些任务会想让它自动化。比如，对一大堆文本文件进行查找替换，对很多照片文件按照比较复杂的规则重命名并放入不同的文件夹。也可能你想写一个小型的数据库应用，一个特定的界面应用，或者一个简单的游戏。

如果你是专业的软件开发人员，你可能需要编写一些C/C++/Java库，但总觉得通常的开发的流程（编写、编译、测试、再次编译等）太慢了。可能给这样的库写一组测试，就是很麻烦的工作了。或许你写了个软件，可以支持插件扩展语言，但你不想为了自己这一个应用，专门设计和实现一种新语言了。

那么，Python正好能满足你的需要。

对于这些任务，你也可以写Unix脚本或者Windows批处理完成，但是shell脚本最擅长移动文件和替换文本，并不适合GUI界面或者游戏开发。你可以写一个C/C++/Java程序，但是可能第一版本的草稿都要很长的开发时间。Python的使用则更加简单，可以在Windows，Mac OS X，以及Unix操作系统上使用，而且可以帮你更快地完成工作。

Python很容易使用，但它是一种真正的编程语言，提供了很多数据结构，也支持大型程序，远超shell脚本或批处理文件的功能。Python还提供比C语言更多的错误检查，而且作为一种“超高级语言”，它有高级的内置数据类型，比如灵活的数组和字典。正因为这些更加通用的数据类型，Python能够应付更多的问题，超过Awk甚至Perl，而且很多东西在Python中至少和那些语言同样简单。

Python 允许你划分程序模块，在其他的 Python 程序中重用。它内置了很多的标准模块，你可以在此基础上开发程序——也可以作为例子，开始学习 Python 编程。例如，文件输入输出，系统调用，套接字，甚至图形界面接口工作包比如 Tk。

Python是一种解释型语言，在程序开发阶段可以为你节省大量时间，因为不需要编译和链接。解释器可以交互式使用，这样就可以方便地尝试语言特性，写一些一次性的程序，或者在自底向上的程序开发中测试功能。它也是一个顺手的桌面计算器。

Python程序的书写是紧凑而易读的。Python代码通常比同样功能的C，C++，Java代码要短很多，原因列举如下：

- 高级数据类型允许在一个表达式中表示复杂的操作；
- 代码块的划分是按照缩进而不是成对的花括号；
- 不需要预先定义变量或参数。

Python是“可扩展的”：如果你知道怎么写C语言程序，就能很容易地给解释器添

加新的内置函数或模块，不论是让关键的程序以最高速度运行，还是把Python程序链接到只提供预编译程序的库（比如硬件相关的图形库）。一旦你真正链接上了，就能在Python解释器中扩展或者控制C语言编写的应用了。

顺便提一下，这种语言的名字（Python意为“蟒蛇”）来自于BBC节目“Monty Python的飞行马戏团”，而与爬行动物没有关系。在文档中用Monty Python来开玩笑不只是可以的，还是推荐的！

现在你已经对Python跃跃欲试了，想要深入了解一些细节了。因为学习语言的最佳方式是使用它，本教程邀请你一边阅读，一边在Python解释器中玩耍。

在下一章节，会讲解使用解释器的方法。看起来相当枯燥，但是对于尝试后续的例子来说，是非常关键的。

教程的其他部分将通过示例介绍Python语言和系统中的不同功能，开始是比较简单的表达式、语句和数据类型，然后是函数和模块，最终接触一些高级概念，比如异常、用户定义的类。

2. 使用 Python 解释器

2.1. 调用解释器

Python 解释器在其被使用的机器上通常安装为 `/usr/local/bin/python3.8`；将 `/usr/local/bin` 加入你的 Unix 终端的搜索路径就可以通过键入以下命令来启动它：

```
python3.8
```

就能运行了 1 。安装时可以选择安装目录，所以解释器也可能在别的地方；可以问问你身边的 Python 大牛，或者你的系统管理员。（比如 `/usr/local/python` 也是比较常用的备选路径）

在 Windows 机器上当你从 Microsoft Store 安装 Python 之后，`python3.8` 命令将可使用。 如果你安装了 py.exe 启动器，你将可以使用 `py` 命令。 参阅 附录：设置环境变量 了解其他启动 Python 的方式。

在主提示符中输入文件结束字符（在 Unix 系统中是 `Control-D`，Windows 系统中是 `Control-Z`）就退出解释器并返回退出状态为0。如果这样不管用，你还可以写这个命令退出： `quit()` 。

解释器的行编辑功能在支持 GNU Readline 库的系统中也包括交互式编辑，历史替换和代码补全等。 检测是否支持行编辑最快速的方式是在首次出现 Python 提示符时输入 `Control-P`。 如果听到“哔”提示音，就说明支持行编辑；请参阅附录 交互式编辑和编辑历史 了解有关功能键的介绍。 如果什么都没发生，或是回显了 `^P` ，说明不支持行编辑；你只能用退格键从当前行中删除字符。

解释器运行的时候有点像 Unix 命令行：在一个标准输入 tty 设备上调用，它能交互式地读取和执行命令；调用时提供文件名参数，或者有个文件重定向到标准输入的话，它就会读取和执行文件中的 脚本。

另一种启动解释器的方式是 `python -c command [arg] …`，其中 *command* 要换成想执行的指令，就像命令行的 `-c` 选项。由于 Python 代码中经常会包含对终端来说比较特殊的字符，通常情况下都建议用英文单引号把 *command* 括起来。

有些 Python 模块也可以作为脚本使用。可以这样输入： `python -m module [arg] …`，这会执行 *module* 的源文件，就跟你在命令行把路径写全了一样。

在运行脚本的时候，有时可能也会需要在运行后进入交互模式。这种时候在文件参数前，加上选项 `-i` 就可以了。

关于所有的命令行选项，请参考 命令行与环境。

2.1.1. 传入参数

如果可能的话，解释器会读取命令行参数，转化为字符串列表存入 `sys` 模块中

的 `argv` 变量中。执行命令 `import sys` 你可以导入这个模块并访问这个列表。这个列表最少也会有一个元素；如果没有给定输入参数，`sys.argv[0]` 就是个空字符串。如果脚本名是标准输入，`sys.argv[0]` 就是 `'-'` 。使用 `-c` *command* 时，`sys.argv[0]` 就会是 `'-c'` 。如果使用选项 `-m` *module*，`sys.argv[0]` 就是包含目录的模块全名。在 `-c` *command* 或 `-m` *module* 之后的选项不会被解释器处理，而会直接留在 `sys.argv` 中给命令或模块来处理。

2.1.2. 交互模式

在终端（tty）输入并执行指令时，我们说解释器是运行在 交互模式（*interactive mode*）。在这种模式中，它会显示 主提示符（*primary prompt*），提示输入下一条指令，通常用三个大于号（ `>>>` ）表示；连续输入行的时候，它会显示 次要提示符，默认是三个点（ `...` ）。进入解释器时，它会先显示欢迎信息、版本信息、版权声明，然后就会出现提示符：

```
$ python3.8
Python 3.8 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

多行指令需要在连续的多行中输入。比如，以 `if` 为例：

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

有关交互模式的更多内容，请参考 交互模式。

2.2. 解释器的运行环境

2.2.1. 源文件的字符编码

默认情况下，Python 源码文件以 UTF-8 编码方式处理。在这种编码方式中，世界上大多数语言的字符都可以同时用于字符串字面值、变量或函数名称以及注释中—尽管标准库中只用常规的 ASCII 字符作为变量或函数名，而且任何可移植的代码都应该遵守此约定。要正确显示这些字符，你的编辑器必须能识别 UTF-8 编码，而且必须使用能支持打开的文件中所有字符的字体。

如果不使用默认编码，要声明文件所使用的编码，文件的 第一 行要写成特殊的注释。语法如下所示：

```
1. # -*- coding: encoding -*-
```

其中 *encoding* 可以是 Python 支持的任意一种 `codecs` 。

比如，要声明使用 Windows-1252 编码，你的源码文件要写成：

```
1. # -*- coding: cp1252 -*-
```

关于 第一行 规则的一种例外情况是，源码以 UNIX "shebang" 行 开头。这种情况下，编码声明就要写在文件的第二行。例如：

```
1. #!/usr/bin/env python3
2. # -*- coding: cp1252 -*-
```

脚注

- 1
- 在Unix系统中，Python 3.x解释器默认安装后的执行文件并不叫作 `python` ，这样才不会与同时安装的Python 2.x冲突。

3. Python 的非正式介绍

在下面的例子中，通过提示符（>>> 与 ...）的出现与否来区分输入和输出：如果你想复现这些例子，当提示符出现后，你必须在提示符后键入例子中的每一个词；不以提示符开头的那些行是解释器的输出。注意例子中某行中出现第二个提示符意味着你必须键入一个空白行；这是用来结束多行命令的。

这个手册中的许多例子都包含注释，甚至交互性命令中也有。Python中的注释以井号 `#` 开头，并且一直延伸到该文本行结束为止。注释可以出现在一行的开头或者是空白和代码的后边，但是不能出现在字符串中间。字符串中的井号就是井号。因为注释是用来阐明代码的，不会被 Python 解释，所以在键入这些例子时，注释是可以被忽略的。

几个例子：

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

3.1. Python 作为计算器使用

让我们尝试一些简单的 Python 命令。启动解释器，等待界面中的提示符， `>>>` （这应该花不了多少时间）。

3.1.1. 数字

解释器就像一个简单的计算器一样：你可以在里面输入一个表达式然后它会写出答案。 表达式的语法很直接：运算符 `+` 、 `-` 、 `*` 、 `/` 的用法和其他大部分语言一样（比如 Pascal 或者 C 语言）；括号 （ `()` ）用来分组。比如：

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

整数（比如 `2` 、 `4` 、 `20` ）有 `int` 类型，有小数部分的（比如 `5.0` 、 `1.6` ）有 `float` 类型。在这个手册的后半部分我们会看到更多的数值类型。

除法运算 （ `/` ）永远返回浮点数类型。如果要做 floor division 得到一个整数结果（忽略小数部分）你可以使用 `//` 运算符；如果要计算余数，可以使用 `%`

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

在Python中，可以使用 `**` 运算符来计算乘方 1

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

等号（`=`）用于给一个变量赋值。然后在下一个交互提示符之前不会有结果显示出来：

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

如果一个变量未定义(未赋值)，试图使用它时会向你提示错误：

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

Python中提供浮点数的完整支持；包含多种混合类型运算数的运算会把整数转换为浮点数：

```
>>> 4 * 3.75 - 1
14.0
```

在交互模式下，上一次打印出来的表达式被赋值给变量 `_`。这意味着当你把Python用作桌面计算器时，继续计算会相对简单，比如：

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

这个变量应该被使用者当作是只读类型。不要向它显式地赋值—你会创建一个和它名字相同独立的本地变量，它会使用魔法行为屏蔽内部变量。

除了 `int` 和 `float`，Python也支持其他类型的数字，例如 `Decimal` 或者 `Fraction`。Python 也内置对 复数 的支持，使用后缀 `j` 或者 `J` 就可以表示虚数部分（例如 `3+5j`）。

3.1.2. 字符串

除了数字，Python 也可以操作字符串。字符串有多种形式，可以使用单引号（`'……'`），双引号（`"……"`）都可以获得同样的结果 2。反斜杠 `\` 可以用来转义：

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

在交互式解释器中，输出的字符串外面会加上引号，特殊字符会使用反斜杠来转义。 虽然有时这看起来会与输入不一样（外面所加的引号可能会改变），但两个字符串是相同的。 如果字符串中有单引号而没有双引号，该字符串外将加双引号来表示，否则就加单引号。 `print()` 函数会生成可读性更强的输出，即略去两边的引号，并且打印出经过转义的特殊字符：

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

如果你不希望前置了 `\` 的字符转义成特殊字符，可以使用 原始字符串 方式，在引号前添加 `r` 即可：

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

字符串字面值可以跨行连续输入。一种方式是用三重引号：`"""…"""` 或 `'''…'''`。字符串中的回车换行会自动包含到字符串中，如果不想包含，在行尾添加一个 `\` 即可。如下例：

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

将产生如下输出（注意最开始的换行没有包括进来）：

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

字符串可以用 `+` 进行连接（粘到一起），也可以用 `*` 进行重复：

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

相邻的两个或多个 字符串字面值 （引号引起来的字符）将会自动连接到一起.

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

把很长的字符串拆开分别输入的时候尤其有用：

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

只能对两个字面值这样操作，变量或表达式不行：

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  File "<stdin>", line 1
    prefix 'thon'
                ^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
                   ^
SyntaxError: invalid syntax
```

如果你想连接变量，或者连接变量和字面值，可以用 `+` 号：

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

字符串是可以被 索引 （下标访问）的，第一个字符索引是 0。单个字符并没有特殊的类型，只是一个长度为一的字符串：

```
>>> word = 'Python'
```

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

索引也可以用负数，这种会从右边开始数：

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

注意 `-0` 和 `0` 是一样的，所以负数索引从 `-1` 开始。

除了索引，字符串还支持 *切片*。索引可以得到单个字符，而 *切片* 可以获取子字符串：

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

注意切片的开始总是被包括在结果中，而结束不被包括。这使得 `s[:i] + s[i:]` 总是等于 `s`

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

切片的索引有默认值；省略开始索引时默认为0，省略结束索引时默认为到字符串的结束：

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

您也可以这么理解切片：将索引视作指向字符 *之间* ，第一个字符的左侧标为0，最后一个字符的右侧标为 *n* ，其中 *n* 是字符串长度。例如：

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
```

```
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

第一行数标注了字符串非负的索引的位置，第二行标注了对应的负的索引。那么从 *i* 到 *j* 的切片就包括了标有 *i* 和 *j* 的位置之间的所有字符。

对于使用非负索引的切片，如果索引不越界，那么得到的切片长度就是起止索引之差。例如， `word[1:3]` 的长度为2.

使用过大的索引会产生一个错误：

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

但是，切片中的越界索引会被自动处理：

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

Python 中的字符串不能被修改，它们是 immutable 的。因此，向字符串的某个索引位置赋值会产生一个错误：

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

如果需要一个不同的字符串，应当新建一个：

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

内建函数 `len()` 返回一个字符串的长度：

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

参见

- 文本序列类型 —- str
- 字符串是一种 序列类型 ，因此也支持序列类型的各种操作。
- 字符串的方法
- 字符串支持许多变换和查找的方法。
- 格式化字符串字面值
- 内嵌表达式的字符串字面值。
- 格式字符串语法
- 使用 `str.format()` 进行字符串格式化。
- printf 风格的字符串格式化
- 这里详述了使用 `%` 运算符进行字符串格式化。

3.1.3. 列表

Python 中可以通过组合一些值得到多种 复合 数据类型。其中最常用的 列表 ，可以通过方括号括起、逗号分隔的一组值得到。一个 列表 可以包含不同类型的元素，但通常使用时各个元素类型相同：

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

和字符串（以及各种内置的 sequence 类型）一样，列表也支持索引和切片：

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

所有的切片操作都返回一个包含所请求元素的新列表。 这意味着以下切片操作会返回列表的一个 浅拷贝：

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

列表同样支持拼接操作：

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

与 immutable 的字符串不同，列表是一个 mutable 类型，就是说，它自己的内容可以改变：

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

你也可以在列表结尾，通过 `append()` 方法 添加新元素 （我们会在后面解释更多关于方法的内容）：

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

给切片赋值也是可以的，这样甚至可以改变列表大小，或者把列表整个清空：

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

内置函数 `len()` 也可以作用到列表上：

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

也可以嵌套列表 （创建包含其他列表的列表），比如说：

```
>>> a = ['a', 'b', 'c']
```

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

3.2. 走向编程的第一步

当然，我们可以将 Python 用于更复杂的任务，而不是仅仅两个和两个一起添加。 例如，我们可以编写 斐波那契数列 的初始子序列，如下所示：

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

这个例子引入了几个新的特点。

- 第一行含有一个 多重赋值： 变量 `a` 和 `b` 同时得到了新值 0 和 1. 最后一行又用了一次多重赋值， 这体现出了右手边的表达式，在任何赋值发生之前就被求值了。右手边的表达式是从左到右被求值的。

- `while` 循环只要它的条件（这里指： `a < 10` ）保持为真就会一直执行。 Python 和 C 一样，任何非零整数都为真；零为假。这个条件也可以是字符串或是列表的值，事实上任何序列都可以；长度非零就为真，空序列就为假。在这个例子里，判断条件是一个简单的比较。标准的比较操作符的写法和 C 语言里是一样： `<` （小于）、 `>` （大于）、 `==` （等于）、 `<=` （小于或等于）、 `>=` （大于或等于）以及 `!=` （不等于）。

- 循环体 是 缩进的 ：缩进是 Python 组织语句的方式。在交互式命令行里，你得给每个缩进的行敲下 Tab 键或者（多个）空格键。实际上用文本编辑器的话，你要准备更复杂的输入方式；所有像样的文本编辑器都有自动缩进的设置。交互式命令行里，当一个组合的语句输入时， 需要在最后敲一个空白行表示完成（因为语法分析器猜不出来你什么时候打的是最后一行）。注意，在同一块语句中的每一行，都要缩进相同的长度。

- `print()` 函数将所有传进来的参数值打印出来. 它和直接输入你要显示的表达式(比如我们之前在计算器的例子里做的)不一样， print() 能处理多个参数，包括浮点数，字符串。 字符串会打印不带引号的内容， 并且在参数项之间会插入一个空格， 这样你就可以很好的把东西格式化， 像这样：

```
>>> i = 256*256
```

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

关键字参数 *end* 可以用来取消输出后面的换行, 或是用另外一个字符串来结尾:

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

脚注

- 1
- 因为 `  ` 比 `-` 有更高的优先级, 所以 `-3  2` 会被解释成 `-(32)` , 因此结果是 `-9` . 为了避免这个并且得到结果 `9` , 你可以用这个式子 `(-3)  2` .
- 2
- 和其他语言不一样的是, 特殊字符比如说 `\n` 在单引号 (`'…'`) 和双引号 (`"…"`) 里有一样的意义. 这两种引号唯一的区别是, 你不需要在单引号里转义双引号 `"` (但是你必须把单引号转义成 `\'`) , 反之亦然.

4. 其他流程控制工具

除了刚刚介绍过的 `while` 语句，Python 中也会使用其他语言中常见的流程控制语句，只是稍有变化。

4.1. if 语句

可能最为人所熟知的编程语句就是 `if` 语句了。例如：

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

可以有零个或多个 `elif` 部分，以及一个可选的 `else` 部分。 关键字 '`elif`' 是 'else if' 的缩写，适合用于避免过多的缩进。 一个 `if` … `elif` … `elif` … 序列可以看作是其他语言中的 `switch` 或 `case` 语句的替代。

4.2. for 语句

Python 中的 `for` 语句与你在 C 或 Pascal 中可能用到的有所不同。Python 中的 `for` 语句并不总是对算术递增的数值进行迭代（如同 Pascal），或是给予用户定义迭代步骤和暂停条件的能力（如同 C），而是对任意序列进行迭代（例如列表或字符串），条目的迭代顺序与它们在序列中出现的顺序一致。 例如（此处英文为双关语）：

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

在遍历同一个集合时修改该集合的代码可能很难获得正确的结果。通常，更直接的做法是循环遍历该集合的副本或创建新集合：

```
# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. range() 函数

如果你确实需要遍历一个数字序列，内置函数 `range()` 会派上用场。它生成算术级数：

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

给定的终止数值并不在要生成的序列里； `range(10)` 会生成10个值，并且是以合法的索引生成一个长度为10的序列。range也可以以另一个数字开头，或者以指定的幅度增加（甚至是负数；有时这也被叫做 '步进'）

```
range(5, 10)
   5, 6, 7, 8, 9

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

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

要以序列的索引来迭代，您可以将 `range()` 和 `len()` 组合如下：

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

然而，在大多数这类情况下，使用 `enumerate()` 函数比较方便，请参见 循环的技巧 。

如果你只打印 range，会出现奇怪的结果：

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

`range()` 所返回的对象在许多方面表现得像一个列表，但实际上却并不是。此对象会在你迭代它时基于所希望的序列返回连续的项，但它没有真正生成列表，这样就能节省空间。

我们称这样的对象为 `iterable`，也就是说，适合作为这样的目标对象：函数和结构期望从中获取连续的项直到所提供项全部耗尽。 我们已经看到 `for` 语句就是这样一种结构，而接受可迭代对象的函数的一个例子是 `sum()` ：

```
1. >>> sum(range(4))  # 0 + 1 + 2 + 3
2. 6
```

稍后我们将看到更多返回可迭代对象以及将可迭代对象作为参数的函数。 最后，也许你会很好奇如何从一个指定范围内获取一个列表。 以下是解决方案：

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

在 数据结构 章节中，我们将讨论 `list()` 的更多细节。

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 `for` 或 `while` 循环.

循环语句可能带有 `else` 子句；它会在循环耗尽了可迭代对象（使用 `for` ）或循环条件变为假值（使用 `while` ）时被执行，但不会在循环被 `break` 语句终止时被执行。 以下搜索素数的循环就是这样的一个例子：

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

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

当和循环一起使用时， `else` 子句与 `try` 语句中的 `else` 子句的共同点多于 `if` 语句中的同类子句：a `try` 语句中的 `else` 子句会在未发生异常时执行，而循环中的 `else` 子句则会在未发生 `break` 时执行。 有关 `try` 语句和异常的更多信息，请参阅 处理异常。

`continue` 语句也是借鉴自 C 语言，表示继续循环中的下一次迭代：

```
>>> 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。当语法上需要一个语句，但程序需要什么动作也不做时，可以使用它。例如：

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

这通常用于创建最小的类：

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

`pass` 的另一个可以使用的场合是在你编写新的代码时作为一个函数或条件子句体的占位符，允许你保持在更抽象的层次上进行思考。 `pass` 会被静默地忽略：

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

4.6. 定义函数

我们可以创建一个输出任意范围内 Fibonacci 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` 引入一个函数 定义。它必须后跟函数名称和带括号的形式参数列表。构成函数体的语句从下一行开始，并且必须缩进。

函数体的第一个语句可以（可选的）是字符串文字；这个字符串文字是函数的文档字符串或 *docstring* 。（有关文档字符串的更多信息，请参阅 文档字符串 部分）有些工具使用文档字符串自动生成在线或印刷文档，或者让用户以交互式的形式浏览代码；在你编写的代码中包含文档字符串是一种很好的做法，所以要养成习惯。

函数的 执行 会引入一个用于函数局部变量的新符号表。 更确切地说，函数中所有的变量赋值都将存储在局部符号表中；而变量引用会首先在局部符号表中查找，然后是外层函数的局部符号表，最后是内置名称表。 因此，全局变量和外层函数的变量不能在函数内部直接赋值（除非是在 `global` 语句中定义的全局变量，或者是在 `nonlocal` 语句中定义的外层函数的变量），尽管它们可以被引用。

在函数被调用时，实际参数（实参）会被引入被调用函数的本地符号表中；因此，实参是通过 按值调用 传递的（其中 值 始终是对象 引用 而不是对象的值）。[1] 当一个函数调用另外一个函数时，将会为该调用创建一个新的本地符号表。

函数定义会把函数名引入当前的符号表中。函数名称的值具有解释器将其识别为用户定义函数的类型。这个值可以分配给另一个名称，该名称也可以作为一个函数使用。这用作一般的重命名机制：

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

如果你学过其他语言，你可能会认为 `fib` 不是函数而是一个过程，因为它并不

返回值。事实上，即使没有 `return` 语句的函数也会返回一个值，尽管它是一个相当无聊的值。这个值称为 `None` （它是内置名称）。一般来说解释器不会打印出单独的返回值 `None` ，如果你真想看到它，你可以使用 `print()`

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

写一个返回斐波那契数列的列表，而不是打印出来的函数，非常简单：

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

此示例中，像往常一样，演示了一些新的 Python 功能：

- `return` 语句会从函数内部返回一个值。 不带表达式参数的 `return` 会返回 `None` 。 函数执行完毕退出也会返回 `None` 。
- `result.append(a)` 语句调用了列表对象 `result` 的 方法 。方法是“属于”一个对象的函数，它被命名为 `obj.methodname` ，其中 `obj` 是某个对象（也可能是一个表达式）， `methodname` 是由对象类型中定义的方法的名称。不同的类型可以定义不同的方法。不同类型的方法可以有相同的名称而不会引起歧义。（可以使用 类 定义自己的对象类型和方法，请参阅 类 ）示例中的方法 `append()` 是为列表对象定义的；它会在列表的最后添加一个新的元素。在这个示例中它相当于 `result = result + [a]` ，但更高效。

4.7. 函数定义的更多形式

给函数定义有可变数目的参数也是可行的。这里有三种形式，可以组合使用。

4.7.1. 参数默认值

最有用的形式是对一个或多个参数指定一个默认值。这样创建的函数，可以用比定义时允许的更少的参数调用，比如：

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    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 ValueError('invalid user response')
        print(reminder)
```

这个函数可以通过几种方式调用：

- 只给出必需的参数：`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!')`

这个示例还介绍了 `in` 关键字。它可以测试一个序列是否包含某个值。

默认值是在 *定义过程* 中在函数定义处计算的，所以

```
i = 5

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

i = 6
f()
```

会打印 `5` 。

重要警告： 默认值只会执行一次。这条规则在默认值为可变对象（列表、字典以及大多数类实例）时很重要。比如，下面的函数会存储在后续调用中传递给它的参数：

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

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

这将打印出

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

如果你不想要在后续调用之间共享默认值，你可以这样写这个函数：

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

4.7.2. 关键字参数

也可以使用形如 `kwarg=value` 的 关键字参数 来调用函数。例如下面的函数：

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

接受一个必需的参数（ `voltage` ）和三个可选的参数（ `state` ， `action` ，和 `type` ）。这个函数可以通过下面的任何一种方式调用：

```
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
```

但下面的函数调用都是无效的：

```
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
```

在函数调用中，关键字参数必须跟随在位置参数的后面。传递的所有关键字参数必须与函数接受的其中一个参数匹配（比如 `actor` 不是函数 `parrot` 的有效参数），它们的顺序并不重要。这也包括非可选参数，（比如 `parrot(voltage=1000)` 也是有效的）。不能对同一个参数多次赋值。下面是一个因为此限制而失败的例子：

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

当存在一个形式为 `**name` 的正式形参时，它会接收一个字典（参见 映射类型 -- dict），其中包含除了与正式形参相对应的关键字参数以外的所有关键字参数。 这可以与一个形式为 `*name`，接收一个包含除了正式形参列表以外的位置参数的 元组 的正式形参（将在下一小节介绍）组合使用（ `**name` 必须出现在 `*name` 之前。） 例如，如果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)
    for kw in keywords:
        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.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
```

```
8.  sketch : Cheese Shop Sketch
```

注意打印时关键字参数的顺序保证与调用函数时提供它们的顺序是相匹配的。

4.7.3. 特殊参数

默认情况下，函数的参数传递形式可以是位置参数或是显式的关键字参数。 为了确保可读性和运行效率，限制允许的参数传递形式是有意义的，这样开发者只需查看函数定义即可确定参数项是仅按位置、按位置也按关键字，还是仅按关键字传递。

函数的定义看起来可以像是这样：

```
1.  def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
2.        -----------    ----------     ----------
3.          |             |                  |
4.          |        Positional or keyword   |
5.          |                                - Keyword only
6.           -- Positional only
```

在这里 `/` 和 `*` 是可选的。 如果使用这些符号则表明可以通过何种形参将参数值传递给函数：仅限位置、位置或关键字，以及仅限关键字。 关键字形参也被称为命名形参。

4.7.3.1. 位置或关键字参数

如果函数定义中未使用 `/` 和 `*`，则参数可以按位置或按关键字传递给函数。

4.7.3.2. 仅限位置参数

在这里还可以发现更多细节，特定形参可以被标记为 仅限位置。 如果是 仅限位置 的形参，则其位置是重要的，并且该形参不能作为关键字传入。 仅限位置形参要放在 `/` （正斜杠）之前。 这个 `/` 被用来从逻辑上分隔仅限位置形参和其它形参。 如果函数定义中没有 `/`，则表示没有仅限位置形参。

在 `/` 之后的形参可以为 位置或关键字 或 仅限关键字。

4.7.3.3. 仅限关键字参数

要将形参标记为 仅限关键字，即指明该形参必须以关键字参数的形式传入，应在参数列表的第一个 *keyword-only* 形参之前放置一个 `*`。

4.7.3.4. 函数示例

请考虑以下示例函数定义并特别注意 `/` 和 `*` 标记：

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

第一个函数定义 `standard_arg` 是最常见的形式，对调用方式没有任何限制，参数可以按位置也可以按关键字传入：

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

第二个函数 `pos_only_arg` 在函数定义中带有 `/` ，限制仅使用位置形参。：

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got an unexpected keyword argument 'arg'
```

第三个函数 `kwd_only_args` 在函数定义中通过 `*` 指明仅允许关键字参数：

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

而最后一个则在同一函数定义中使用了全部三种调用方式：

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
```

```
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got an unexpected keyword argument 'pos_only'
```

最后，请考虑这个函数定义，它的位置参数 `name` 和 `**kwds` 之间由于存在关键字名称 `name` 而可能产生潜在冲突：

```
def foo(name, **kwds):
    return 'name' in kwds
```

任何调用都不可能让它返回 `True`，因为关键字 `'name'` 将总是绑定到第一个形参。 例如：

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

但使用 `/` （仅限位置参数）就可能做到，因为它允许 `name` 作为位置参数，也允许 `'name'` 作为关键字参数的关键字名称：

```
def foo(name, /, **kwds):
    return 'name' in kwds
>>> foo(1, **{'name': 2})
True
```

换句话说，仅限位置形参的名称可以在 `**kwds` 中使用而不产生歧义。

4.7.3.5. 概括

用例将确定要在函数定义中使用的参数：

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

作为指导：

- 如果你希望形参名称对用户来说不可用，则使用仅限位置形参。 这适用于形参名称没有实际意义，以及当你希望强制规定调用时的参数顺序，或是需要同时收受一些位置形参和任意关键字形参等情况。

- 当形参名称有实际意义，以及显式指定形参名称可使函数定义更易理解，或者当你想要防止用户过于依赖传入参数的位置时，则使用仅限关键字形参。
- 对于 API 来说，使用仅限位置形参可以防止形参名称在未来被修改时造成破坏性的 API 变动。

4.7.4. 任意的参数列表

最后，最不常用的选项是可以使用任意数量的参数调用函数。这些参数会被包含在一个元组里（参见 元组和序列 ）。在可变数量的参数之前，可能会出现零个或多个普通参数。：

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

一般来说，这些 可变参数 将在形式参数列表的末尾，因为它们收集传递给函数的所有剩余输入参数。出现在 *args 参数之后的任何形式参数都是 ‘仅关键字参数’，也就是说它们只能作为关键字参数而不能是位置参数。：

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

4.7.5. 解包参数列表

当参数已经在列表或元组中但要为需要单独位置参数的函数调用解包时，会发生相反的情况。例如，内置的 range() 函数需要单独的 *start* 和 *stop* 参数。如果它们不能单独使用，可以使用 * -操作符 来编写函数调用以便从列表或元组中解包参数：

```
>>> 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]
```

同样的方式，字典可使用 ** 操作符 来提供关键字参数：

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
```

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

4.7.6. Lambda 表达式

可以用 `lambda` 关键字来创建一个小的匿名函数。这个函数返回两个参数的和: `lambda a, b: a+b` 。Lambda函数可以在需要函数对象的任何地方使用。它们在语法上限于单个表达式。从语义上来说，它们只是正常函数定义的语法糖。与嵌套函数定义一样，lambda函数可以引用包含范围的变量:

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

上面的例子使用一个lambda表达式来返回一个函数。另一个用法是传递一个小函数作为参数:

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

4.7.7. 文档字符串

以下是有关文档字符串的内容和格式的一些约定。

第一行应该是对象目的的简要概述。为简洁起见，它不应显式声明对象的名称或类型，因为这些可通过其他方式获得（除非名称恰好是描述函数操作的动词）。这一行应以大写字母开头，以句点结尾。

如果文档字符串中有更多行，则第二行应为空白，从而在视觉上将摘要与其余描述分开。后面几行应该是一个或多个段落，描述对象的调用约定，它的副作用等。

Python解析器不会从Python中删除多行字符串文字的缩进，因此处理文档的工具必须在需要时删除缩进。这是使用以下约定完成的。文档字符串第一行 之后 的第一个非空行确定整个文档字符串的缩进量。（我们不能使用第一行，因为它通常与字符串的开头引号相邻，因此它的缩进在字符串文字中不明显。）然后从字符串的所有行的开头剥离与该缩进 "等效" 的空格。 缩进的行不应该出现，但是如果它

们出现，则应该剥离它们的所有前导空格。应在扩展标签后测试空白的等效性（通常为8个空格）。

下面是一个多行文档字符串的例子：

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

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

4.7.8. 函数标注

函数标注 是关于用户自定义函数中使用的类型的完全可选元数据信息（有关详情请参阅 **PEP 3107** 和 **PEP 484** ）。

函数标注 以字典的形式存放在函数的 `annotations` 属性中，并且不会影响函数的任何其他部分。 形参标注的定义方式是在形参名称后加上冒号，后面跟一个表达式，该表达式会被求值为标注的值。 返回值标注的定义方式是加上一个组合符号 `->` ，后面跟一个表达式，该标注位于形参列表和表示 `def` 语句结束的冒号之间。 下面的示例有一个位置参数，一个关键字参数以及返回值带有相应标注：

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

4.8. 小插曲：编码风格

现在你将要写更长，更复杂的Python代码，是时候讨论一下 代码风格。大多数语言都能使用不同的风格编写（或更简洁，格式化的）；有些比其他的更具有可读性。能让其他人轻松阅读你的代码总是一个好主意，采用一种好的编码风格对此有很大帮助。

对于Python，**PEP 8** 已经成为大多数项目所遵循的风格指南；它促进了一种非常易读且令人赏心悦目的编码风格。每个Python开发人员都应该在某个时候阅读它；以下是为你提取的最重要的几个要点：

- 使用4个空格缩进，不要使用制表符。

4个空格是一个在小缩进（允许更大的嵌套深度）和大缩进（更容易阅读）的一种很好的折中方案。制表符会引入混乱，最好不要使用它。

- 换行，使一行不超过79个字符。

这有助于使用小型显示器的用户，并且可以在较大的显示器上并排放置多个代码文件。

- 使用空行分隔函数和类，以及函数内的较大的代码块。
- 如果可能，把注释放到单独的一行。
- 使用文档字符串。
- 在运算符前后和逗号后使用空格，但不能直接在括号内使用：`a = f(1, 2) + g(3, 4)`。
- 以一致的规则为你的类和函数命名；按照惯例应使用 `UpperCamelCase` 来命名类，而以 `lowercase_with_underscores` 来命名函数和方法。 始终应使用 `self` 来命名第一个方法参数（有关类和方法的更多信息请参阅 初探类）。
- 如果你的代码旨在用于国际环境，请不要使用花哨的编码。Python 默认的 UTF-8 或者纯 ASCII 在任何情况下都能有最好的表现。
- 同样，哪怕只有很小的可能，遇到说不同语言的人阅读或维护代码，也不要在标识符中使用非ASCII字符。

脚注

- 1
- 实际上，通过对象引用调用 会是一个更好的表述，因为如果传递的是可变对象，则调用者将看到被调用者对其做出的任何更改（插入到列表中的元素）。

5. 数据结构

本章将详细介绍一些您已经了解的内容，并添加了一些新内容。

5.1. 列表的更多特性

列表数据类型还有很多的方法。这里是列表对象方法的清单：

- `list.append(x)`
- 在列表的末尾添加一个元素。相当于 `a[len(a):] = [x]` 。
- `list.extend(iterable)`
- 使用可迭代对象中的所有元素来扩展列表。相当于 `a[len(a):] = iterable` 。
- `list.insert(i, x)`
- 在给定的位置插入一个元素。第一个参数是要插入的元素的索引，所以 `a.insert(0, x)` 插入列表头部， `a.insert(len(a), x)` 等同于 `a.append(x)` 。
- `list.remove(x)`
- 移除列表中第一个值为 *x* 的元素。如果没有这样的元素，则抛出 `ValueError` 异常。
- `list.pop([i])`
- 删除列表中给定位置的元素并返回它。如果没有给定位置， `a.pop()` 将会删除并返回列表中的最后一个元素。（ 方法签名中 *i* 两边的方括号表示这个参数是可选的，而不是要你输入方括号。你会在 Python 参考库中经常看到这种表示方法）。
- `list.clear()`
- 删除列表中所有的元素。相当于 `del a[:]` 。
- `list.index(x[, start[, end]])`
- 返回列表中第一个值为 *x* 的元素的从零开始的索引。如果没有这样的元素将会抛出 `ValueError` 异常。

可选参数 *start* 和 *end* 是切片符号，用于将搜索限制为列表的特定子序列。返回的索引是相对于整个序列的开始计算的，而不是 *start* 参数。

- `list.count(x)`
- 返回元素 *x* 在列表中出现的次数。
- `list.sort(key=None, reverse=False)`

- 对列表中的元素进行排序（参数可用于自定义排序，解释请参见 `sorted()` ）。
- `list.reverse()`
- 反转列表中的元素。
- `list.copy()`
- 返回列表的一个浅拷贝。相当于 `a[:]` 。

列表方法示例：

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

你可能已经注意到，像 `insert` ， `remove` 或者 `sort` 方法，只修改列表，没有打印出返回值—它们返回默认值 `None` 。[1] 这是Python中所有可变数据结构的设计原则。

你可能会注意到的另一件事是并非所有数据或可以排序或比较。 例如， `[None, 'hello', 10]` 就不可排序，因为整数不能与字符串比较，而 *None* 不能与其他类型比较。 并且还存在一些没有定义顺序关系的类型。 例如， `3+4j < 5+7j` 就不是一个合法的比较。

5.1.1. 列表作为栈使用

列表方法使得列表作为堆栈非常容易，最后一个插入，最先取出（“后进先出”）。要添加一个元素到堆栈的顶端，使用 `append()` 。要从堆栈顶部取出一个元素，使用 `pop()` ，不用指定索引。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. 列表作为队列使用

列表也可以用作队列，其中先添加的元素被最先取出 (“先进先出”)；然而列表用作这个目的相当低效。因为在列表的末尾添加和弹出元素非常快，但是在列表的开头插入或弹出元素却很慢 (因为所有的其他元素都必须移动一位)。

若要实现一个队列， `collections.deque` 被设计用于快速地从两端操作。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

列表推导式提供了一个更简单的创建列表的方法。常见的用法是把某种操作应用于序列或可迭代对象的每个元素上，然后使用其结果来创建列表，或者通过满足某些特定条件元素来创建子序列。

例如，假设我们想创建一个平方列表，像这样

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

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

注意这里创建（或被重写）的名为 `x` 的变量在for循环后仍然存在。我们可以计算平方列表的值而不会产生任何副作用

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

或者，等价于

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

上面这种写法更加简洁易读。

列表推导式的结构是由一对方括号所包含的以下内容：一个表达式，后面跟一个 `for` 子句，然后是零个或多个 `for` 或 `if` 子句。 其结果将是一个新列表，由对表达式依据后面的 `for` 和 `if` 子句的内容进行求值计算而得出。 举例来说，以下列表推导式会将两个列表中不相等的元素组合起来：

```
>>> [(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)]
```

而它等价于

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

注意在上面两个代码片段中， `for` 和 `if` 的顺序是相同的。

如果表达式是一个元组（例如上面的 `(x, y)` ），那么就必须加上括号

```
>>> 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 <module>
    [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]
```

列表推导式可以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

列表推导式中的初始表达式可以是任何表达式，包括另一个列表推导式。

考虑下面这个 3x4的矩阵，它由3个长度为4的列表组成

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

下面的列表推导式将交换其行和列

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

如上节所示，嵌套的列表推导式是基于跟随其后的 `for` 进行求值的，所以这个例子等价于：

```
>>> 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，也等价于

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

实际应用中，你应该会更喜欢使用内置函数去组成复杂的流程语句。 `zip()` 函数将会很好地处理这种情况

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

关于本行中星号的详细说明，参见 解包参数列表。

5.2. del 语句

有一种方式可以从列表按照给定的索引而不是值来移除一个元素： 那就是 `del` 语句。 它不同于会返回一个值的 `pop()` 方法。 `del` 语句也可以用来从列表中移除切片或者清空整个列表（我们之前用过的方式是将一个空列表赋值给指定的切片）。 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` 也可以被用来删除整个变量

```
>>> del a
```

此后再引用 `a` 时会报错（直到另一个值被赋给它）。我们会在后面了解到 `del` 的其他用法。

5.3. 元组和序列

我们看到列表和字符串有很多共同特性，例如索引和切片操作。他们是 序列 数据类型（参见 序列类型 —— list, tuple, range）中的两种。随着 Python 语言的发展，其他的序列类型也会被加入其中。这里介绍另一种标准序列类型：元组。

一个元组由几个被逗号隔开的值组成，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，元组在输出时总是被圆括号包围的，以便正确表示嵌套元组。输入时圆括号可有可无，不过经常会是必须的（如果这个元组是一个更大的表达式的一部分）。给元组中的一个单独的元素赋值是不允许的，当然你可以创建包含可变对象的元组，例如列表。

虽然元组可能看起来与列表很像，但它们通常是在不同的场景被使用，并且有着不同的用途。元组是 immutable （不可变的），其序列通常包含不同种类的元素，并且通过解包（这一节下面会解释）或者索引来访问（如果是 `namedtuples` 的话甚至还可以通过属性访问）。列表是 mutable （可变的），并且列表中的元素一般是同种类型的，并且通过迭代访问。

一个特殊的问题是构造包含0个或1个元素的元组：为了适应这种情况，语法有一些额外的改变。空元组可以直接被一对空圆括号创建，含有一个元素的元组可以通过在这个元素后添加一个逗号来构建（圆括号里只有一个值的话不够明确）。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 元组打包 的一个例子：值 `12345` ，`54321` 和 `'hello!'` 被打包进元组。其逆操作也是允许的

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

这被称为 序列解包 也是很恰当的，因为解包操作的等号右侧可以是任何序列。序列解包要求等号左侧的变量数与右侧序列里所含的元素数相同。注意可变参数其实也只是元组打包和序列解包的组合。

5.4. 集合

Python也包含有 集合 类型。集合是由不重复元素组成的无序的集。它的基本用法包括成员检测和消除重复元素。集合对象也支持像 联合，交集，差集，对称差分等数学运算。

花括号或 `set()` 函数可以用来创建集合。注意：要创建一个空集合你只能用 `set()` 而不能用 `{}` ，因为后者是创建一个空字典，这种数据结构我们会在下一节进行讨论。

以下是一些简单的示例：

```
>>> 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 a or b or both
{'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'}
```

类似于 列表推导式，集合也支持推导式形式

```
>>> 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 内置数据类型是 字典 (参见 映射类型 — dict)。字典在其他语言里可能会被叫做 联合内存 或 联合数组。与以连续整数为索引的序列不同，字典是以 关键字 为索引的，关键字可以是任意不可变类型，通常是字符串或数字。如果一个元组只包含字符串、数字或元组，那么这个元组也可以用作关键字。但如果元组直接或间接地包含了可变对象，那么它就不能用作关键字。列表不能用作关键字，因为列表可以通过索引、切片或 `append()` 和 `extend()` 之类的方法来改变。

理解字典的最好方式，就是将它看做是一个 键: 值 对的集合，键必须是唯一的（在一个字典中）。一对花括号可以创建一个空字典: `{}` 。另一种初始化字典的方式是在一对花括号里放置一些以逗号分隔的键值对，而这也是字典输出的方式。

字典主要的操作是使用关键字存储和解析值。也可以用 `del` 来删除一个键值对。如果你使用了一个已经存在的关键字来存储值，那么之前与这个关键字关联的值就会被遗忘。用一个不存在的键来取值则会报错。

对一个字典执行 `list(d)` 将返回包含该字典中所有键的列表，按插入次序排列 (如需其他排序，则要使用 `sorted(d)`)。要检查字典中是否存在一个特定键，可使用 `in` 关键字。

以下是使用字典的一些简单示例

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

`dict()` 构造函数可以直接从键值对序列里创建字典。

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

```
2. {'sape': 4139, 'guido': 4127, 'jack': 4098}
```

此外，字典推导式可以从任意的键值表达式中创建字典

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

当关键字是简单字符串时，有时直接通过关键字参数来指定键值对更方便

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

5.6. 循环的技巧

当在字典中循环时，用 `items()` 方法可将关键字和对应的值同时取出

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

当在序列中循环时，用 `enumerate()` 函数可以将索引位置和其对应的值同时取出

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

当同时在两个或更多序列中循环时，可以用 `zip()` 函数将其内元素一一匹配。

```
>>> 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.
```

当逆向循环一个序列时，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
```

如果要按某个指定顺序循环一个序列，可以用 `sorted()` 函数，它可以在不改动原序列的基础上返回一个新的排好序的序列

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

```
...     print(f)
...
apple
banana
orange
pear
```

有时可能会想在循环时修改列表内容，一般来说改为创建一个新列表是比较简单且安全的

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7. 深入条件控制

`while` 和 `if` 条件句中可以使用任意操作，而不仅仅是比较操作。

比较操作符 `in` 和 `not in` 校验一个值是否在（或不在）一个序列里。操作符 `is` 和 `is not` 比较两个对象是不是同一个对象，这只跟像列表这样的可变对象有关。所有的比较操作符都有相同的优先级，且这个优先级比数值运算符低。

比较操作可以传递。例如 `a < b == c` 会校验是否 `a` 小于 `b` 并且 `b` 等于 `c`。

比较操作可以通过布尔运算符 `and` 和 `or` 来组合，并且比较操作（或其他任何布尔运算）的结果都可以用 `not` 来取反。这些操作符的优先级低于比较操作符；在它们之中，`not` 优先级最高，`or` 优先级最低，因此 `A and not B or C` 等价于 `(A and (not B)) or C`。和之前一样，你也可以在这种式子里使用圆括号。

布尔运算符 `and` 和 `or` 也被称为 短路 运算符：它们的参数从左至右解析，一旦可以确定结果解析就会停止。例如，如果 `A` 和 `C` 为真而 `B` 为假，那么 `A and B and C` 不会解析 `C`。当作用于普通值而非布尔值时，短路操作符的返回值通常是最后一个变量。

也可以把比较操作或者逻辑表达式的结果赋值给一个变量，例如

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

请注意 Python 与 C 不同，在表达式内部赋值必须显式地使用海象运算符 `:=` 来完成。 这避免了一类This avoids a common class of problems encountered in C 程序中常见的错误：想要在表达式中写 `==` 时却写成了 `=`。

5.8. 序列和其它类型的比较

序列对象通常可以与相同序列类型的其他对象比较。 这种比较使用 字典式 顺序：首先比较开头的两个对应元素，如果两者不相等则比较结果就由此确定；如果两者相等则比较之后的两个元素，以此类推，直到有一个序列被耗尽。 如果要比较的两个元素本身又是相同类型的序列，则会递归地执行字典式顺序比较。 如果两个序列中所有的对应元素都相等，则两个序列也将被视为相等。 如果一个序列是另一个的初始子序列，则较短的序列就被视为较小（较少）。 对于字符串来说，字典式顺序是使用 Unicode 码位序号对单个字符排序。 下面是一些相同类型序列之间比较的例子：

```
1. (1, 2, 3)              < (1, 2, 4)
2. [1, 2, 3]              < [1, 2, 4]
3. 'ABC' < 'C' < 'Pascal' < 'Python'
4. (1, 2, 3, 4)           < (1, 2, 4)
5. (1, 2)                 < (1, 2, -1)
6. (1, 2, 3)             == (1.0, 2.0, 3.0)
7. (1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

注意对不同类型对象来说，只要待比较对象提供了合适的比较方法，就可以使用 `<` 和 `>` 来比较。例如，混合数值类型是通过他们的数值进行比较的，所以 0 等于 0.0，等等。否则，解释器将抛出一个 `TypeError` 异常，而不是随便给出一个结果。

脚注

- 1
- 别的语言可能会返回一个可变对象，他们允许方法连续执行，例如 `d->insert("a")->remove("b")->sort();` 。

6. 模块

如果你从Python解释器退出并再次进入，之前的定义（函数和变量）都会丢失。因此，如果你想编写一个稍长些的程序，最好使用文本编辑器为解释器准备输入并将该文件作为输入运行。这被称作编写 脚本 。随着程序变得越来越长，你或许会想把它拆分成几个文件，以方便维护。你亦或想在不同的程序中使用一个便捷的函数， 而不必把这个函数复制到每一个程序中去。

为支持这些，Python有一种方法可以把定义放在一个文件里，并在脚本或解释器的交互式实例中使用它们。这样的文件被称作 模块 ；模块中的定义可以 导入 到其它模块或者 主 模块（你在顶级和计算器模式下执行的脚本中可以访问的变量集合）。

模块是一个包含Python定义和语句的文件。文件名就是模块名后跟文件后缀 `.py` 。在一个模块内部，模块名（作为一个字符串）可以通过全局变量 `__name__` 的值获得。例如，使用你最喜爱的文本编辑器在当前目录下创建一个名为 `fibo.py` 的文件， 文件中含有以下内容：

```
# Fibonacci numbers module

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

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

现在进入Python解释器，并用以下命令导入该模块：

```
>>> import fibo
```

在当前的符号表中，这并不会直接进入到定义在 `fibo` 函数内的名称；它只是进入到模块名 `fibo` 中。你可以用模块名访问这些函数：

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

```
>>> fibo.__name__
'fibo'
```

如果你想经常使用某个函数，你可以把它赋值给一个局部变量：

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. 有关模块的更多信息

模块可以包含可执行的语句以及函数定义。这些语句用于初始化模块。它们仅在模块 第一次 在 import 语句中被导入时才执行。 1 （当文件被当作脚本运行时，它们也会执行。）

每个模块都有它自己的私有符号表，该表用作模块中定义的所有函数的全局符号表。因此，模块的作者可以在模块内使用全局变量，而不必担心与用户的全局变量发生意外冲突。另一方面，如果你知道自己在做什么，则可以用跟访问模块内的函数的同样标记方法，去访问一个模块的全局变量， `modname.itemname` 。

模块可以导入其它模块。习惯上但不要求把所有 `import` 语句放在模块（或脚本）的开头。被导入的模块名存放在调入模块的全局符号表中。

`import` 语句有一个变体，它可以把名字从一个被调模块内直接导入到现模块的符号表里。例如：

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这并不会把被调模块名引入到局部变量表里（因此在这个例子里， `fibo` 是未被定义的）。

还有一个变体甚至可以导入模块内定义的所有名称：

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这会调入所有非以下划线（ `_` ）开头的名称。 在多数情况下，Python程序员都不会使用这个功能，因为它在解释器中引入了一组未知的名称，而它们很可能会覆盖一些你已经定义过的东西。

注意通常情况下从一个模块或者包内调入 `*` 的做法是不太被接受的， 因为这通常会导致代码的可读性很差。不过，在交互式编译器中为了节省打字可以这么用。

如果模块名称之后带有 `as` ，则跟在 `as` 之后的名称将直接绑定到所导入的模块。

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这会和 `import fibo` 方式一样有效地调入模块， 唯一的区别是它以 `fib` 的名称存在的。

It can also be used when utilising `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

注解

出于效率的考虑，每个模块在每个解释器会话中只被导入一次。因此，如果你更改了你的模块，则必须重新启动解释器， 或者，如果它只是一个要交互式地测试的模块，请使用 `importlib.reload()` ，例如 `import importlib; importlib.reload(modulename)` 。

6.1.1. 以脚本的方式执行模块

当你用下面方式运行一个Python模块:

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

模块里的代码会被执行，就好像你导入了模块一样，但是 `__name__` 被赋值为 `"__main__"` 。 这意味着通过在你的模块末尾添加这些代码:

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

你既可以把这个文件当作脚本又可当作一个可调入的模块来使用， 因为那段解析命令行的代码只有在当模块是以“main”文件的方式执行的时候才会运行:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

如果模块是被导入的，那些代码是不运行的:

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

这经常用于为模块提供一个方便的用户接口，或用于测试（以脚本的方式运行模块从而执行一些测试套件）。

6.1.2. 模块搜索路径

当一个名为 `spam` 的模块被导入的时候，解释器首先寻找具有该名称的内置模块。如果没有找到，然后解释器从 `sys.path` 变量给出的目录列表里寻找名为 `spam.py` 的文件。 `sys.path` 初始有这些目录地址：

- 包含输入脚本的目录（或者未指定文件时的当前目录）。
- `PYTHONPATH` （一个包含目录名称的列表，它和shell变量 `PATH` 有一样的语法）。
- 取决于安装的默认设置

注解

在支持符号链接的文件系统上，包含输入脚本的目录是在追加符号链接后才计算出来的。换句话说，包含符号链接的目录并 没有 被添加到模块的搜索路径上。

在初始化后，Python程序可以更改 `sys.path` 。包含正在运行脚本的文件目录被放在搜索路径的开头处， 在标准库路径之前。这意味着将加载此目录里的脚本，而不是标准库中的同名模块。 除非有意更换，否则这是错误。更多信息请参阅 标准模块。

6.1.3. “编译过的”Python文件

为了加速模块载入，Python在 `__pycache__` 目录里缓存了每个模块的编译后版本，名称为 `module.version.pyc` ，其中名称中的版本字段对编译文件的格式进行编码； 它一般使用Python版本号。例如，在CPython版本3.3中，spam.py的编译版本将被缓存为 `__pycache__/spam.cpython-33.pyc` 。此命名约定允许来自不同发行版和不同版本的Python的已编译模块共存。

Python根据编译版本检查源的修改日期，以查看它是否已过期并需要重新编译。这是一个完全自动化的过程。此外，编译的模块与平台无关，因此可以在具有不同体系结构的系统之间共享相同的库。

Python在两种情况下不会检查缓存。首先，对于从命令行直接载入的模块，它从来都是重新编译并且不存储编译结果；其次，如果没有源模块，它不会检查缓存。为了支持无源文件（仅编译）发行版本， 编译模块必须是在源目录下，并且绝对不能有源模块。

给专业人士的一些小建议：

- 你可以在Python命令中使用 `-O` 或者 `-OO` 开关， 以减小编译后模块的大小。 `-O` 开关去除断言语句， `-OO` 开关同时去除断言语句和 **doc** 字符串。由于有些程序可能依赖于这些，你应当只在清楚自己在做什么时才使用这个选项。“优化过的”模块有一个 `opt-` 标签并且通常小些。将来的发行版本或许会更改优化的效果。

- 一个从 `.pyc` 文件读出的程序并不会比它从 `.py` 读出时运行的更快，`.pyc` 文件唯一快的地方在于载入速度。
- `compileall` 模块可以为一个目录下的所有模块创建.pyc文件。
- 关于这个过程，**PEP 3147** 中有更多细节，包括一个决策流程图。

6.2. 标准模块

Python附带了一个标准模块库，在单独的文档Python库参考（以下称为“库参考”）中进行了描述。一些模块内置于解释器中；它们提供对不属于语言核心但仍然内置的操作的访问，以提高效率或提供对系统调用等操作系统原语的访问。这些模块的集合是一个配置选项，它也取决于底层平台。例如，`winreg` 模块只在Windows操作系统上提供。一个特别值得注意的模块 `sys`，它被内嵌到每一个Python解释器中。变量 `sys.ps1` 和 `sys.ps2` 定义用作主要和辅助提示的字符串：

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

这两个变量只有在编译器是交互模式下才被定义。

`sys.path` 变量是一个字符串列表，用于确定解释器的模块搜索路径。该变量被初始化为从环境变量 `PYTHONPATH` 获取的默认路径，或者如果 `PYTHONPATH` 未设置，则从内置默认路径初始化。你可以使用标准列表操作对其进行修改：

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

6.3. dir() 函数

内置函数 `dir()` 用于查找模块定义的名称。 它返回一个排序过的字符串列表：

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

如果没有参数， `dir()` 会列出你当前定义的名称：

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

注意：它列出所有类型的名称：变量，模块，函数，等等。

`dir()` 不会列出内置函数和变量的名称。如果你想要这些，它们的定义是在标准模块 `builtins` 中：

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
```

```
10.   'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
11.   'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
12.   'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
13.   'NotImplementedError', 'OSError', 'OverflowError',
14.   'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
15.   'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
16.   'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
17.   'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
18.   'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
19.   'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
20.   'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
21.   '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
22.   'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
23.   'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
24.   'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
25.   'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
26.   'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
27.   'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
28.   'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
29.   'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
30.   'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
31.   'zip']
```

6.4. 包

包是一种通过用“带点号的模块名”来构造 Python 模块命名空间的方法。 例如，模块名 `A.B` 表示 `A` 包中名为 `B` 的子模块。正如模块的使用使得不同模块的作者不必担心彼此的全局变量名称一样，使用加点的模块名可以使得 NumPy 或 Pillow 等多模块软件包的作者不必担心彼此的模块名称一样。

假设你想为声音文件和声音数据的统一处理，设计一个模块集合（一个“包”）。由于存在很多不同的声音文件格式（通常由它们的扩展名来识别，例如： `.wav` ， `.aiff` ， `.au` ），因此为了不同文件格式间的转换，你可能需要创建和维护一个不断增长的模块集合。 你可能还想对声音数据还做很多不同的处理（例如，混声，添加回声，使用均衡器功能，创造人工立体声效果）， 因此为了实现这些处理，你将另外写一个无穷尽的模块流。这是你的包的可能结构（以分层文件系统的形式表示）：

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

当导入这个包时，Python搜索 `sys.path` 里的目录，查找包的子目录。

必须要有 `__init__.py` 文件才能让 Python 将包含该文件的目录当作包。 这样可以防止具有通常名称例如 `string` 的目录在无意中隐藏稍后在模块搜索路径上出现的有效模块。 在最简单的情况下， `__init__.py` 可以只是一个空文件，但它也可以执行包的初始化代码或设置 `__all__` 变量，具体将在后文介绍。

包的用户可以从包中导入单个模块，例如：

```
import sound.effects.echo
```

这会加载子模块 `sound.effects.echo` 。但引用它时必须使用它的全名。

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

导入子模块的另一种方法是

```
from sound.effects import echo
```

这也会加载子模块 `echo` ，并使其在没有包前缀的情况下可用，因此可以按如下方式使用：

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

另一种形式是直接导入所需的函数或变量：

```
from sound.effects.echo import echofilter
```

同样，这也会加载子模块 `echo` ，但这会使其函数 `echofilter()` 直接可用：

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

请注意，当使用 `from package import item` 时，item可以是包的子模块（或子包），也可以是包中定义的其他名称，如函数，类或变量。 `import` 语句首先测试是否在包中定义了item；如果没有，它假定它是一个模块并尝试加载它。如果找不到它，则引发 `ImportError` 异常。

相反，当使用 `import item.subitem.subsubitem` 这样的语法时，除了最后一项之外的每一项都必须是一个包；最后一项可以是模块或包，但不能是前一项中定义的类或函数或变量。

6.4.1. 从包中导入 *

当用户写 `from sound.effects import *` 会发生什么？理想情况下，人们希望这会以某种方式传递给文件系统，找到包中存在哪些子模块，并将它们全部导入。这可能需要很长时间，导入子模块可能会产生不必要的副作用，这种副作用只有在显式导入子模块时才会发生。

唯一的解决方案是让包作者提供一个包的显式索引。 `import` 语句使用下面的规范：如果一个包的 `__init__.py` 代码定义了一个名为 `__all__` 的列表，它会被视为在遇到 `from package import` 时应该导入的模块名列表。在发布该包的新版本时，包作者可以决定是否让此列表保持更新。包作者如果认为从他们的包中导入的操作没有必要被使用，也可以决定不支持此列表。例如，文件

`sound/effects/__init__.py` 可以包含以下代码：

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

这意味着 `from sound.effects import *` 将导入 `sound` 包的三个命名子模块。

如果没有定义 `__all__`，`from sound.effects import *` 语句 不 会从包 `sound.effects` 中导入所有子模块到当前命名空间；它只确保导入了包 `sound.effects`（可能运行任何在 `__init__.py` 中的初始化代码），然后导入包中定义的任何名称。这包括 `__init__.py`` 定义的任何名称（以及显式加载的子模块）。它还包括由之前的 `import` 语句显式加载的包的任何子模块。思考下面的代码：

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

在这个例子中，`echo` 和 `surround` 模块是在执行 `from…import` 语句时导入到当前命名空间中的，因为它们定义在 `sound.effects` 包中。（这在定义了 `__all__` 时也有效。）

虽然某些模块被设计为在使用 `import *` 时只导出遵循某些模式的名称，但在生产代码中它仍然被认为是不好的做法。

请记住，使用 `from package import specific_submodule` 没有任何问题！ 实际上，除非导入的模块需要使用来自不同包的同名子模块，否则这是推荐的表示法。

6.4.2. 子包参考

当包被构造成子包时（与示例中的 `sound` 包一样），你可以使用绝对导入来引用兄弟包的子模块。例如，如果模块 `sound.filters.vocoder` 需要在 `sound.effects` 包中使用 `echo` 模块，它可以使用 `from sound.effects import echo`。

你还可以使用import语句的 `from module import name` 形式编写相对导入。这些导入使用前导点来指示相对导入中涉及的当前包和父包。例如，从 `surround` 模块，你可以使用：

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

请注意，相对导入是基于当前模块的名称进行导入的。由于主模块的名称总是 `"__main__"`，因此用作Python应用程序主模块的模块必须始终使用绝对导入。

6.4.3. 多个目录中的包

包支持另一个特殊属性， `path` 。它被初始化为一个列表，其中包含在执行该文件中的代码之前保存包的文件 `init.py` 的目录的名称。这个变量可以修改；这样做会影响将来对包中包含的模块和子包的搜索。

虽然通常不需要此功能，但它可用于扩展程序包中的模块集。

脚注

- 1
- 实际上，函数定义也是“被执行”的“语句”；模块级函数定义的执行在模块的全局符号表中输入该函数名。

7. 输入输出

有几种方法可以显示程序的输出；数据可以以人类可读的形式打印出来，或者写入文件以供将来使用。本章将讨论一些可能性。

7.1. 更漂亮的输出格式

到目前为止，我们遇到了两种写入值的方法：表达式语句 和 `print()` 函数。（第三种是使用文件对象的 `write()` 方法；标准输出文件可以作为 `sys.stdout` 引用。更多相关信息可参考标准库指南。）

通常，你需要更多地控制输出的格式，而不仅仅是打印空格分隔的值。有几种格式化输出的方法。

- 要使用 格式化字符串字面值 ，请在字符串的开始引号或三引号之前加上一个 `f` 或 `F` 。在此字符串中，你可以在 `{` 和 `}` 字符之间写可以引用的变量或字面值的 Python 表达式。

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- 字符串的 `str.format()` 方法需要更多的手动操作。你仍将使用 `{` 和 `}` 来标记变量将被替换的位置，并且可以提供详细的格式化指令，但你还需要提供要格式化的信息。

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

- 最后，你可以使用字符串切片和连接操作自己完成所有的字符串处理，以创建你可以想象的任何布局。字符串类型有一些方法可以执行将字符串填充到给定列宽的有用操作。

当你不需要花哨的输出而只是想快速显示某些变量以进行调试时，可以使用 `repr()` or `str()` 函数将任何值转化为字符串。

`str()` 函数是用于返回人类可读的值的表示，而 `repr()` 是用于生成解释器可读的表示（如果没有等效的语法，则会强制执行 `SyntaxError` ）对于没有人类可读性的表示的对象， `str()` 将返回和 `repr()` 一样的值。很多值使用任一函数都具有相同的表示，比如数字或类似列表和字典的结构。特殊的是字符串有两个不同的表示。

几个例子：

```
>>> 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'))"
```

`string` 模块包含一个 `Template` 类，它提供了另一种将值替换为字符串的方法，使用类似 `$x` 的占位符并用字典中的值替换它们，但对格式的控制要少的多。

7.1.1. 格式化字符串文字

格式化字符串字面值 （常简称为 f-字符串）能让你在字符串前加上 `f` 和 `F` 并将表达式写成 `{expression}` 来在字符串中包含 Python 表达式的值。

可选的格式说明符可以跟在表达式后面。这样可以更好地控制值的格式化方式。以下示例将pi舍入到小数点后三位：

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

在 `':'` 后传递一个整数可以让该字段成为最小字符宽度。这在使列对齐时很有用。：

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

其他的修饰符可用于在格式化之前转化值。 `'!a'` 应用 `ascii()` ， `'!s'` 应用 `str()` ，还有 `'!r'` 应用 `repr()` ：

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

有关这些格式规范的参考，请参阅参考指南 格式规格迷你语言。

7.1.2. 字符串的 format() 方法

`str.format()` 方法的基本用法如下所示：

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

花括号和其中的字符（称为格式字段）将替换为传递给 `str.format()` 方法的对象。花括号中的数字可用来表示传递给 `str.format()` 方法的对象的位置。

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

如果在 `str.format()` 方法中使用关键字参数，则使用参数的名称引用它们的值。：

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

位置和关键字参数可以任意组合：

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

如果你有一个非常长的格式字符串，你不想把它拆开，那么你最好按名称而不是位置引用变量来进行格式化。这可以通过简单地传递字典和使用方括号 `'[]'` 访问键来完成：

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

这也可以通过使用 `'**'` 符号将表作为关键字参数传递。：

```
>>> 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
```

这在与内置函数 `vars()` 结合使用时非常有用，它会返回包含所有局部变量的字典。

例如，下面几行代码生成一组整齐的列，其中包含给定的整数和它的平方以及立方：

```
>>> 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
```

关于使用 `str.format()` 进行字符串格式化的完整概述，请参阅 格式字符串语法。

7.1.3. 手动格式化字符串

这是同一个平方和立方的表，手动格式化的：

```
>>> 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
```

（注意每列之间的一个空格是通过使用 `print()` 的方式添加的：它总是在其参

数间添加空格。）

字符串对象的 `str.rjust()` 方法通过在左侧填充空格来对给定宽度的字段中的字符串进行右对齐。类似的方法还有 `str.ljust()` 和 `str.center()` 。这些方法不会写入任何东西，它们只是返回一个新的字符串，如果输入的字符串太长，它们不会截断字符串，而是原样返回；这虽然会弄乱你的列布局，但这通常比另一种方法好，后者会在显示值时可能不准确（如果你真的想截断，你可以添加一个切片操作，例如 `x.ljust(n)[:n]` 。）

还有另外一个方法， `str.zfill()` ，它会在数字字符串的左边填充零。它能识别正负号：

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

7.1.4. 旧的字符串格式化方法

`%` 操作符也可以用作字符串格式化。它将左边的参数解释为一个很像 `sprintf()` 风格 的格式字符串，应用到右边的参数，并返回一个由此格式化操作产生的字符串。例如：

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

可在 printf 风格的字符串格式化 部分找到更多信息。

7.2. 读写文件

`open()` 返回一个 `file object`，最常用的有两个参数：`open(filename, mode)`。

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

第一个参数是包含文件名的字符串。第二个参数是另一个字符串，其中包含一些描述文件使用方式的字符。*mode* 可以是 `'r'` ，表示文件只能读取，`'w'` 表示只能写入（已存在的同名文件会被删除），还有 `'a'` 表示打开文件以追加内容；任何写入的数据会自动添加到文件的末尾。`'r+'` 表示打开文件进行读写。*mode* 参数是可选的；省略时默认为 `'r'` 。

通常文件是以 *text mode* 打开的，这意味着从文件中读取或写入字符串时，都会以指定的编码方式进行编码。如果未指定编码格式，默认值与平台相关（参见 `open()`)。在mode 中追加的 `'b'` 则以 *binary mode* 打开文件：现在数据是以字节对象的形式进行读写的。这个模式应该用于所有不包含文本的文件。

在文本模式下读取时，默认会把平台特定的行结束符（Unix 上的 `\n` ，Windows 上的 `\r\n` ）转换为 `\n` 。在文本模式下写入时，默认会把出现的 `\n` 转换回平台特定的结束符。这样在幕后修改文件数据对文本文件来说没有问题，但是会破坏二进制数据例如 `JPEG` 或 `EXE` 文件中的数据。请一定要注意在读写此类文件时应使用二进制模式。

在处理文件对象时，最好使用 `with` 关键字。 优点是当子句体结束后文件会正确关闭，即使在某个时刻引发了异常。 而且使用 `with` 相比等效的 `try` - `finally` 代码块要简短得多：

```
>>> with open('workfile') as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

如果你没有使用 `with` 关键字，那么你应该调用 `f.close()` 来关闭文件并立即释放它使用的所有系统资源。如果你没有显式地关闭文件，Python的垃圾回收器最终将销毁该对象并为你关闭打开的文件，但这个文件可能会保持打开状态一段时间。另外一个风险是不同的Python实现会在不同的时间进行清理。

通过 `with` 语句或者调用 `f.close()` 关闭文件对象后，尝试使用该文件对象将自动失败。：

```
>>> f.close()
>>> f.read()
```

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

7.2.1. 文件对象的方法

本节中剩下的例子将假定你已创建名为 `f` 的文件对象。

要读取文件内容，请调用 `f.read(size)` ，它会读取一些数据并将其作为字符串（在文本模式下）或字节串对象（在二进制模式下）返回。 *size* 是一个可选的数值参数。 当 *size* 被省略或者为负数时，将读取并返回整个文件的内容；如果文件的大小是你的机器内存的两倍就会出现问题。 当取其他值时，将读取并返回至多 *size* 个字符（在文本模式下）或 *size* 个字节（在二进制模式下）。 如果已到达文件末尾， `f.read()` 将返回一个空字符串（ `''` ）。

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

`f.readline()` 从文件中读取一行；换行符（ `\n` ）留在字符串的末尾，如果文件不以换行符结尾，则在文件的最后一行省略。这使得返回值明确无误；如果 `f.readline()` 返回一个空的字符串，则表示已经到达了文件末尾，而空行使用 `'\n'` 表示，该字符串只包含一个换行符。：

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

要从文件中读取行，你可以循环遍历文件对象。这是内存高效，快速的，并简化代码：

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

如果你想以列表的形式读取文件中的所有行，你也可以使用 `list(f)` 或 `f.readlines()` 。

`f.write(string)` 会把 *string* 的内容写入到文件中，并返回写入的字符数。：

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

```
2. 15
```

在写入其他类型的对象之前，需要先把它们转化为字符串（在文本模式下）或者字节对象（在二进制模式下）：

```
1. >>> value = ('the answer', 42)
2. >>> s = str(value)  # convert the tuple to string
3. >>> f.write(s)
4. 18
```

`f.tell()` 返回一个整数，给出文件对象在文件中的当前位置，表示为二进制模式下时从文件开始的字节数，以及文本模式下的不透明数字。

要改变文件对象的位置，请使用 `f.seek(offset, whence)`。 通过向一个参考点添加 *offset* 来计算位置；参考点由 *whence* 参数指定。 *whence* 的 0 值表示从文件开头起算，1 表示使用当前文件位置，2 表示使用文件末尾作为参考点。 *whence* 如果省略则默认值为 0，即使用文件开头作为参考点。

```
1. >>> f = open('workfile', 'rb+')
2. >>> f.write(b'0123456789abcdef')
3. 16
4. >>> f.seek(5)      # Go to the 6th byte in the file
5. 5
6. >>> f.read(1)
7. b'5'
8. >>> f.seek(-3, 2)  # Go to the 3rd byte before the end
9. 13
10. >>> f.read(1)
11. b'd'
```

在文本文件（那些在模式字符串中没有 `b` 的打开的文件）中，只允许相对于文件开头搜索（使用 `seek(0, 2)` 搜索到文件末尾是个例外）并且唯一有效的 *offset* 值是那些能从 `f.tell()` 中返回的或者是零。其他 *offset* 值都会产生未定义的行为。

文件对象有一些额外的方法，例如 `isatty()` 和 `truncate()` ，它们使用频率较低；有关文件对象的完整指南请参阅库参考。

7.2.2. 使用 json 保存结构化数据

字符串可以很轻松地写入文件并从文件中读取出来。数字可能会费点劲，因为 `read()` 方法只能返回字符串，这些字符串必须传递给类似 `int()` 的函数，它会接受类似 `'123'` 这样的字符串并返回其数字值 123。当你想保存诸如嵌套列表和字典这样更复杂的数据类型时，手动解析和序列化会变得复杂。

Python 允许你使用称为 JSON (JavaScript Object Notation) 的流行数据交换格式，而不是让用户不断的编写和调试代码以将复杂的数据类型保存到文件

中。名为 `json` 的标准模块可以采用 Python 数据层次结构，并将它们转化为字符串表示形式；这个过程称为 *serializing* 。从字符串表示中重建数据称为 *deserializing* 。在序列化和反序列化之间，表示对象的字符串可能已存储在文件或数据中，或通过网络连接发送到某个远程机器。

注解

JSON格式通常被现代应用程序用于允许数据交换。许多程序员已经熟悉它，这使其成为互操作性的良好选择。

如果你有一个对象 `x` ，你可以用一行简单的代码来查看它的 JSON 字符串表示：

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

`dumps()` 函数的另一个变体叫做 `dump()` ，它只是将对象序列化为 text file 。因此，如果 `f` 是一个 text file 对象，我们可以这样做：

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

要再次解码对象，如果 `f` 是一个打开的以供阅读的 text file 对象：

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

这种简单的序列化技术可以处理列表和字典，但是在JSON中序列化任意类的实例需要额外的努力。 `json` 模块的参考包含对此的解释。

参见

`pickle` - 封存模块

与 JSON 不同，*pickle* 是一种允许对任意复杂 Python 对象进行序列化的协议。因此，它为 Python 所特有，不能用于与其他语言编写的应用程序通信。默认情况下它也是不安全的：如果数据是由熟练的攻击者精心设计的，则反序列化来自不受信任来源的 pickle 数据可以执行任意代码。

8. 错误和异常

到目前为止，我们还没有提到错误消息，但是如果你已经尝试过那些例子，你可能已经看过了一些错误消息。 目前（至少）有两种可区分的错误：语法错误 和 异常。

8.1. 语法错误

语法错误又称解析错误，可能是你在学习Python 时最容易遇到的错误：

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

解析器会输出出现语法错误的那一行，并显示一个“箭头”，指向这行里面检测到第一个错误。 错误是由箭头指示的位置 上面 的 token 引起的（或者至少是在这里被检测出的）：在示例中，在 `print()` 这个函数中检测到了错误，因为在它前面少了个冒号（ `':'` ）。文件名和行号也会被输出，以便输入来自脚本文件时你能知道去哪检查。

8.2. 异常

即使语句或表达式在语法上是正确的，但在尝试执行时，它仍可能会引发错误。在执行时检测到的错误被称为异常，异常不一定会导致严重后果：你将很快学会如何在Python程序中处理它们。 但是，大多数异常并不会被程序处理，此时会显示如下所示的错误信息：

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

错误信息的最后一行告诉我们程序遇到了什么类型的错误。异常有不同的类型，而其类型名称将会作为错误信息的一部分中打印出来：上述示例中的异常类型依次是： `ZeroDivisionError` ， `NameError` 和 `TypeError` 。作为异常类型打印的字符串是发生的内置异常的名称。对于所有内置异常都是如此，但对于用户定义的异常则不一定如此（虽然这是一个有用的规范）。标准的异常类型是内置的标识符（而不是保留关键字）。

这一行的剩下的部分根据异常类型及其原因提供详细信息。

错误信息的前一部分以堆栈回溯的形式显示发生异常时的上下文。通常它包含列出源代码行的堆栈回溯；但是它不会显示从标准输入中读取的行。

内置异常 列出了内置异常和它们的含义。

8.3. 处理异常

可以编写处理所选异常的程序。请看下面的例子，它会要求用户一直输入，直到输入的是一个有效的整数，但允许用户中断程序（使用 `Control-C` 或操作系统支持的其他操作）；请注意用户引起的中断可以通过引发 `KeyboardInterrupt` 异常来指示。：

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

`try` 语句的工作原理如下。

- 首先，执行 *try* 子句 （ `try` 和 `except` 关键字之间的（多行）语句）。
- 如果没有异常发生，则跳过 *except* 子句 并完成 `try` 语句的执行。
- 如果在执行try 子句时发生了异常，则跳过该子句中剩下的部分。然后，如果异常的类型和 `except` 关键字后面的异常匹配，则执行 except 子句，然后继续执行 `try` 语句之后的代码。
- 如果发生的异常和 except 子句中指定的异常不匹配，则将其传递到外部的 `try` 语句中；如果没有找到处理程序，则它是一个 未处理异常，执行将停止并显示如上所示的消息。

一个 `try` 语句可能有多个 except 子句，以指定不同异常的处理程序。 最多会执行一个处理程序。 处理程序只处理相应的 try 子句中发生的异常，而不处理同一 `try` 语句内其他处理程序中的异常。 一个 except 子句可以将多个异常命名为带括号的元组，例如：

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

如果发生的异常和 `except` 子句中的类是同一个类或者是它的基类，则异常和except子句中的类是兼容的（但反过来则不成立 —- 列出派生类的except 子句与基类兼容）。例如，下面的代码将依次打印 B, C, D

```
class B(Exception):
    pass

class C(B):
```

```
    pass

class D(C):
    pass

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

请注意如果 except 子句被颠倒（把 `except B` 放到第一个），它将打印 B，B，B —- 即第一个匹配的 except 子句被触发。

最后的 except 子句可以省略异常名，以用作通配符。但请谨慎使用，因为以这种方式很容易掩盖真正的编程错误！它还可用于打印错误消息，然后重新引发异常（同样允许调用者处理异常）：

```
import sys

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

`try` … `except` 语句有一个可选的 *else* 子句，在使用时必须放在所有的 except 子句后面。对于在try 子句不引发异常时必须执行的代码来说很有用。例如：

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

使用 `else` 子句比向 `try` 子句添加额外的代码要好，因为它避免了意外捕获

由 `try` … `except` 语句保护的代码未引发的异常。

发生异常时，它可能具有关联值，也称为异常 参数 。参数的存在和类型取决于异常类型。

except 子句可以在异常名称后面指定一个变量。这个变量和一个异常实例绑定，它的参数存储在 `instance.args` 中。为了方便起见，异常实例定义了 `str()` ，因此可以直接打印参数而无需引用 `.args` 。也可以在抛出之前首先实例化异常，并根据需要向其添加任何属性。：

```
>>> 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
```

如果异常有参数，则它们将作为未处理异常的消息的最后一部分（'详细信息'）打印。

异常处理程序不仅处理 try 子句中遇到的异常，还处理 try 子句中调用（即使是间接地）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: division by zero
```

8.4. 抛出异常

`raise` 语句允许程序员强制发生指定的异常。例如：

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

`raise` 唯一的参数就是要抛出的异常。这个参数必须是一个异常实例或者是一个异常类（派生自 `Exception` 的类）。如果传递的是一个异常类，它将通过调用没有参数的构造函数来隐式实例化：

```
raise ValueError  # shorthand for 'raise ValueError()'
```

如果你需要确定是否引发了异常但不打算处理它，则可以使用更简单的 `raise` 语句形式重新引发异常

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5. 用户自定义异常

程序可以通过创建新的异常类来命名它们自己的异常（有关Python 类的更多信息，请参阅 类）。异常通常应该直接或间接地从 `Exception` 类派生。

可以定义异常类，它可以执行任何其他类可以执行的任何操作，但通常保持简单，通常只提供许多属性，这些属性允许处理程序为异常提取有关错误的信息。在创建可能引发多个不同错误的模块时，通常的做法是为该模块定义的异常创建基类，并为不同错误条件创建特定异常类的子类：

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

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

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

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

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

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

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

大多数异常都定义为名称以“Error”结尾，类似于标准异常的命名。

许多标准模块定义了它们自己的异常，以报告它们定义的函数中可能出现的错误。有关类的更多信息，请参见类 类。

8.6. 定义清理操作

`try` 语句有另一个可选子句，用于定义必须在所有情况下执行的清理操作。例如：

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

如果存在 `finally` 子句，则 `finally` 子句将作为 `try` 语句结束前的最后一项任务被执行。 `finally` 子句不论 `try` 语句是否产生了异常都会被执行。 以下几点讨论了当异常发生时一些更复杂的情况：

- 如果在执行 `try` 子句期间发生了异常，该异常可由一个 `except` 子句进行处理。 如果异常没有被 `except` 子句所处理，则该异常会在 `finally` 子句执行之后被重新引发。
- 异常也可能在 `except` 或 `else` 子句执行期间发生。 同样地，该异常会在 `finally` 子句执行之后被重新引发。
- 如果在执行 `try` 语句时遇到一个 `break` ， `continue` 或 `return` 语句，则 `finally` 子句将在执行 `break` ， `continue` 或 `return` 语句之前被执行。
- 如果 `finally` 子句中包含一个 `return` 语句，则 `finally` 子句的 `return` 语句将在执行 `try` 子句的 `return` 语句之前取代后者被执行。

例如

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

一个更为复杂的例子：

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

正如你所看到的，`finally` 子句在任何情况下都会被执行。 两个字符串相除所引发的 `TypeError` 不会由 `except` 子句处理，因此会在 `finally` 子句执行后被重新引发。

在实际应用程序中，`finally` 子句对于释放外部资源（例如文件或者网络连接）非常有用，无论是否成功使用资源。

8.7. 预定义的清理操作

某些对象定义了在不再需要该对象时要执行的标准清理操作，无论使用该对象的操作是成功还是失败。请查看下面的示例，它尝试打开一个文件并把其内容打印到屏幕上。：

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

这个代码的问题在于，它在这部分代码执行完后，会使文件在一段不确定的时间内处于打开状态。这在简单脚本中不是问题，但对于较大的应用程序来说可能是个问题。 `with` 语句允许像文件这样的对象能够以一种确保它们得到及时和正确的清理的方式使用。：

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

执行完语句后，即使在处理行时遇到问题，文件 *f* 也始终会被关闭。和文件一样，提供预定义清理操作的对象将在其文档中指出这一点。

9. 类

类提供了一种组合数据和功能的方法。创建一个新类意味着创建一个新 类型 的对象，从而允许创建一个该类型的新 实例 。每个类的实例可以拥有保存自己状态的属性。一个类的实例也可以有改变自己状态的（定义在类中的）方法。

和其他编程语言相比，Python 用非常少的新语法和语义将类加入到语言中。它是 C++ 和 Modula-3 中类机制的结合。Python 的类提供了面向对象编程的所有标准特性：类继承机制允许多个基类，派生类可以覆盖它基类的任何方法，一个方法可以调用基类中相同名称的的方法。对象可以包含任意数量和类型的数据。和模块一样，类也拥有 Python 天然的动态特性：它们在运行时创建，可以在创建后修改。

在C++术语中，通常类成员（包括数据成员）是 *public* （除了见下文 私有变量 ），所有成员函数都是 *virtual* 。与在Modula-3中一样，没有用于从其方法引用对象成员的简写：方法函数使用表示对象的显式第一个参数声明，该参数由调用隐式提供。与Smalltalk一样，类本身也是对象。这为导入和重命名提供了语义。与C++和Modula-3不同，内置类型可以用作用户扩展的基类。此外，与C++一样，大多数具有特殊语法（算术运算符，下标等）的内置运算符都可以重新定义为类实例。

(Lacking universally accepted terminology to talk about classes, I will makeoccasional use of Smalltalk and C++ terms. I would use Modula-3 terms, sinceits object-oriented semantics are closer to those of Python than C++, but Iexpect that few readers have heard of it.)

9.1. 名称和对象

对象具有个性，多个名称（在多个作用域内）可以绑定到同一个对象。这在其他语言中称为别名。乍一看Python时通常不会理解这一点，在处理不可变的基本类型（数字，字符串，元组）时可以安全地忽略它。但是，别名对涉及可变对象，如列表，字典和大多数其他类型，的Python代码的语义可能会产生惊人的影响。这通常用于程序的好处，因为别名在某些方面表现得像指针。例如，传递一个对象很便宜，因为实现只传递一个指针；如果函数修改了作为参数传递的对象，调用者将看到更改 —- 这就不需要像 Pascal 中那样使用两个不同的参数传递机制。

9.2. Python 作用域和命名空间

在介绍类之前，我首先要告诉你一些Python的作用域规则。类定义对命名空间有一些巧妙的技巧，你需要知道作用域和命名空间如何工作才能完全理解正在发生的事情。顺便说一下，关于这个主题的知识对任何高级Python程序员都很有用。

让我们从一些定义开始。

namespace （命名空间）是一个从名字到对象的映射。 大部分命名空间当前都由 Python 字典实现，但一般情况下基本不会去关注它们（除了要面对性能问题时），而且也有可能在将来更改。 下面是几个命名空间的例子：存放内置函数的集合（包含 `abs()` 这样的函数，和内建的异常等）；模块中的全局名称；函数调用中的局部名称。 从某种意义上说，对象的属性集合也是一种命名空间的形式。 关于命名空间的重要一点是，不同命名空间中的名称之间绝对没有关系；例如，两个不同的模块都可以定义一个 `maximize` 函数而不会产生混淆 —— 模块的用户必须在其前面加上模块名称。

顺便说明一下，我把任何跟在一个点号之后的名称都称为 属性 —— 例如，在表达式 `z.real` 中， `real` 是对象 `z` 的一个属性。按严格的说法，对模块中名称的引用属于属性引用：在表达式 `modname.funcname` 中， `modname` 是一个模块对象而 `funcname` 是它的一个属性。在此情况下在模块的属性和模块中定义的全局名称之间正好存在一个直观的映射：它们共享相同的命名空间！ [1]

属性可以是只读或者可写的。如果为后者，那么对属性的赋值是可行的。模块属性是可以写，你可以写出 `modname.the_answer = 42` 。可写的属性同样可以用 `del` 语句删除。例如， `del modname.the_answer` 将会从名为 `modname` 的对象中移除 `the_answer` 属性。

在不同时刻创建的命名空间拥有不同的生存期。包含内置名称的命名空间是在 Python 解释器启动时创建的，永远不会被删除。模块的全局命名空间在模块定义被读入时创建；通常，模块命名空间也会持续到解释器退出。被解释器的顶层调用执行的语句，从一个脚本文件读取或交互式地读取，被认为是 `__main__` 模块调用的一部分，因此它们拥有自己的全局命名空间。（内置名称实际上也存在于一个模块中；这个模块称作 `builtins` 。）

一个函数的本地命名空间在这个函数被调用时创建，并在函数返回或抛出一个不在函数内部处理的错误时被删除。（事实上，比起描述到底发生了什么，忘掉它更好。）当然，每次递归调用都会有它自己的本地命名空间。

一个 作用域 是一个命名空间可直接访问的 Python 程序的文本区域。 这里的“可直接访问”意味着对名称的非限定引用会尝试在命名空间中查找名称。

Although scopes are determined statically, they are used dynamically. At anytime during execution, there are at least three nested scopes whose namespacesare directly accessible:

- 最先搜索的最内部作用域包含局部名称
- 从最近的封闭作用域开始搜索的任何封闭函数的范围包含非局部名称，也包括非全局名称
- 倒数第二个作用域包含当前模块的全局名称
- 最外面的范围（最后搜索）是包含内置名称的命名空间

如果一个名称被声明为全局变量，则所有引用和赋值将直接指向包含该模块的全局名称的中间作用域。 要重新绑定在最内层作用域以外找到的变量，可以使用 `nonlocal` 语句声明为非本地变量。 如果没有被声明为非本地变量，这些变量将是只读的（尝试写入这样的变量只会在最内层作用域中创建一个 新的 局部变量，而同名的外部变量保持不变）。

通常，当前局部作用域将（按字面文本）引用当前函数的局部名称。 在函数以外，局部作用域将引用与全局作用域相一致的命名空间：模块的命名空间。 类定义将在局部命名空间内再放置另一个命名空间。

重要的是应该意识到作用域是按字面文本来确定的：在一个模块内定义的函数的全局作用域就是该模块的命名空间，无论该函数从什么地方或以什么别名被调用。另一方面，实际的名称搜索是在运行时动态完成的 —- 但是，语言定义在 编译时是朝着静态名称解析的方向演化的，因此不要过于依赖动态名称解析！ （事实上，局部变量已经是被静态确定了。）

Python 的一个特殊之处在于 – 如果不存在生效的 `global` 语句 – 对名称的赋值总是进入最内层作用域。 赋值不会复制数据 —- 它们只是将名称绑定到对象。 删除也是如此：语句 `del x` 会从局部命名空间的引用中移除对 `x` 的绑定。 事实上，所有引入新名称的操作都使用局部作用域：特别地， `import` 语句和函数定义会在局部作用域中绑定模块或函数名称。

`global` 语句可被用来表明特定变量生存于全局作用域并且应当在其中被重新绑定； `nonlocal` 语句表明特定变量生存于外层作用域中并且应当在其中被重新绑定。

9.2.1. 作用域和命名空间示例

This is an example demonstrating how to reference the different scopes andnamespaces, and how `global` and `nonlocal` affect variablebinding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
```

```
8.
9.      def do_global():
10.         global spam
11.         spam = "global spam"
12.
13.     spam = "test spam"
14.     do_local()
15.     print("After local assignment:", spam)
16.     do_nonlocal()
17.     print("After nonlocal assignment:", spam)
18.     do_global()
19.     print("After global assignment:", spam)
20.
21. scope_test()
22. print("In global scope:", spam)
```

示例代码的输出是：

```
1. After local assignment: test spam
2. After nonlocal assignment: nonlocal spam
3. After global assignment: nonlocal spam
4. In global scope: global spam
```

请注意 局部 赋值（这是默认状态）不会改变 *scope_test* 对 *spam* 的绑定。 `nonlocal` 赋值会改变 *scope_test* 对 *spam* 的绑定，而 `global` 赋值会改变模块层级的绑定。

您还可以在 `global` 赋值之前看到之前没有 *spam* 的绑定。

9.3. 初探类

类引入了一些新语法，三种新对象类型和一些新语义。

9.3.1. 类定义语法

最简单的类定义看起来像这样：

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

类定义与函数定义（`def` 语句）一样必须被执行才会起作用。（你可以尝试将类定义放在 `if` 语句的一个分支或是函数的内部。）

在实践中，类定义内的语句通常都是函数定义，但也允许有其他语句，有时还很有用 --- 我们会稍后再回来说明这个问题。 在类内部的函数定义通常具有一种特别形式的参数列表，这是方法调用的约定规范所指明的 --- 这个问题也将在稍后再说明。

当进入类定义时，将创建一个新的命名空间，并将其用作局部作用域 --- 因此，所有对局部变量的赋值都是在这个新命名空间之内。 特别的，函数定义会绑定到这里的新函数名称。

当（从结尾处）正常离开类定义时，将创建一个 类对象。 这基本上是一个包围在类定义所创建命名空间内容周围的包装器；我们将在下一节了解有关类对象的更多信息。 原始的（在进入类定义之前起作用的）局部作用域将重新生效，类对象将在这里被绑定到类定义头所给出的类名称（在这个示例中为 `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 Python 中所有属性引用所使用的标准语法：`obj.name`。 有效的属性名称是类对象被创建时存在于类命名空间中的所有名称。 因此，如果类定义是这样的：

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
```

```
6.          return 'hello world'
```

那么 `MyClass.i` 和 `MyClass.f` 就是有效的属性引用，将分别返回一个整数和一个函数对象。 类属性也可以被赋值，因此可以通过赋值来更改 `MyClass.i` 的值。 `__doc__` 也是一个有效的属性，将返回所属类的文档字符串： `"A simple example class"` 。

类的 *实例化* 使用函数表示法。 可以把类对象视为是返回该类的一个新实例的不带参数的函数。 举例来说（假设使用上述的类）：

```
1. x = MyClass()
```

创建类的新 *实例* 并将此对象分配给局部变量 `x` 。

实例化操作（“调用”类对象）会创建一个空对象。 许多类喜欢创建带有特定初始状态的自定义实例。 为此类定义可能包含一个名为 `__init__()` 的特殊方法，就像这样：

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

当一个类定义了 `__init__()` 方法时，类的实例化操作会自动为新创建的类实例发起调用 `__init__()` 。 因此在这个示例中，可以通过以下语句获得一个经初始化的新实例：

```
1. x = MyClass()
```

当然， `__init__()` 方法还可以有额外参数以实现更高灵活性。 在这种情况下，提供给类实例化运算符的参数将被传递给 `__init__()` 。 例如，：

```
1. >>> class Complex:
2. ...     def __init__(self, realpart, imagpart):
3. ...         self.r = realpart
4. ...         self.i = imagpart
5. ...
6. >>> x = Complex(3.0, -4.5)
7. >>> x.r, x.i
8. (3.0, -4.5)
```

9.3.3. 实例对象

现在我们可以用实例对象做什么？实例对象理解的唯一操作是属性引用。有两种有效的属性名称，数据属性和方法。

数据属性 对应于 Smalltalk 中的“实例变量”，以及 C++ 中的“数据成员”。数据属性不需要声明；像局部变量一样，它们将在第一次被赋值时产生。 例如，

如果 `x` 是上面创建的 `MyClass` 的实例，则以下代码段将打印数值 `16`，且不保留任何追踪信息：

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

另一类实例属性引用称为 方法。 方法是“从属于”对象的函数。 （在 Python 中，方法这个术语并不是类实例所特有的：其他对象也可以有方法。 例如，列表对象具有 append, insert, remove, sort 等方法。 然而，在以下讨论中，我们使用方法一词将专指类实例对象的方法，除非另外显式地说明。）

实例对象的有效方法名称依赖于其所属的类。 根据定义，一个类中所有是函数对象的属性都是定义了其实例的相应方法。 因此在我们的示例中，`x.f` 是有效的方法引用，因为 `MyClass.f` 是一个函数，而 `x.i` 不是方法，因为 `MyClass.i` 不是一个函数。 但是 `x.f` 与 `MyClass.f` 并不是一回事 —— 它是一个 方法对象，不是函数对象。

9.3.4. 方法对象

通常，方法在绑定后立即被调用：

```
x.f()
```

在 `MyClass` 示例中，这将返回字符串 `'hello world'`。 但是，立即调用一个方法并不是必须的： `x.f` 是一个方法对象，它可以被保存起来以后再调用。 例如：

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

将继续打印 `hello world`，直到结束。

当一个方法被调用时到底发生了什么？ 你可能已经注意到上面调用 `x.f()` 时并没有带参数，虽然 `f()` 的函数定义指定了一个参数。 这个参数发生了什么事？ 当不带参数地调用一个需要参数的函数时 Python 肯定会引发异常 —— 即使参数实际未被使用…

实际上，你可能已经猜到了答案：方法的特殊之处就在于实例对象会作为函数的第一个参数被传入。 在我们的示例中，调用 `x.f()` 其实就相当于 `MyClass.f(x)`。 总之，调用一个具有 *n* 个参数的方法就相当于调用再多一个参数的对应函数，这个参数值为方法所属实例对象，位置在其他参数之前。

如果你仍然无法理解方法的运作原理，那么查看实现细节可能会澄清问题。 当一个实例的非数据属性被引用时，将搜索实例所属的类。 如果被引用的属性名称表示一个有效的类属性中的函数对象，会通过打包（指向）查找到的实例对象和函数对象到一个抽象对象的方式来创建方法对象：这个抽象对象就是方法对象。 当附带参数列表调用方法对象时，将基于实例对象和参数列表构建一个新的参数列表，并使用这个新参数列表调用相应的函数对象。

9.3.5. 类和实例变量

一般来说，实例变量用于每个实例的唯一数据，而类变量用于类的所有实例共享的属性和方法：

```
class Dog:

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

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

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

正如 名称和对象 中已讨论过的，共享数据可能在涉及 mutable 对象例如列表和字典的时候导致令人惊讶的结果。 例如以下代码中的 *tricks* 列表不应该被用作类变量，因为所有的 *Dog* 实例将只共享一个单独的列表：

```
class Dog:

    tricks = []             # mistaken use of a class variable

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

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

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

```
['roll over', 'play dead']
```

正确的类设计应该使用实例变量：

```
class Dog:

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

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

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

9.4. 补充说明

如果同样的属性名称同时出现在实例和类中，则属性查找会优先选择实例：

```
>>> class Warehouse:
        purpose = 'storage'
        region = 'west'

>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

数据属性可以被方法以及一个对象的普通用户（“客户端”）所引用。 换句话说，类不能用于实现纯抽象数据类型。 实际上，在 Python 中没有任何东西能强制隐藏数据 —— 它是完全基于约定的。 （而在另一方面，用 C 语言编写的 Python 实现则可以完全隐藏实现细节，并在必要时控制对象的访问；此特性可以通过用 C 编写 Python 扩展来使用。）

客户端应当谨慎地使用数据属性 —— 客户端可能通过直接操作数据属性的方式破坏由方法所维护的固定变量。 请注意客户端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可用性，只要保证避免名称冲突 —— 再次提醒，在此使用命名约定可以省去许多令人头痛的麻烦。

在方法内部引用数据属性（或其他方法！）并没有简便方式。 我发现这实际上提升了方法的可读性：当浏览一个方法代码时，不会存在混淆局部变量和实例变量的机会。

方法的第一个参数常常被命名为 `self`。 这也不过就是一个约定：`self` 这一名称在 Python 中绝对没有特殊含义。 但是要注意，不遵循此约定会使得你的代码对其他 Python 程序员来说缺乏可读性，而且也可以想像一个 类浏览器 程序的编写可能会依赖于这样的约定。

任何一个作为类属性的函数都为该类的实例定义了一个相应方法。 函数定义的文本并非必须包含于类定义之内：将一个函数对象赋值给一个局部变量也是可以的。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 `C` 类的引用函数对象的属性，因而它们就都是 `C` 的实例的方法 —- 其中 `h` 完全等同于 `g`。 但请注意，本示例的做法通常只会令程序的阅读者感到迷惑。

方法可以通过使用 `self` 参数的方法属性调用其他方法：

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

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

方法可以通过与普通函数相同的方式引用全局名称。 与方法相关联的全局作用域就是包含其定义的模块。 （类永远不会被作为全局作用域。） 虽然我们很少会有充分的理由在方法中使用全局作用域，但全局作用域存在许多合法的使用场景：举个例子，导入到全局作用域的函数和模块可以被方法所使用，在其中定义的函数和类也一样。 通常，包含该方法的类本身是在全局作用域中定义的，而在下一节中我们将会发现为何方法需要引用其所属类的很好的理由。

每个值都是一个对象，因此具有 类 （也称为 类型），并存储为 `object.__class__`。

9.5. 继承

当然，如果不支持继承，语言特性就不值得称为“类”。派生类定义的语法如下所示：

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

名称 `BaseClassName` 必须定义于包含派生类定义的作用域中。 也允许用其他任意表达式代替基类名称所在的位置。 这有时也可能会用得上，例如，当基类定义在另一个模块中的时候：

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

派生类定义的执行过程与基类相同。 当构造类对象时，基类会被记住。 此信息将被用来解析属性引用：如果请求的属性在类中找不到，搜索将转往基类中进行查找。 如果基类本身也派生自其他某个类，则此规则将被递归地应用。

派生类的实例化没有任何特殊之处：`DerivedClassName()` 会创建该类的一个新实例。 方法引用将按以下方式解析：搜索相应的类属性，如有必要将按基类继承链逐步向下查找，如果产生了一个函数对象则方法引用就生效。

派生类可能会重载其基类的方法。 因为方法在调用同一对象的其他方法时没有特殊权限，调用同一基类中定义的另一方法的基类方法最终可能会调用覆盖它的派生类的方法。 （对 C++ 程序员的提示：Python 中所有的方法实际上都是 `virtual` 方法。）

在派生类中的重载方法实际上可能想要扩展而非简单地替换同名的基类方法。 有一种方式可以简单地直接调用基类方法：即调用 `BaseClassName.methodname(self, arguments)`。 有时这对客户端来说也是有用的。 （请注意仅当此基类可在全局作用域中以 `BaseClassName` 的名称被访问时方可使用此方式。）

Python有两个内置函数可被用于继承机制：

- 使用 `isinstance()` 来检查一个实例的类型：`isinstance(obj, int)` 仅会在 `obj.__class__` 为 `int` 或某个派生自 `int` 的类时为 `True`。
- 使用 `issubclass()` 来检查类的继承关系：`issubclass(bool, int)` 为 `True`，因为 `bool` 是 `int` 的子类。 但是，`issubclass(float, int)` 为 `False`，因为 `float` 不是 `int` 的子类。

9.5.1. 多重继承

Python supports a form of multiple inheritance as well. A class definition withmultiple base classes looks like this:

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

对于多数应用来说，在最简单的情况下，你可以认为搜索从父类所继承属性的操作是深度优先、从左至右的，当层次结构中存在重叠时不会在同一个类中搜索两次。因此，如果某一属性在 `DerivedClassName` 中未找到，则会到 `Base1` 中搜索它，然后（递归地）到 `Base1` 的基类中搜索，如果在那里未找到，再到 `Base2` 中搜索，依此类推。

真实情况比这个更复杂一些；方法解析顺序会动态改变以支持对 `super()` 的协同调用。 这种方式在某些其他多重继承型语言中被称为后续方法调用，它比单继承型语言中的 super 调用更强大。

动态改变顺序是有必要的，因为所有多重继承的情况都会显示出一个或更多的菱形关联（即至少有一个父类可通过多条路径被最底层类所访问）。 例如，所有类都是继承自 `object`，因此任何多重继承的情况都提供了一条以上的路径可以通向 `object`。 为了确保基类不会被访问一次以上，动态算法会用一种特殊方式将搜索顺序线性化， 保留每个类所指定的从左至右的顺序，只调用每个父类一次，并且保持单调（即一个类可以被子类化而不影响其父类的优先顺序）。 总而言之，这些特性使得设计具有多重继承的可靠且可扩展的类成为可能。 要了解更多细节，请参阅 https://www.python.org/download/releases/2.3/mro/。

9.6. 私有变量

那种仅限从一个对象内部访问的“私有”实例变量在 Python 中并不存在。 但是，大多数 Python 代码都遵循这样一个约定：带有一个下划线的名称 （例如 `_spam` ） 应该被当作是 API 的非仅供部分 （无论它是函数、方法或是数据成员）。 这应当被视为一个实现细节，可能不经通知即加以改变。

由于存在对于类私有成员的有效使用场景（例如避免名称与子类所定义的名称相冲突），因此存在对此种机制的有限支持，称为 名称改写。 任何形式为 `spam` 的标识符（至少带有两个前缀下划线，至多一个后缀下划线）的文本将被替换为 `_classname` `spam` ，其中 `classname` 为去除了前缀下划线的当前类名称。 这种改写不考虑标识符的句法位置，只要它出现在类定义内部就会进行。

名称改写有助于让子类重载方法而不破坏类内方法调用。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)
```

上面的示例即使在 `MappingSubclass` 引入了一个 `update` 标识符的情况下也不会出错，因为它会在 `Mapping` 类中被替换为 `_Mapping` `update` 而在 `MappingSubclass` 类中被替换为 `_MappingSubclass__update` 。

请注意，改写规则的设计主要是为了避免意外冲突；访问或修改被视为私有的变量仍然是可能的。这在特殊情况下甚至会很有用，例如在调试器中。

请注意传递给 `exec()` 或 `eval()` 的代码不会将发起调用类的类名视作当前类；这类似于 `global` 语句的效果，因此这种效果仅限于同时经过字节码编译的代码。 同样的限制也适用于 `getattr()` ， `setattr()` 和 `delattr()` ，以及对于 `dict` 的直接引用。

9.7. 杂项说明

有时会需要使用类似于 Pascal 的“record”或 C 的“struct”这样的数据类型，将一些命名数据项捆绑在一起。 这种情况适合定义一个空类：

```
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
```

一段需要特定抽象数据类型的 Python 代码往往可以被传入一个模拟了该数据类型的方法的类作为替代。 例如，如果你有一个基于文件对象来格式化某些数据的函数，你可以定义一个带有 `read()` 和 `readline()` 方法从字符串缓存获取数据的类，并将其作为参数传入。

实例方法对象也具有属性： `m.self` 就是带有 `m()` 方法的实例对象，而 `m.func` 则是该方法所对应的函数对象。

9.8. 迭代器

到目前为止，您可能已经注意到大多数容器对象都可以使用 `for` 语句：

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

这种访问风格清晰、简洁又方便。 迭代器的使用非常普遍并使得 Python 成为一个统一的整体。 在幕后，`for` 语句会调用容器对象中的 `iter()` 。 该函数返回一个定义了 `next()` 方法的迭代器对象，该方法将逐一访问容器中的元素。 当元素用尽时， `next()` 将引发 `StopIteration` 异常来通知终止 `for` 循环。 你可以使用 `next()` 内置函数来调用 `next()` 方法；这个例子显示了它的运作方式：

```
>>> 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 <module>
    next(it)
StopIteration
```

看过迭代器协议的幕后机制，给你的类添加迭代器行为就很容易了。 定义一个 `iter()` 方法来返回一个带有 `next()` 方法的对象。 如果类已定义了 `next()` ，则 `iter()` 可以简单地返回 `self` ：

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
```

```
6.
7.      def __iter__(self):
8.          return self
9.
10.     def __next__(self):
11.         if self.index == 0:
12.             raise StopIteration
13.         self.index = self.index - 1
14.         return self.data[self.index]
```

```
1.  >>> rev = Reverse('spam')
2.  >>> iter(rev)
3.  <__main__.Reverse object at 0x00A1DB50>
4.  >>> for char in rev:
5.  ...     print(char)
6.  ...
7.  m
8.  a
9.  p
10. s
```

9.9. 生成器

Generator 是一个用于创建迭代器的简单而强大的工具。 它们的写法类似标准的函数，但当它们要返回数据时会使用 `yield` 语句。 每次对生成器调用 `next()` 时，它会从上次离开位置恢复执行（它会记住上次执行语句时的所有数据值）。 显示如何非常容易地创建生成器的示例如下：

```
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
```

可以用生成器来完成的操作同样可以用前一节所描述的基于类的迭代器来完成。但生成器的写法更为紧凑，因为它会自动创建 `iter()` 和 `next()` 方法。

另一个关键特性在于局部变量和执行状态会在每次调用之间自动保存。 这使得该函数相比使用 `self.index` 和 `self.data` 这种实例变量的方式更易编写且更为清晰。

除了会自动创建方法和保存程序状态，当生成器终结时，它们还会自动引发 `StopIteration`。 这些特性结合在一起，使得创建迭代器能与编写常规函数一样容易。

9.10. 生成器表达式

某些简单的生成器可以写成简洁的表达式代码，所用语法类似列表推导式，将外层为圆括号而非方括号。 这种表达式被设计用于生成器将立即被外层函数所使用的情况。 生成器表达式相比完整的生成器更紧凑但较不灵活，相比等效的列表推导式则更为节省内存。

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

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

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

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

脚注

- 1
- 存在一个例外。 模块对象有一个秘密的只读属性 `dict`，它返回用于实现模块命名空间的字典；`dict` 是属性但不是全局名称。 显然，使用这个将违反命名空间实现的抽象，应当仅被用于事后调试器之类的场合。

10. 标准库简介

10.1. 操作系统接口

`os` 模块提供了许多与操作系统交互的函数：

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python38'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

一定要使用 `import os` 而不是 `from os import *` 。这将避免内建的 `open()` 函数被 `os.open()` 隐式替换掉，它们的使用方式大不相同。

内置的 `dir()` 和 `help()` 函数可用作交互式辅助工具，用于处理大型模块，如 `os` ：

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

对于日常文件和目录管理任务， `shutil` 模块提供了更易于使用的更高级别的接口：

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

10.2. 文件通配符

`glob` 模块提供了一个在目录中使用通配符搜索创建文件列表的函数：

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

10.3. 命令行参数

通用实用程序脚本通常需要处理命令行参数。这些参数作为列表存储在 `sys` 模块的 *argv* 属性中。例如，以下输出来自在命令行运行 `python demo.py one two three`

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

`argparse` 模块提供了一种更复杂的机制来处理命令行参数。 以下脚本可提取一个或多个文件名，并可选择要显示的行数：

```
import argparse

parser = argparse.ArgumentParser(prog = 'top',
    description = 'Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

当在通过 `python top.py –lines=5 alpha.txt beta.txt` 在命令行运行时，该脚本会将 `args.lines` 设为 `5` 并将 `args.filenames` 设为 `['alpha.txt', 'beta.txt']` 。

10.4. 错误输出重定向和程序终止

`sys` 模块还具有 *stdin* ， *stdout* 和 *stderr* 的属性。后者对于发出警告和错误消息非常有用，即使在 *stdout* 被重定向后也可以看到它们：

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

终止脚本的最直接方法是使用 `sys.exit()` 。

10.5. 字符串模式匹配

`re` 模块为高级字符串处理提供正则表达式工具。对于复杂的匹配和操作，正则表达式提供简洁，优化的解决方案：

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

当只需要简单的功能时，首选字符串方法因为它们更容易阅读和调试：

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

10.6. 数学

`math` 模块提供对浮点数学的底层C库函数的访问：

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

`random` 模块提供了进行随机选择的工具：

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

`statistics` 模块计算数值数据的基本统计属性（均值，中位数，方差等）：

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

SciPy项目 <https://scipy.org> 有许多其他模块用于数值计算。

10.7. 互联网访问

有许多模块可用于访问互联网和处理互联网协议。其中两个最简单的 `urllib.request` 用于从URL检索数据，以及 `smtplib` 用于发送邮件：

```
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8')  # Decoding the binary data to text.
...         if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...             print(line)
...
<BR>Nov. 25, 09:43:32 PM EST

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

（请注意，第二个示例需要在localhost上运行的邮件服务器。）

10.8. 日期和时间

`datetime` 模块提供了以简单和复杂的方式操作日期和时间的类。虽然支持日期和时间算法，但实现的重点是有效的成员提取以进行输出格式化和操作。该模块还支持可感知时区的对象。

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

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

10.9. 数据压缩

常见的数据存档和压缩格式由模块直接支持，包括： `zlib` ， `gzip` ， `bz2` ， `lzma` ， `zipfile` 和 `tarfile` 。：

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

10.10. 性能测量

一些Python用户对了解同一问题的不同方法的相对性能产生了浓厚的兴趣。Python提供了一种可以立即回答这些问题的测量工具。

例如，元组封包和拆包功能相比传统的交换参数可能更具吸引力。 `timeit` 模块可以快速演示在运行效率方面一定的优势：

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

与 `timeit` 的精细粒度级别相反， `profile` 和 `pstats` 模块提供了用于在较大的代码块中识别时间关键部分的工具。

10.11. 质量控制

开发高质量软件的一种方法是在开发过程中为每个函数编写测试，并在开发过程中经常运行这些测试。

`doctest` 模块提供了一个工具，用于扫描模块并验证程序文档字符串中嵌入的测试。测试构造就像将典型调用及其结果剪切并粘贴到文档字符串一样简单。这通过向用户提供示例来改进文档，并且它允许doctest模块确保代码保持对文档的真实：

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

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

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

`unittest` 模块不像 `doctest` 模块那样易于使用，但它允许在一个单独的文件中维护更全面的测试集：

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

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

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

10.12. 自带电池

Python有“自带电池”的理念。通过其包的复杂和强大功能可以最好地看到这一点。例如：

- `xmlrpc.client` 和 `xmlrpc.server` 模块使远程过程调用实现了几乎无关紧要的任务。尽管有模块名称，但不需要直接了解或处理XML。
- `email` 包是一个用于管理电子邮件的库，包括MIME和其他：基于 **RFC 2822** 的邮件文档。与 `smtplib` 和 `poplib` 实际上发送和接收消息不同，电子邮件包具有完整的工具集，用于构建或解码复杂的消息结构（包括附件）以及实现互联网编码和标头协议。
- `json` 包为解析这种流行的数据交换格式提供了强大的支持。 `csv` 模块支持以逗号分隔值格式直接读取和写入文件，这些格式通常由数据库和电子表格支持。 XML处理由 `xml.etree.ElementTree` ， `xml.dom` 和 `xml.sax` 包支持。这些模块和软件包共同大大简化了Python应用程序和其他工具之间的数据交换。
- `sqlite3` 模块是SQLite数据库库的包装器，提供了一个可以使用稍微非标准的SQL语法更新和访问的持久数据库。
- 国际化由许多模块支持，包括 `gettext` ， `locale` ，以及 `codecs` 包。

11. 标准库简介 — 第二部分

第二部分涵盖了专业编程所需要的更高级的模块。这些模块很少用在小脚本中。

11.1. 格式化输出

`reprlib` 模块提供了一个定制化版本的 `repr()` 函数，用于缩略显示大型或深层嵌套的容器对象：

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

`pprint` 模块提供了更加复杂的打印控制，其输出的内置对象和用户自定义对象能够被解释器直接读取。当输出结果过长而需要折行时，“美化输出机制”会添加换行符和缩进，以更清楚地展示数据结构：

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

`textwrap` 模块能够格式化文本段落，以适应给定的屏幕宽度：

```
>>> 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` 模块处理与特定地域文化相关的数据格式。`locale` 模块的 `format` 函数包含一个 `grouping` 属性，可直接将数字格式化为带有组分隔符的样式：

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
```

```
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2. 模板

`string` 模块包含一个通用的 `Template` 类，具有适用于最终用户的简化语法。它允许用户在不更改应用逻辑的情况下定制自己的应用。

上述格式化操作是通过占位符实现的，占位符由 `$` 加上合法的 Python 标识符（只能包含字母、数字和下划线）构成。一旦使用花括号将占位符括起来，就可以在后面直接跟上更多的字母和数字而无需空格分割。 `$$` 将被转义成单个字符 `$`：

```
>>> 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.'
```

如果在字典或关键字参数中未提供某个占位符的值，那么 `substitute()` 方法将抛出 `KeyError` 。对于邮件合并类型的应用，用户提供的数据有可能是不完整的，此时使用 `safe_substitute()` 方法更加合适 —— 如果数据缺失，它会直接将占位符原样保留。

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

Template 的子类可以自定义定界符。例如，以下是某个照片浏览器的批量重命名功能，采用了百分号作为日期、照片序号和照片格式的占位符：

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

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

img_1074.jpg --> Ashley_0.jpg
```

```
16.  img_1076.jpg --> Ashley_1.jpg
17.  img_1077.jpg --> Ashley_2.jpg
```

模板的另一个应用是将程序逻辑与多样的格式化输出细节分离开来。这使得对 XML 文件、纯文本报表和 HTML 网络报表使用自定义模板成为可能。

11.3. 使用二进制数据记录格式

`struct` 模块提供了 `pack()` 和 `unpack()` 函数，用于处理不定长度的二进制记录格式。下面的例子展示了在不使用 `zipfile` 模块的情况下，如何循环遍历一个 ZIP 文件的所有头信息。Pack 代码 `"H"` 和 `"I"` 分别代表两字节和四字节无符号整数。 `"<"` 代表它们是标准尺寸的小尾型字节序：

```
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

线程是一种对于非顺序依赖的多个任务进行解耦的技术。多线程可以提高应用的响应效率，当接收用户输入的同时，保持其他任务在后台运行。一个有关的应用场景是，将 I/O 和计算运行在两个并行的线程中。

以下代码展示了高阶的 `threading` 模块如何在后台运行任务，且不影响主程序的继续运行：

```
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.')
```

多线程应用面临的主要挑战是，相互协调的多个线程之间需要共享数据或其他资源。为此，threading 模块提供了多个同步操作原语，包括线程锁、事件、条件变量和信号量。

尽管这些工具非常强大，但微小的设计错误却可以导致一些难以复现的问题。因此，实现多任务协作的首选方法是将对资源的所有请求集中到一个线程中，然后使用 `queue` 模块向该线程供应来自其他线程的请求。应用程序使用 `Queue` 对象进行线程间通信和协调，更易于设计，更易读，更可靠。

11.5. 日志

`logging` 模块提供功能齐全且灵活的日志记录系统。在最简单的情况下，日志消息被发送到文件或 `sys.stderr`

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

这会产生以下输出：

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

默认情况下，informational 和 debugging 消息被压制，输出会发送到标准错误流。其他输出选项包括将消息转发到电子邮件，数据报，套接字或 HTTP 服务器。新的过滤器可以根据消息优先级选择不同的路由方式： `DEBUG` ， `INFO` ， `WARNING` ， `ERROR` ，和 `CRITICAL` 。

日志系统可以直接从 Python 配置，也可以从用户配置文件加载，以便自定义日志记录而无需更改应用程序。

11.6. 弱引用

Python 会自动进行内存管理（对大多数对象进行引用计数并使用 garbage collection 来清除循环引用）。 当某个对象的最后一个引用被移除后不久就会释放其所占用的内存。

此方式对大多数应用来说都适用，但偶尔也必须在对象持续被其他对象所使用时跟踪它们。 不幸的是，跟踪它们将创建一个会令其永久化的引用。 `weakref` 模块提供的工具可以不必创建引用就能跟踪对象。 当对象不再需要时，它将自动从一个弱引用表中被移除，并为弱引用对象触发一个回调。 典型应用包括对创建开销较大的对象进行缓存：

```
>>> 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:/python38/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7. 用于操作列表的工具

许多对于数据结构的需求可以通过内置列表类型来满足。 但是，有时也会需要具有不同效费比的替代实现。

`array` 模块提供了一种 `array()` 对象，它类似于列表，但只能存储类型一致的数据且存储密集更高。 下面的例子演示了一个以两个字节为存储单元的无符号二进制数值的数组 (类型码为 `"H"`)，而对于普通列表来说，每个条目存储为标准 Python 的 int 对象通常要占用16 个字节：

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

`collections` 模块提供了一种 `deque()` 对象，它类似于列表，但从左端添加和弹出的速度较快，而在中间查找的速度较慢。 此种对象适用于实现队列和广度优先树搜索：

```
>>> 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)
```

在替代的列表实现以外，标准库也提供了其他工具，例如 `bisect` 模块具有用于操作排序列表的函数：

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

`heapq` 模块提供了基于常规列表来实现堆的函数。 最小值的条目总是保持在位

置零。 这对于需要重复访问最小元素而不希望运行完整列表排序的应用来说非常有用：

```
>>> 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. 十进制浮点运算

`decimal` 模块提供了一种 `Decimal` 数据类型用于十进制浮点运算。 相比内置的 `float` 二进制浮点实现，该类特别适用于

- 财务应用和其他需要精确十进制表示的用途，
- 控制精度，
- 控制四舍五入以满足法律或监管要求，
- 跟踪有效小数位，或
- 用户期望结果与手工完成的计算相匹配的应用程序。

例如，使用十进制浮点和二进制浮点数计算70美分手机和5%税的总费用，会产生的不同结果。如果结果四舍五入到最接近的分数差异会更大：

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

`Decimal` 表示的结果会保留尾部的零，并根据具有两个有效位的被乘数自动推出四个有效位。 `Decimal` 可以模拟手工运算来避免当二进制浮点数无法精确表示十进制数时会导致的问题。

精确表示特性使得 `Decimal` 类能够执行对于二进制浮点数来说不适用的模运算和相等性检测：

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

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

`decimal` 模块提供了运算所需要的足够精度：

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

12. 虚拟环境和包

12.1. 概述

Python应用程序通常会使用不在标准库内的软件包和模块。应用程序有时需要特定版本的库，因为应用程序可能需要修复特定的错误，或者可以使用库的过时版本的接口编写应用程序。

这意味着一个Python安装可能无法满足每个应用程序的要求。如果应用程序A需要特定模块的1.0版本但应用程序B需要2.0版本，则需求存在冲突，安装版本1.0或2.0将导致某一个应用程序无法运行。

这个问题的解决方案是创建一个 virtual environment，一个目录树，其中安装有特定Python版本，以及许多其他包。

然后，不同的应用将可以使用不同的虚拟环境。 要解决先前需求相冲突的例子，应用程序 A 可以拥有自己的 安装了 1.0 版本的虚拟环境，而应用程序 B 则拥有安装了 2.0 版本的另一个虚拟环境。 如果应用程序 B 要求将某个库升级到 3.0 版本，也不会影响应用程序 A 的环境。

12.2. 创建虚拟环境

用于创建和管理虚拟环境的模块称为 `venv` 。 `venv` 通常会安装你可用的最新版本的 Python。如果您的系统上有多个版本的 Python，您可以通过运行 `python3` 或您想要的任何版本来选择特定的Python版本。

要创建虚拟环境，请确定要放置它的目录，并将 `venv` 模块作为脚本运行目录路径：

```
1. python3 -m venv tutorial-env
```

如果它不存在，这将创建 `tutorial-env` 目录，并在其中创建包含Python解释器，标准库和各种支持文件的副本的目录。

虚拟环境的常用目录位置是 `.venv` 。 这个名称通常会令该目录在你的终端中保持隐藏，从而避免需要对所在目录进行额外解释的一般名称。 它还能防止与某些工具所支持的 `.env` 环境变量定义文件发生冲突。

创建虚拟环境后，您可以激活它。

在Windows上，运行：

```
1. tutorial-env\Scripts\activate.bat
```

在Unix或MacOS上，运行：

```
1. source tutorial-env/bin/activate
```

（这个脚本是为bash shell编写的。如果你使用 **csh** 或 **fish** shell，你应该改用 `activate.csh` 或 `activate.fish` 脚本。）

Activating the virtual environment will change your shell's prompt to show whatvirtual environment you're using, and modify the environment so that running `python` will get you that particular version and installation of Python.For example:

```
1. $ source ~/envs/tutorial-env/bin/activate
2. (tutorial-env) $ python
3. Python 3.5.1 (default, May  6 2016, 10:59:36)
4.   ...
5. >>> import sys
6. >>> sys.path
7. ['', '/usr/local/lib/python35.zip', ...,
8. '~/envs/tutorial-env/lib/python3.5/site-packages']
```

```
9. >>>
```

12.3. 使用pip管理包

你可以使用一个名为 **pip** 的程序来安装、升级和移除软件包。默认情况下 `pip` 将从 Python Package Index <https://pypi.org> 安装软件包。你可以在浏览器中访问 Python Package Index 或是使用 `pip` 受限的搜索功能：

```
(tutorial-env) $ pip search astronomy
skyfield               - Elegant astronomy for Python
gary                   - Galactic astronomy and gravitational dynamics.
novas                  - The United States Naval Observatory NOVAS astronomy
library
astroobs               - Provides astronomy ephemeris to plan telescope
observations
PyAstronomy            - A collection of astronomy related tools for Python.
...
```

`pip` 有许多子命令："search"、"install"、"uninstall"、"freeze"等等。（请参阅 安装 Python 模块 指南以了解 `pip` 的完整文档。）

您可以通过指定包的名称来安装最新版本的包：

```
(tutorial-env) $ pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

您还可以通过提供包名称后跟 `==` 和版本号来安装特定版本的包：

```
(tutorial-env) $ pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

如果你重新运行这个命令，`pip` 会注意到已经安装了所请求的版本并且什么都不做。您可以提供不同的版本号来获取该版本，或者您可以运行 `pip install --upgrade` 将软件包升级到最新版本：

```
(tutorial-env) $ pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`pip uninstall` 后跟一个或多个包名称将从虚拟环境中删除包。

`pip show` 将显示有关特定包的信息：

```
(tutorial-env) $ pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`pip list` 将显示虚拟环境中安装的所有软件包：

```
(tutorial-env) $ pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

*pip freeze`* 将生成一个类似的已安装包列表，但输出使用 `pip install` 期望的格式。一个常见的约定是将此列表放在 `requirements.txt` 文件中：

```
(tutorial-env) $ pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

然后可以将 `requirements.txt` 提交给版本控制并作为应用程序的一部分提供。然后用户可以使用 `install -r` 安装所有必需的包：

```
(tutorial-env) $ pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` 有更多选择。有关 `pip` 的完整文档，请参阅 安装 Python 模块 指南。当您编写一个包并希望在 Python 包索引中使它可用时，请参考 分发 Python 模块 指南。

13. 接下来？

阅读本教程可能会增强您对使用Python的兴趣 - 您应该热衷于应用Python来解决您的实际问题。你应该去哪里了解更多？

本教程是Python文档集的一部分。其他文档：

- Python 标准库：

您应该浏览本手册，该手册提供了有关标准库中的类型，功能和模块的完整（尽管简洁）参考资料。标准的Python发行版包含 很多 的附加代码。有些模块可以读取Unix邮箱，通过HTTP检索文档，生成随机数，解析命令行选项，编写CGI程序，压缩数据以及许多其他任务。浏览标准库参考可以了解更多可用的内容。

- 安装 Python 模块 解释了怎么安装由其他Python开发者编写的模块。
- Python 语言参考： Python的语法和语义的详细解释。尽管阅读完非常繁重，但作为语言本身的完整指南是有用的。

更多Python资源：

- https://www.python.org ：主要的Python网站。它包含代码，文档以及指向Web上与Python相关的页面的链接。该网站世界很多地区都有镜像，如欧洲，日本和澳大利亚；镜像可能比主站点更快，具体取决于您的地理位置。
- https://docs.python.org ：快速访问Python的文档。
- https://pypi.org: The Python Package Index，以前也被昵称为 Cheese Shop [1]，是可下载用户自制 Python 模块的索引。 当你要开始发布代码时，你可以在此处进行注册以便其他人能找到它。
- https://code.activestate.com/recipes/langs/python/ ：Python Cookbook是一个相当大的代码示例集，更多的模块和有用的脚本。特别值得注意的贡献收集在一本名为Python Cookbook（O'Reilly & Associates，ISBN 0-596-00797-3）的书中。
- http://www.pyvideo.org 从会议和用户组会议中收集与Python相关的视频的链接。
- https://scipy.org ：Ecientific Python项目包括用于快速阵列计算和操作的模块，以及用于诸如线性代数，傅里叶变换，非线性求解器，随机数分布，统计分析等的一系列包。

对于与Python相关的问题和问题报告，您可以发布到新闻组 *comp.lang.python* ，或者将它们发送到邮件列表python-list@python.org。新闻组和邮件列表是互通的，因此发布到一个地方将自动转

发给另一个。每天有数百个帖子，询问（和回答）问题，建议新功能，以及宣布新模块。邮件列表档案可在 https://mail.python.org/pipermail/ 上找到。

在发问之前，请务必查看以下列表 常见问题 （或简写为 FAQ）。常见问题包含了很多一次又一次问到的问题及答案，并且可能已经包含了您的问题的解决方案。

备注

- 1
- “Cheese Shop”是 Monty Python 的一个短剧：一位顾客来到一家奶酪商店，但无论他要哪种奶酪，店员都说没有货。

14. 交互式编辑和编辑历史

某些版本的 Python 解释器支持编辑当前输入行和编辑历史记录，类似 Korn shell 和 GNU Bash shell 的功能 。这个功能使用了 GNU Readline 来实现，一个支持多种编辑方式的库。这个库有它自己的文档，在这里我们就不重复说明了。

14.1. Tab 补全和编辑历史

在解释器启动的时候，补全变量和模块名的功能将 自动打开，以便在按下 `Tab` 键的时候调用补全函数。它会查看 Python 语句名称，当前局部变量和可用的模块名称。处理像 `string.a` 的表达式，它会求值在最后一个 `'.'` 之前的表达式，接着根据求值结果对象的属性给出补全建议。如果拥有 `getattr()` 方法的对象是表达式的一部分，注意这可能会执行程序定义的代码。默认配置下会把编辑历史记录保存在用户目录下名为 `.python_history` 的文件。在下一次 Python 解释器会话期间，编辑历史记录仍旧可用。

14.2. 默认交互式解释器的替代品

Python 解释器与早期版本的相比，向前迈进了一大步；无论怎样，还有些希望的功能：如果能在编辑连续行时建议缩进（解析器知道接下来是否需要缩进符号），那将很棒。补全机制可以使用解释器的符号表。有命令去检查（甚至建议）括号，引号以及其他符号是否匹配。

一个可选的增强型交互式解释器是 IPython，它已经存在了有一段时间，它具有 tab 补全，探索对象和高级历史记录管理功能。它还可以彻底定制并嵌入到其他应用程序中。另一个相似的增强型交互式环境是 bpython。

15. 浮点算术：争议和限制

浮点数在计算机硬件中表示为以 2 为基数（二进制）的小数。举例而言，十进制的小数

```
0.125
```

等于 1/10 + 2/100 + 5/1000 ，同理，二进制的小数

```
0.001
```

等于0/2 + 0/4 + 1/8。这两个小数具有相同的值，唯一真正的区别是第一个是以 10 为基数的小数表示法，第二个则是 2 为基数。

不幸的是，大多数的十进制小数都不能精确地表示为二进制小数。这导致在大多数情况下，你输入的十进制浮点数都只能近似地以二进制浮点数形式储存在计算机中。

用十进制来理解这个问题显得更加容易一些。考虑分数 1/3 。我们可以得到它在十进制下的一个近似值

```
0.3
```

或者，更近似的，：

```
0.33
```

或者，更近似的，：

```
0.333
```

以此类推。结果是无论你写下多少的数字，它都永远不会等于 1/3 ，只是更加更加地接近 1/3 。

同样的道理，无论你使用多少位以 2 为基数的数码，十进制的 0.1 都无法精确地表示为一个以 2 为基数的小数。 在以 2 为基数的情况下， 1/10 是一个无限循环小数

```
0.0001100110011001100110011001100110011001100110011...
```

在任何一个位置停下，你都只能得到一个近似值。因此，在今天的大部分架构上，浮点数都只能近似地使用二进制小数表示，对应分数的分子使用每 8 字节的前 53 位表示，分母则表示为 2 的幂次。在 1/10 这个例子中，相应的二进制分数

是 `3602879701896397 / 2 ** 55` ，它很接近 1/10 ，但并不是 1/10 。

大部分用户都不会意识到这个差异的存在，因为 Python 只会打印计算机中存储的二进制值的十进制近似值。在大部分计算机中，如果 Python 想把 0.1 的二进制对应的精确十进制打印出来，将会变成这样

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

这比大多数人认为有用的数字更多，因此Python通过显示舍入值来保持可管理的位数

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

牢记，即使输出的结果看起来好像就是 1/10 的精确值，实际储存的值只是最接近 1/10 的计算机可表示的二进制分数。

有趣的是，有许多不同的十进制数共享相同的最接近的近似二进制小数。例如，`0.1` 、 `0.10000000000000001` 、 `0.1000000000000000055511151231257827021181583404541015625` 全都近似于 `3602879701896397 / 2 ** 55` 。由于所有这些十进制值都具有相同的近似值，因此可以显示其中任何一个，同时仍然保留不变的 `eval(repr(x)) == x` 。

在历史上，Python 提示符和内置的 `repr()` 函数会选择具有 17 位有效数字的来显示，即 `0.10000000000000001` 。 从 Python 3.1 开始，Python（在大多数系统上）现在能够选择这些表示中最短的并简单地显示 `0.1` 。

请注意这种情况是二进制浮点数的本质特性：它不是 Python 的错误，也不是你代码中的错误。 你会在所有支持你的硬件中的浮点运算的语言中发现同样的情况（虽然某些语言在默认状态或所有输出模块下都不会 显示 这种差异）。

想要更美观的输出，你可能会希望使用字符串格式化来产生限定长度的有效位数：

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

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

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

必须重点了解的是，这在实际上只是一个假象：你只是将真正的机器码值进行了舍入操作再 显示 而已。

一个假象还可能导致另一个假象。 例如，由于这个 0.1 并非真正的 1/10，将

三个 0.1 的值相加也不一定能恰好得到 0.3:

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

而且，由于这个 0.1 无法精确表示 1/10 的值而这个 0.3 也无法精确表示 3/10 的值，使用 `round()` 函数进行预先舍入也是没用的:

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

虽然这些小数无法精确表示其所要代表的实际值， `round()` 函数还是可以用来“事后舍入”，使得实际的结果值可以做相互比较:

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

二进制浮点运算会造成许多这样的“意外”。 有关 "0.1" 的问题会在下面的“表示性错误”一节中更详细地描述。 请参阅 浮点数的危险性 一文了解有关其他常见意外现象的更详细介绍。

正如那篇文章的结尾所言，“对此问题并无简单的答案。” 但是也不必过于担心浮点数的问题！ Python 浮点运算中的错误是从浮点运算硬件继承而来，而在大多数机器上每次浮点运算得到的 2**53 数码位都会被作为 1 个整体来处理。 这对大多数任务来说都已足够，但你确实需要记住它并非十进制算术，且每次浮点运算都可能会导致新的舍入错误。

虽然病态的情况确实存在，但对于大多数正常的浮点运算使用来说，你只需简单地将最终显示的结果舍入为你期望的十进制数值即可得到你期望的结果。 `str()` 通常已足够，对于更精度的控制可参看 格式字符串语法 中 `str.format()` 方法的格式描述符。

对于需要精确十进制表示的使用场景，请尝试使用 `decimal` 模块，该模块实现了适合会计应用和高精度应用的十进制运算。

另一种形式的精确运算由 `fractions` 模块提供支持，该模块实现了基于有理数的算术运算（因此可以精确表示像 1/3 这样的数值）。

如果你是浮点运算的重度用户，你应该看一下数值运算 Python 包 NumPy 以及由 SciPy 项目所提供的许多其它数学和统计运算包。 参见 <https://scipy.org>。

Python 也提供了一些工具，可以在你真的 想要 知道一个浮点数精确值的少数情况下提供帮助。 例如 `float.as_integer_ratio()` 方法会将浮点数表示为一个分数:

```
>>> x = 3.14159
```

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

由于这是一个精确的比值，它可以被用来无损地重建原始值：

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

`float.hex()` 方法会以十六进制（以 16 为基数）来表示浮点数，同样能给出保存在你的计算机中的精确值：

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

这种精确的十六进制表示法可被用来精确地重建浮点值：

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

由于这种表示法是精确的，它适用于跨越不同版本（平台无关）的 Python 移植数值，以及与支持相同格式的其他语言（例如 Java 和 C99）交换数据.

另一个有用的工具是 `math.fsum()` 函数，它有助于减少求和过程中的精度损失。它会在数值被添加到总计值的时候跟踪“丢失的位”。 这可以很好地保持总计值的精确度， 使得错误不会积累到能影响结果总数的程度：

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

15.1. 表示性错误

本小节将详细解释 "0.1" 的例子，并说明你可以怎样亲自对此类情况进行精确分析。 假定前提是已基本熟悉二进制浮点表示法。

表示性错误 是指某些（其实是大多数）十进制小数无法以二进制（以 2 为基数的计数制）精确表示这一事实造成的错误。 这就是为什么 Python（或者 Perl、C、C++、Java、Fortran 以及许多其他语言）经常不会显示你所期待的精确十进制数值的主要原因。

为什么会这样？ 1/10 是无法用二进制小数精确表示的。 目前（2000年11月）几乎所有使用 IEEE-754 浮点运算标准的机器以及几乎所有系统平台都会将 Python 浮点数映射为 IEEE-754 “双精度类型”。 754 双精度类型包含 53 位精度，因此在输入时，计算会尽量将 0.1 转换为以 *J/2**N* 形式所能表示的

最接近分数，其中 *J* 为恰好包含 53 个二进制位的整数。 重新将

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

写为

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

并且由于 *J* 恰好有 53 位（即 `>= 252` 但 `< 2 53`），*N* 的最佳值为 56：

```
1. >>> 2**52 <=  2**56 // 10  < 2**53
2. True
```

也就是说，56 是唯一的 *N* 值能令 *J* 恰好有 53 位。 这样 *J* 的最佳可能值就是经过舍入的商：

```
1. >>> q, r = divmod(2**56, 10)
2. >>> r
3. 6
```

由于余数超过 10 的一半，最佳近似值可通过四舍五入获得：

```
1. >>> q+1
2. 7205759403792794
```

这样在 754 双精度下 1/10 的最佳近似值为：

```
1. 7205759403792794 / 2 ** 56
```

分子和分母都除以二则结果小数为：

```
1. 3602879701896397 / 2 ** 55
```

请注意由于我们做了向上舍入，这个结果实际上略大于 1/10；如果我们没有向上舍入，则商将会略小于 1/10。 但无论如何它都不会是 精确的 1/10！

因此计算永远不会“看到”1/10：它实际看到的就是上面所给出的小数，它所能达到的最佳 754 双精度近似值：

```
1. >>> 0.1 * 2 ** 55
2. 3602879701896397.0
```

如果我们将该小数乘以 10**55，我们可以看到该值输出为 55 位的十进制数：

```
1. >>> 3602879701896397 * 10 ** 55 // 2 ** 55
```

```
2. 1000000000000000055511151231257827021181583404541015625
```

这意味着存储在计算机中的确切数值等于十进制数值 0.1000000000000000055511151231257827021181583404541015625。许多语言（包括较旧版本的 Python）都不会显示这个完整的十进制数值，而是将结果舍入为 17 位有效数字：

```
1. >>> format(0.1, '.17f')
2. '0.10000000000000001'
```

`fractions` 和 `decimal` 模块可令进行此类计算更加容易：

```
1. >>> from decimal import Decimal
2. >>> from fractions import Fraction
3.
4. >>> Fraction.from_float(0.1)
5. Fraction(3602879701896397, 36028797018963968)
6.
7. >>> (0.1).as_integer_ratio()
8. (3602879701896397, 36028797018963968)
9.
10. >>> Decimal.from_float(0.1)
11. Decimal('0.1000000000000000055511151231257827021181583404541015625')
12.
13. >>> format(Decimal.from_float(0.1), '.17')
14. '0.10000000000000001'
```

16. 附录

16.1. 交互模式

16.1.1. 错误处理

当发生错误时，解释器会打印错误信息和错误堆栈。在交互模式下，将返回到主命令提示符；如果输入内容来自文件，在打印错误堆栈之后，程序会以非零状态退出。（这里所说的错误不包括 `try` 语句中由 `except` 所捕获的异常。）有些错误是无条件致命的，会导致程序以非零状态退出；比如内部逻辑矛盾或内存耗尽。所有错误信息都会被写入标准错误流；而命令的正常输出则被写入标准输出流。

将中断字符（通常为 `Control-C` 或 `Delete` ）键入主要或辅助提示会取消输入并返回主提示符。 `1` 在执行命令时键入中断引发的 `KeyboardInterrupt` 异常，可以由 `try` 语句处理。

16.1.2. 可执行的Python脚本

在BSD等类Unix系统上，Python脚本可以直接执行，就像shell脚本一样，第一行添加：

```
1.  #!/usr/bin/env python3.5
```

（假设解释器位于用户的 `PATH` ）脚本的开头，并将文件设置为可执行。 `#!` 必须是文件的前两个字符。在某些平台上，第一行必须以Unix样式的行结尾（ `'\n'` ）结束，而不是以Windows（ `'\r\n'` ）行结尾。请注意，散列或磅字符 `'#'` 在Python中代表注释开始。

可以使用 **chmod** 命令为脚本提供可执行模式或权限。

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

在Windows系统上，没有“可执行模式”的概念。 Python安装程序自动将 `.py` 文件与 `python.exe` 相关联，这样双击Python文件就会将其作为脚本运行。扩展也可以是 `.pyw` ，在这种情况下，会隐藏通常出现的控制台窗口。

16.1.3. 交互式启动文件

当您以交互方式使用Python时，每次启动解释器时都会执行一些标准命令，这通常很方便。您可以通过将名为 `PYTHONSTARTUP` 的环境变量设置为包含启动命令的

文件名来实现。这类似于Unix shell的 `.profile` 功能。

This file is only read in interactive sessions, not when Python reads commandsfrom a script, and not when `/dev/tty` is given as the explicit source ofcommands (which otherwise behaves like an interactive session). It is executedin the same namespace where interactive commands are executed, so that objectsthat it defines or imports can be used without qualification in the interactivesession. You can also change the prompts `sys.ps1` and `sys.ps2` in thisfile.

如果你想从当前目录中读取一个额外的启动文件，你可以使用像 `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())` 这样的代码在全局启动文件中对它进行编程。如果要在脚本中使用启动文件，则必须在脚本中显式执行此操作：

```
1. import os
2. filename = os.environ.get('PYTHONSTARTUP')
3. if filename and os.path.isfile(filename):
4.     with open(filename) as fobj:
5.         startup_file = fobj.read()
6.     exec(startup_file)
```

16.1.4. 定制模块

Python提供了两个钩子来让你自定义它： `sitecustomize` 和 `usercustomize` 。要查看其工作原理，首先需要找到用户site-packages目录的位置。启动Python并运行此代码：

```
1. >>> import site
2. >>> site.getusersitepackages()
3. '/home/user/.local/lib/python3.5/site-packages'
```

现在，您可以在该目录中创建一个名为 `usercustomize.py` 的文件，并将所需内容放入其中。它会影响Python的每次启动，除非它以 `-s` 选项启动，以禁用自动导入。

`sitecustomize` 以相同的方式工作，但通常由计算机管理员在全局 site-packages 目录中创建，并在 `usercustomize` 之前被导入。有关详情请参阅 `site` 模块的文档。

脚注

- 1
- GNU Readline 包的问题可能会阻止这种情况。