Python Tutorial

发布 *3.12.1*

Guido van Rossum and the Python development team

十二月 12, 2023

Python Software Foundation
Email: docs@python.org

Contents

1 课前甜点 **3**

2 Python 解释器 **5**
2.1 调用解释器 . . . 5
2.1.1 传入参数 . . . 6
2.1.2 交互模式 . . . 6
2.2 解释器的运行环境 . . . 6
2.2.1 源文件的字符编码 . . . 6

3 Python 速览 **9**
3.1 Python 用作计算器 . . . 9
3.1.1 数字 . . . 9
3.1.2 文本 . . . 11
3.1.3 列表 . . . 14
3.2 走向编程的第一步 . . . 15

4 更多控制流工具 **17**
4.1 `if` 语句 . . . 17
4.2 `for` 语句 . . . 18
4.3 `range()` 函数 . . . 18
4.4 循环中的 `break`、`continue` 语句及 `else` 子句 . . . 19
4.5 `pass` 语句 . . . 20
4.6 `match` 语句 . . . 20
4.7 定义函数 . . . 23
4.8 函数定义详解 . . . 24
4.8.1 默认值参数 . . . 24
4.8.2 关键字参数 . . . 25
4.8.3 特殊参数 . . . 26
4.8.4 任意实参列表 . . . 29
4.8.5 解包实参列表 . . . 29
4.8.6 Lambda 表达式 . . . 30
4.8.7 文档字符串 . . . 30
4.8.8 函数注解 . . . 31
4.9 小插曲：编码风格 . . . 31

5 数据结构 **33**
5.1 列表详解 . . . 33
5.1.1 用列表实现堆栈 . . . 34
5.1.2 用列表实现队列 . . . 35
5.1.3 列表推导式 . . . 35
5.1.4 嵌套的列表推导式 . . . 36

5.2 del 语句 . . . 37
5.3 元组和序列 . . . 38
5.4 集合 . . . 39
5.5 字典 . . . 39
5.6 循环的技巧 . . . 40
5.7 深入条件控制 . . . 42
5.8 序列和其他类型的比较 . . . 42

6 模块 **45**
6.1 模块详解 . . . 46
6.1.1 以脚本方式执行模块 . . . 47
6.1.2 模块搜索路径 . . . 47
6.1.3 “已编译的” Python 文件 . . . 48
6.2 标准模块 . . . 48
6.3 dir() 函数 . . . 49
6.4 包 . . . 50
6.4.1 从包中导入 * . . . 51
6.4.2 相对导入 . . . 52
6.4.3 多目录中的包 . . . 52

7 输入与输出 **53**
7.1 更复杂的输出格式 . . . 53
7.1.1 格式化字符串字面值 . . . 54
7.1.2 字符串 format() 方法 . . . 55
7.1.3 手动格式化字符串 . . . 56
7.1.4 旧式字符串格式化方法 . . . 56
7.2 读写文件 . . . 57
7.2.1 文件对象的方法 . . . 57
7.2.2 使用 json 保存结构化数据 . . . 59

8 错误和异常 **61**
8.1 语法错误 . . . 61
8.2 异常 . . . 61
8.3 异常的处理 . . . 62
8.4 触发异常 . . . 64
8.5 异常链 . . . 65
8.6 用户自定义异常 . . . 66
8.7 定义清理操作 . . . 66
8.8 预定义的清理操作 . . . 67
8.9 引发和处理多个不相关的异常 . . . 67
8.10 用注释细化异常情况 . . . 69

9 类 **71**
9.1 名称和对象 . . . 71
9.2 Python 作用域和命名空间 . . . 72
9.2.1 作用域和命名空间示例 . . . 73
9.3 初探类 . . . 73
9.3.1 类定义语法 . . . 73
9.3.2 Class 对象 . . . 74
9.3.3 实例对象 . . . 75
9.3.4 方法对象 . . . 75
9.3.5 类和实例变量 . . . 76
9.4 补充说明 . . . 77
9.5 继承 . . . 78
9.5.1 多重继承 . . . 78
9.6 私有变量 . . . 79
9.7 杂项说明 . . . 80
9.8 迭代器 . . . 80
9.9 生成器 . . . 81

9.10 生成器表达式 . . . 82

10 标准库简介 83
10.1 操作系统接口 . . . 83
10.2 文件通配符 . . . 84
10.3 命令行参数 . . . 84
10.4 错误输出重定向和程序终止 . . . 84
10.5 字符串模式匹配 . . . 85
10.6 数学 . . . 85
10.7 互联网访问 . . . 86
10.8 日期和时间 . . . 86
10.9 数据压缩 . . . 86
10.10 性能测量 . . . 87
10.11 质量控制 . . . 87
10.12 自带电池 . . . 88

11 标准库简介——第二部分 89
11.1 格式化输出 . . . 89
11.2 模板 . . . 90
11.3 使用二进制数据记录格式 . . . 91
11.4 多线程 . . . 91
11.5 日志记录 . . . 92
11.6 弱引用 . . . 92
11.7 用于操作列表的工具 . . . 93
11.8 十进制浮点运算 . . . 94

12 虚拟环境和包 95
12.1 概述 . . . 95
12.2 创建虚拟环境 . . . 95
12.3 使用 pip 管理包 . . . 96

13 接下来？ 99

14 交互式编辑和编辑历史 101
14.1 Tab 补全和编辑历史 . . . 101
14.2 默认交互式解释器的替代品 . . . 101

15 浮点算术：争议和限制 103
15.1 表示性错误 . . . 106

16 附录 109
16.1 交互模式 . . . 109
16.1.1 错误处理 . . . 109
16.1.2 可执行的 Python 脚本 . . . 109
16.1.3 交互式启动文件 . . . 110
16.1.4 定制模块 . . . 110

A 术语对照表 111

B 文档说明 123
B.1 Python 文档的贡献者 . . . 123

C 历史和许可证 125
C.1 该软件的历史 . . . 125
C.2 获取或以其他方式使用 Python 的条款和条件 . . . 126
C.2.1 用于 PYTHON 3.12.1 的 PSF 许可协议 . . . 126
C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议 . . . 127
C.2.3 用于 PYTHON 1.6.1 的 CNRI 许可协议 . . . 128
C.2.4 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议 . . . 129

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.12.1 DOCUMENTATION . . . 129
C.3 收录软件的许可与鸣谢 . . . 129
C.3.1 Mersenne Twister . . . 130
C.3.2 套接字 . . . 130
C.3.3 异步套接字服务 . . . 131
C.3.4 Cookie 管理 . . . 131
C.3.5 执行追踪 . . . 132
C.3.6 UUencode 与 UUdecode 函数 . . . 132
C.3.7 XML 远程过程调用 . . . 133
C.3.8 test_epoll . . . 133
C.3.9 Select kqueue . . . 134
C.3.10 SipHash24 . . . 134
C.3.11 strtod 和 dtoa . . . 135
C.3.12 OpenSSL . . . 135
C.3.13 expat . . . 138
C.3.14 libffi . . . 139
C.3.15 zlib . . . 139
C.3.16 cfuhash . . . 140
C.3.17 libmpdec . . . 140
C.3.18 W3C C14N 测试套件 . . . 141
C.3.19 audioop . . . 142

D 版权所有 **143**

索引 **145**

Python 是一门易于学习、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 官网（https://www.python.org/）上免费提供了 Python 解释器和扩展的标准库，包括源码和适用于各操作系统的机器码形式，并可自由地分发。Python 官网还包含许多免费的第三方 Python 模块、程序和工具发布包及文档链接。

Python 解释器易于扩展，使用 C 或 C++（或其他 C 能调用的语言）即可为 Python 扩展新功能和数据类型。Python 也可用作定制软件中的扩展程序语言。

本教程只是简单介绍了 Python 语言概念和功能。读者在阅读本教程时最好使用 Python 解释器以便随时动手练习。本教程中的所有示例都是相互独立的并可离线阅读。

标准库与模块的内容详见 library-index。reference-index 是更正规的语言定义。如要编写 C 或 C++ 扩展请参考 extending-index 和 c-api-index。此外，深入讲解 Python 的书籍也有很多。

本教程对每一个功能的介绍并不完整，甚至没有涉及全部常用功能，只是介绍了 Python 中最值得学习的功能，旨在让读者快速感受一下 Python 的特色。学完本教程的读者可以阅读和编写 Python 模块和程序，也可以继续学习 library-index。

强烈推荐阅读术语对照表。

CHAPTER 1

课前甜点

如果您的工作主要是用电脑完成的，总有一天您会想能不能自动执行一些任务。比如，对大量文本文件执行查找、替换操作；利用复杂的规则重命名、重排序一堆照片文件；也可能您想编写一个小型数据库、或开发专用的图形界面应用，甚至是开发一个简单的游戏。

作为一名专业软件开发人员，您可能要处理 C/C++/Java 库，但编码、编译、测试、再编译这些开发流程太慢了；也许您正在给这些库开发测试套件，但总觉得这项工作真是枯燥乏味。又或许，您开发了个使用扩展语言的软件，却不想为这个软件专门设计一种新语言。

那么，Python 正好能满足您的需要。

你可以针对这些任务编写 Unix shell 脚本或 Windows 批处理文件，但 shell 脚本擅长的是移动文件和改变文本数据，而不适合编写 GUI 应用或游戏。你可以编写 C/C++/Java 程序，但即使只完成一个初始版程序也需要耗费很长的开发时间。Python 则更为简单易用，同时支持 Windows, macOS 和 Unix 操作系统，并能帮助你更快速地完成工作。

Python 虽然简单易用，但它可是真正的编程语言，提供了大量的数据结构，也支持开发大型程序，远超 shell 脚本或批处理文件；Python 提供的错误检查比 C 还多；作为一种“非常高级的语言”，它内置了灵活的数组与字典等高级数据类型。正因为配备了更通用的数据类型，Python 比 Awk，甚至 Perl 能解决更多问题，而且，很多时候，Python 比这些语言更简单。

Python 支持把程序分割为模块，以便在其他 Python 程序中复用。它还内置了大量标准模块，作为开发程序的基础——您还可以把这些模块当作学习 Python 编程的实例。这些模块包括 I/O、系统调用、套接字，甚至还包括 Tk 图形用户界面工作套件。

Python 是一种解释型语言，不需要编译和链接，可以节省大量开发时间。它的解释器实现了交互式操作，轻而易举地就能试用各种语言功能，编写临时程序，或在自底向上的程序开发中测试功能。同时，它还是一个超好用的计算器。

Python 程序简洁、易读，通常比实现同种功能的 C、C++、Java 代码短很多，原因如下：

- 高级数据类型允许在单一语句中表述复杂操作；
- 使用缩进，而不是括号实现代码块分组；
- 无需预声明变量或参数。

Python “可以扩展”：会开发 C 语言程序，就能快速上手为解释器增加新的内置函数或模块，不论是让核心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，硬件图形库）。只要下点功夫，就能把 Python 解释器和用 C 开发的应用链接在一起，用它来扩展和控制该应用。

顺便提一句，本语言的命名源自 BBC 的“Monty Python 飞行马戏团”，与爬行动物无关（Python 原义为“蟒蛇”）。欢迎大家在文档中引用 Monty Python 小品短篇集，多多益善！

现在，您已经对 Python 跃跃欲试，想深入了解一些细节了吧。要知道，学习语言的最佳方式是上手实践，建议您边阅读本教程，边在 Python 解释器中练习。

下一章介绍解释器的用法。这部分内容有些单调乏味，但对上手实践后面的例子来说却至关重要。

本教程的其他部分将利用各种示例，介绍 Python 语言、系统的功能，开始只是简单的表达式、语句和数据类型，然后是函数、模块，最后，介绍一些高级概念，如，异常、用户定义的类等功能。

CHAPTER 2

Python 解释器

2.1 调用解释器

Python 解释器在可用的机器上通常被安装为 `/usr/local/bin/python3.12`；将 `/usr/local/bin` 加入你的 Unix shell 的搜索路径就可以通过输入以下命令来启动它:

```
python3.12
```

这样，就可以在 shell 中运行 Python 了[1] 。因为可以选择安装目录，解释器也有可能安装在别的位置；如果还不明白，就去问问身边的 Python 大神或系统管理员。（例如，常见备选路径还有 `/usr/local/python`。）

在 Windows 机器上当你从 Microsoft Store 安装 Python 之后，`python3.12` 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。请参阅 setting-envvars 了解其他启动 Python 的方式。

在主提示符中，输入文件结束符（Unix 里是 `Control-D`，Windows 里是 `Control-Z`），就会退出解释器，退出状态码为 0。如果不能退出，还可以输入这个命令：`quit()`。

在支持 GNU Readline 库的系统中，解释器的行编辑功能包括交互式编辑、历史替换、代码补全等。检测是否支持命令行编辑最快速的方式是，在首次出现 Python 提示符时，输入 `Control-P`。听到 “哔” 提示音，说明支持行编辑；请参阅附录*交互式编辑和编辑历史*，了解功能键。如果没有反应，或回显了 `^P`，则说明不支持行编辑；只能用退格键删除当前行的字符。

解释器的操作方式类似 Unix Shell：用与 tty 设备关联的标准输入调用时，可以交互式地读取和执行命令；以文件名参数，或标准输入文件调用时，则读取并执行文件中的*脚本*。

另一种启动解释器的方式是 `python -c command [arg] ...`，这将执行 *command* 中的语句，相当于 shell 的 `-c` 选项。由于 Python 语句经常包含空格或其他会被 shell 特殊对待的字符，通常建议用引号将整个 *command* 括起来。

Python 模块也可以当作脚本使用。输入：`python -m module [arg] ...`，会执行 *module* 的源文件，这跟在命令行把路径写全了一样。

在交互模式下运行脚本文件，只要在脚本名称参数前，加上选项 `-i` 就可以了。

命令行的所有选项详见 using-on-general。

[1] Unix 系统中，为了不与同时安装的 Python 2.x 冲突，Python 3.x 解释器默认安装的执行文件名不是 `python`。

2.1.1 传入参数

解释器读取命令行参数，把脚本名与其他参数转化为字符串列表存到 `sys` 模块的 `argv` 变量里。执行 `import sys`，可以导入这个模块，并访问该列表。该列表最少有一个元素；未给定输入参数时，`sys.argv[0]` 是空字符串。给定脚本名是 `'-'`（标准输入）时，`sys.argv[0]` 是 `'-'`。使用 `-c` *command* 时，`sys.argv[0]` 是 `'-c'`。如果使用选项 `-m` *module*，`sys.argv[0]` 就是包含目录的模块全名。解释器不处理 `-c` *command* 或 `-m` *module* 之后的选项，而是直接留在 `sys.argv` 中由命令或模块来处理。

2.1.2 交互模式

在终端（tty）输入并执行指令时，解释器在 *交互模式*（*interactive mode*）中运行。在这种模式中，会显示 *主提示符*，提示输入下一条指令，主提示符通常用三个大于号（`>>>`）表示；输入连续行时，显示 *次要提示符*，默认是三个点（`...`）。进入解释器时，首先显示欢迎信息、版本信息、版权声明，然后才是提示符：

```
$ python3.12
Python 3.12 (default, April 4 2022, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

输入多行架构的语句时，要用连续行。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!
```

交互模式的内容详见*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 源码文件的编码是 UTF-8。这种编码支持世界上大多数语言的字符，可以用于字符串字面值、变量、函数名及注释——尽管标准库只用常规的 ASCII 字符作为变量名或函数名，可移植代码都应遵守此约定。要正确显示这些字符，编辑器必须能识别 UTF-8 编码，而且必须使用支持文件中所有字符的字体。

如果不使用默认编码，则要声明文件的编码，文件的 *第一行要写成特殊注释*。句法如下：

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

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

比如，声明使用 Windows-1252 编码，源码文件要写成：

```
# -*- coding: cp1252 -*-
```

第一行的规则也有一种例外情况，源码以*UNIX "shebang" 行* 开头。此时，编码声明要写在文件的第二行。例如：

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

备注

CHAPTER 3

Python 速览

下面的例子以是否显示提示符（»> 与...）区分输入与输出：输入例子中的代码时，要键入以提示符开头的行中提示符后的所有内容；未以提示符开头的行是解释器的输出。注意，例子中的某行出现的第二个提示符是用来结束多行命令的，此时，要键入一个空白行。

本手册中的许多例子，甚至交互式命令都包含注释。Python 注释以 # 开头，直到该物理行结束。注释可以在行开头，或空白符与代码之后，但不能在字符串里面。字符串中的 # 号就是 # 号。注释用于阐明代码，Python 不解释注释，键入例子时，可以不输入注释。

示例如下：

```
# 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 用作计算器

现在，尝试一些简单的 Python 命令。启动解释器，等待主提示符（>>> ）出现。

3.1.1 数字

解释器像一个简单的计算器：你可以输入一个表达式，它将给出结果值。表达式语法很直观：运算符 +，-，* 和 / 可被用来执行算术运算；圆括号 (()) 可被用来进行分组。例如:

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

整数（如，2、4、20 ）的类型是 int，带小数（如，5.0、1.6 ）的类型是 float。本教程后半部分将介绍更多数字类型。

除法运算 (/) 总是返回浮点数。如果要做*floor division* 得到一个整数结果你可以使用 // 运算符；要计算余数你可以使用 %:

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

Python 用 ** 运算符计算乘方[1]：

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

等号（=）用于给变量赋值。赋值后，下一个交互提示符的位置不显示任何结果：

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

如果变量未定义（即，未赋值），使用该变量会提示错误：

```
>>> 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 全面支持浮点数；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
```

交互模式下，上次输出的表达式会赋给变量 _。把 Python 当作计算器时，用该变量实现下一步计算更简单，例如：

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

最好把该变量当作只读类型。不要为它显式赋值，否则会创建一个同名独立局部变量，该变量会用它的魔法行为屏蔽内置变量。

除了 int 和 float，Python 还支持其他数字类型，例如 Decimal 或 Fraction。Python 还内置支持 复数，后缀 j 或 J 用于表示虚数（例如 3+5j ）。

[1] ** 比 - 的优先级更高，所以 -3**2 会被解释成 -(3**2) ，因此，结果是 -9。要避免这个问题，并且得到 9，可以用 (-3)**2。

3.1.2 文本

除了数字 Python 还可以操作文本（由 str 类型表示，称为 “字符串”）。这包括字符"!", 单词"rabbit", 名称"Paris", 句子"Got your back." 等等. "Yay! :)"。它们可以用成对的单引号 ('...') 或双引号 ("...") 来标示，结果完全相同[2]。

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!"  # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975'  # digits and numerals enclosed in quotes are also strings
'1975'
```

要标示引号本身，我们需要对它进行 “转义”，即在前面加一个 \。或者，我们也可以使用不同类型的引号:

```
>>> '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 Python shell 中，字符串定义和输出字符串看起来可能不同。print() 函数会略去标示用的引号，并打印经过转义的特殊字符，产生更为易读的输出:

```
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s)  # with print(), special characters are interpreted, so \n produces␣
↪new line
First line.
Second line.
```

如果不希望前置 \ 的字符转义成特殊字符，可以使用 原始字符串，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
```

原始字符串还有一个微妙的限制：一个原始字符串不能以奇数个 \ 字符结束；请参阅 此 FAQ 条目了解更多信息及绕过的办法。

字符串字面值可以包含多行。一种实现方式是使用三重引号："""..."""或'''...'''。字符串中将自动包括行结束符，但也可以在换行的地方添加一个 \ 来避免此情况。参见以下示例:

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

输出如下（请注意开始的换行符没有被包括在内）:

[2] 与其他语言不同，特殊字符如 \n 在单引号（'...'）和双引号（"..."）里的意义一样。这两种引号唯一的区别是，不需要在单引号里转义双引号 "（但此时必须把单引号转义成 \'），反之亦然。

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

字符串可以用 + 合并（粘到一起），也可以用 * 重复：

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

相邻的两个或多个 *字符串字面值*（引号标注的字符）会自动合并：

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

拼接分隔开的长字符串时，这个功能特别实用：

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

这项功能只能用于两个字面值，不能用于变量或表达式：

```
>>> 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
```

合并多个变量，或合并变量与字面值，要用 +：

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

字符串支持 *索引*（下标访问），第一个字符的索引是 0。单字符没有专用的类型，就是长度为一的字符串：

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

索引还支持负数，用负数索引时，从右边开始计数：

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

注意，-0 和 0 一样，因此，负数索引从 -1 开始。

除了索引操作，还支持 *切片*。索引用来获取单个字符，而 *切片* 允许你获取子字符串：

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

切片索引的默认值很有用；省略开始索引时，默认值为 0，省略结束索引时，默认为到字符串的结尾：

```
>>> 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'
```

注意，输出结果包含切片开始，但不包含切片结束。因此，`s[:i] + s[i:]` 总是等于 `s`：

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

还可以这样理解切片，索引指向的是字符 *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* ，*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
```

第一行数字是字符串中索引 0...6 的位置，第二行数字是对应的负数索引位置。*i* 到 *j* 的切片由 *i* 和 *j* 之间所有对应的字符组成。

对于使用非负索引的切片，如果两个索引都不越界，切片长度就是起止索引之差。例如，`word[1:3]` 的长度是 2。

索引越界会报错：

```
>>> 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，切片会自动处理越界索引：

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

Python 字符串不能修改，是*immutable* 的。因此，为字符串中某个索引位置赋值会报错：

```
>>> 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
```

要生成不同的字符串，应新建一个字符串：

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

内置函数 `len()` 返回字符串的长度：

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

参见：

textseq 字符串是 *序列类型*，支持序列类型的各种操作。

string-methods 字符串支持很多变形与查找方法。

f-strings 内嵌表达式的字符串字面值。

formatstrings 使用 `str.format()` 格式化字符串。

old-string-formatting 这里详述了用 `%` 运算符格式化字符串的操作。

3.1.3 列表

Python 支持多种 *复合数据类型*，可将不同值组合在一起。最常用的 *列表*，是用方括号标注，逗号分隔的一组值。列表可以包含不同类型的元素，但一般情况下，各个元素的类型相同：

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

和字符串（及其他内置*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]
```

切片操作返回包含请求元素的新列表。以下切片操作会返回列表的 浅拷贝：

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

列表还支持合并操作：

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

与*immutable* 字符串不同，列表是*mutable* 类型，其内容可以改变：

```
>>> 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]
```

你也可以在通过使用 `list.append()` *方法*，在列表末尾添加新条目（我们将在后面介绍更多相关的方法）：

```
>>> 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]
```

为切片赋值可以改变列表大小，甚至清空整个列表：

```
>>> 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()` 也支持列表：

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

还可以嵌套列表（创建包含其他列表的列表），例如：

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

3.2 走向编程的第一步

当然，我们还能用 Python 完成比二加二更复杂的任务。例如，我们可以像下面这样写出 斐波那契数列 初始部分的子序列：

```
>>> # 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
```

(下页继续)

(续上页)

```
5
8
```

本例引入了几个新功能。

- 第一行中的 *多重赋值*：变量 a 和 b 同时获得新值 0 和 1 。最后一行又用了一次多重赋值，体现了，等号右边的所有表达式的值，都是在这一语句对任何变量赋新值之前求出来的——求值顺序为从左到右。
- while 循环只要条件（这里是 a < 10）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列表类型的值，事实上，任何序列都可以：长度非零就为真，空序列则为假。示例中的判断只是最简单的比较。比较操作符的写法和 C 语言一样：<（小于）、>（大于）、==（等于）、<=（小于等于）、>=（大于等于）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，得为每个缩进的行输入空格（或制表符）。使用文本编辑器可以实现更复杂的输入方式；所有像样的文本编辑器都支持自动缩进。交互式输入复合语句时，要在最后输入空白行表示完成（因为解析器不知道哪一行代码是代码块的最后一行）。注意，同一块语句的每一行的缩进相同。
- print() 函数输出给定参数的值。除了可以以单一的表达式作为参数（比如，前面的计算器的例子），它还能处理多个参数，包括浮点数与字符串。它输出的字符串不带引号，且各参数项之间会插入一个空格，这样可以实现更好的格式化操作：

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

关键字参数 *end* 可以取消输出后面的换行, 或用另一个字符串结尾：

```
>>> 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,
```

备注

CHAPTER 4

更多控制流工具

除了刚介绍的 `while` 语句，Python 还用了一些别的。我们将在本章中遇到它们。

4.1 `if` 语句

最让人耳熟能详的语句应当是 `if` 语句：

```
>>> 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
```

可有零个或多个 `elif` 部分，`else` 部分也是可选的。关键字'`elif`'是'else if'的缩写，用于避免过多的缩进。`if` ... `elif` ... `elif` ... 序列可以当作其它语言中 `switch` 或 `case` 语句的替代品。

如果是把一个值与多个常量进行比较，或者检查特定类型或属性，`match` 语句更有用。详见*match 语句*。

4.2 for 语句

Python 的 for 语句与 C 或 Pascal 中的不同。Python 的 for 语句不迭代算术递增数值（如 Pascal），或是给予用户定义迭代步骤和结束条件的能力（如 C），而是在列表或字符串等任意序列的元素上迭代，按它们在序列中出现的顺序。例如（这不是有意要暗指什么）：

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

很难正确地在迭代多项集的同时修改多项集的内容。更简单的方法是迭代多项集的副本或者创建新的多项集：

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# 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

内置函数 range() 用于生成等差数列：

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

生成的序列绝不会包括给定的终止值；range(10) 生成 10 个值——长度为 10 的序列的所有合法索引。range 可以不从 0 开始，且可以按给定的步长递增（即使是负数步长）：

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

要按索引迭代序列，可以组合使用 range() 和 len()：

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

不过大多数情况下 enumerate() 函数很方便，详见循环的技巧。

如果直接打印一个 range 会发生意想不到的事情：

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

range() 返回的对象在很多方面和列表的行为一样，但其实它和列表不一样。该对象只有在被迭代时才一个一个地返回所期望的列表项，并没有真正生成过一个含有全部项的列表，从而节省了空间。

这种对象称为可迭代对象*iterable*，适合作为需要获取一系列值的函数或程序构件的参数。for 语句就是这样的程序构件；以可迭代对象作为参数的函数例如 sum()：

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

之后我们会看到更多返回可迭代对象，或以可迭代对象作为参数的函数。在数据结构 这一章中，我们将讨论 list() 的更多细节。

4.4 循环中的 break、continue 语句及 else 子句

break 语句将跳出最近的一层 for 或 while 循环。

for 或 while 循环可以包括 else 子句。

在 for 循环中，else 子句会在循环成功结束最后一次迭代之后执行。

在 while 循环中，它会在循环条件变为假值后执行。

无论哪种循环，如果因为 break 而结束，那么 else 子句就 **不会**执行。

下面的搜索质数的 for 循环就是一个例子：

```
>>> 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
```

（没错，这段代码就是这么写。仔细看：else 子句属于 for 循环，**不属于** if 语句。）

else 子句用于循环时比起 if 语句的 else 子句，更像 try 语句的。try 语句的 else 子句在未发生异常时执行，循环的 else 子句则在未发生 break 时执行。try 语句和异常详见异常的处理。

continue 语句，同样借鉴自 C 语言，以执行循环的下一次迭代来继续：

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

4.5 pass 语句

pass 语句不执行任何动作。语法上需要一个语句，但程序毋需执行任何动作时，可以使用该语句。例如：

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

这常用于创建一个最小的类：

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

pass 还可用作函数或条件语句体的占位符，让你保持在更抽象的层次进行思考。pass 会被默默地忽略：

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

4.6 match 语句

match 语句接受一个表达式并把它的值与一个或多个 case 块给出的一系列模式进行比较。这表面上像 C、Java 或 JavaScript（以及许多其他程序设计语言）中的 switch 语句，但其实它更像 Rust 或 Haskell 中的模式匹配。只有第一个匹配的模式会被执行，并且它还可以提取值的组成部分（序列的元素或对象的属性）赋给变量。

最简单的形式是将一个主语值与一个或多个字面值进行比较：

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
```

（下页继续）

(续上页)

```
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

注意最后一个代码块：“变量名” _ 被作为 通配符并必定会匹配成功。如果没有 case 匹配成功，则不会执行任何分支。

你可以用 | （“或”）将多个字面值组合到一个模式中：

```
case 401 | 403 | 404:
    return "Not allowed"
```

形如解包赋值的模式可被用于绑定变量：

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

请仔细学习此代码！第一个模式有两个字面值，可视为前述字面值模式的扩展。接下来的两个模式结合了一个字面值和一个变量，变量 绑定了来自主语（point）的一个值。第四个模式捕获了两个值，使其在概念上与解包赋值 (x, y) = point 类似。

如果用类组织数据，可以用“类名后接一个参数列表”这种很像构造器的形式，把属性捕获到变量里：

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

一些内置类（如 dataclass）为属性提供了一个顺序，此时，可以使用位置参数。自定义类可通过在类中设置特殊属性 __match_args__，为属性指定其在模式中对应的位置。若设为 (”x”, ”y”)，则以下模式相互等价（且都把属性 y 绑定到变量 var）：

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

建议这样来阅读一个模式——通过将其视为赋值语句等号左边的一种扩展形式，来理解各个变量被设为何值。match 语句只会为单一的名称（如上面的 var）赋值，而不会赋值给带点号的名称（如 foo.bar）、

属性名（如上面的 x= 和 y=）和类名（是通过其后的"(...)" 来识别的，如上面的 Point）。

模式可以任意嵌套。举例来说，如果我们有一个由 Point 组成的列表，且 Point 添加了 __match_args__ 时，我们可以这样来匹配它：

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

我们可以为模式添加 if 作为守卫子句。如果守卫子句的值为假，那么 match 会继续尝试匹配下一个 case 块。注意是先将值捕获，再对守卫子句求值：

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

该语句的一些其它关键特性：

- 与解包赋值类似，元组和列表模式具有完全相同的含义并且实际上都能匹配任意序列，区别是它们不能匹配迭代器或字符串。
- 序列模式支持扩展解包：[x, y, *rest] 和 (x, y, *rest) 和相应的解包赋值做的事是一样的。接在 * 后的名称也可以为 _，所以 (x, y, *_) 匹配含至少两项的序列，而不必绑定剩余的项。
- 映射模式：{"bandwidth": b, "latency": l} 从字典中捕获 "bandwidth" 和 "latency" 的值。额外的键会被忽略，这一点与序列模式不同。**rest 这样的解包也支持。（但 **_ 将会是冗余的，故不允许使用。）
- 使用 as 关键字可以捕获子模式：

  ```
  case (Point(x1, y1), Point(x2, y2) as p2): ...
  ```

  将把输入中的第二个元素捕获为 p2 （只要输入是包含两个点的序列）
- 大多数字面值是按相等性比较的，但是单例对象 True、False 和 None 则是按 id 比较的。
- 模式可以使用具名常量。它们必须作为带点号的名称出现，以防止它们被解释为用于捕获的变量：

  ```
  from enum import Enum
  class Color(Enum):
      RED = 'red'
      GREEN = 'green'
      BLUE = 'blue'

  color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

  match color:
  ```

(下页继续)

(续上页)

```
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

更详细的说明和更多示例，可参阅以教程格式撰写的 **PEP 636**。

4.7 定义函数

下列代码创建一个可以输出限定数值内的斐波那契数列函数：

```
>>> 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
```

定义函数使用关键字 `def`，后跟函数名与括号内的形参列表。函数语句从下一行开始，并且必须缩进。

函数内的第一条语句是字符串时，该字符串就是文档字符串，也称为 *docstring*，详见文档字符串。利用文档字符串可以自动生成在线文档或打印版文档，还可以让开发者在浏览代码时直接查阅文档；Python 开发者最好养成在代码中加入文档字符串的好习惯。

函数在 执行时使用函数局部变量符号表，所有函数变量赋值都存在局部符号表中；引用变量时，首先，在局部符号表里查找变量，然后，是外层函数局部符号表，再是全局符号表，最后是内置名称符号表。因此，尽管可以引用全局变量和外层函数的变量，但最好不要在函数内直接赋值（除非是 `global` 语句定义的全局变量，或 `nonlocal` 语句定义的外层函数变量）。

在调用函数时会将实际参数（实参）引入到被调用函数的局部符号表中；因此，实参是使用 按值调用 来传递的（其中的 值始终是对象的 引用 而不是对象的值）。[1] 当一个函数调用另外一个函数时，会为该调用创建一个新的局部符号表。

函数定义在当前符号表中把函数名与函数对象关联在一起。解释器把函数名指向的对象作为用户自定义函数。还可以使用其他名称指向同一个函数对象，并访问访该函数：

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

`fib` 不返回值，因此，其他语言不把它当作函数，而是当作过程。事实上，没有 `return` 语句的函数也返回值，只不过这个值比较是 `None`（是一个内置名称）。一般来说，解释器不会输出单独的返回值 `None`，如需查看该值，可以使用 `print()`：

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

编写不直接输出斐波那契数列运算结果，而是返回运算结果列表的函数也非常简单：

[1] 实际上，对象引用调用这种说法更好，因为，传递的是可变对象时，调用者能发现被调者做出的任何更改（插入列表的元素）。

```
>>> 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]
```

本例也新引入了一些 Python 功能：

- return 语句返回函数的值。return 语句不带表达式参数时，返回 None。函数执行完毕退出也返回 None。
- 语句 result.append(a) 调用了列表对象 result 的 *方法*。方法是‘从属于’对象的函数，其名称为 obj.methodname，其中 obj 是某个对象（可以是一个表达式），methodname 是由对象的类型定义的方法名称。不同的类型定义了不同的方法。不同的类型的方法可以使用相同的名称而不会产生歧义。（使用 *类* 可以定义自己的对象类型和方法，参见*类*。）在示例中显示的方法 append() 是由列表对象定义的；它会在列表的末尾添加一个新元素。在本例中它等同于 result = result + [a]，但效率更高。

4.8 函数定义详解

函数定义支持可变数量的参数。这里列出三种可以组合使用的形式。

4.8.1 默认值参数

为参数指定默认值是非常有用的方式。调用函数时，可以使用比定义时更少的参数，例如：

```
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)
```

该函数可以用以下方式调用：

- 只给出必选实参：ask_ok('Do you really want to quit?')
- 给出一个可选实参：ask_ok('OK to overwrite the file?', 2)
- 给出所有实参：ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

本例还使用了关键字 in，用于确认序列中是否包含某个值。

默认值在 *定义作用域* 里的函数定义中求值，所以：

```
i = 5

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

i = 6
f()
```

上例输出的是 5。

重要警告：默认值只计算一次。默认值为列表、字典或类实例等可变对象时，会产生与该规则不同的结果。例如，下面的函数会累积后续调用时传递的参数：

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

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

输出结果如下：

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

不想在后续调用之间共享默认值时，应以如下方式编写函数：

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

4.8.2 关键字参数

kwarg=value 形式的*关键字参数*也可以用于调用函数。函数示例如下：

```
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, "!")
```

该函数接受一个必选参数（voltage）和三个可选参数（state, action 和 type）。该函数可用下列方式调用：

```
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
```

以下调用函数的方式都无效：

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

函数调用时，关键字参数必须跟在位置参数后面。所有传递的关键字参数都必须匹配一个函数接受的参数（比如，actor 不是函数 parrot 的有效参数），关键字参数的顺序并不重要。这也包括必选参数，（比如，parrot(voltage=1000) 也有效）。不能对同一个参数多次赋值，下面就是一个因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 argument 'a'
```

最后一个形参为 **name 形式时，接收一个字典（详见 typesmapping），该字典包含与函数中已定义形参对应之外的所有关键字参数。**name 形参可以与 *name 形参（下一小节介绍）组合使用（*name 必须在 **name 前面），*name 形参接收一个元组，该元组包含形参列表之外的位置参数。例如，可以定义下面这样的函数：

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

该函数可以用如下方式调用：

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

输出结果如下：

```
-- 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
sketch : Cheese Shop Sketch
```

注意，关键字参数在输出结果中的顺序与调用函数时的顺序一致。

4.8.3 特殊参数

默认情况下，参数可以按位置或显式关键字传递给 Python 函数。为了让代码易读、高效，最好限制参数的传递方式，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、按位置或关键字，还是仅按关键字传递。

函数定义如下：

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
```

(下页继续)

(续上页)

```
        |                                  - Keyword only
         -- Positional only
```

/ 和 * 是可选的。这些符号表明形参如何把参数值传递给函数：位置、位置或关键字、关键字。关键字形参也叫作命名形参。

位置或关键字参数

函数定义中未使用 / 和 * 时，参数可以按位置或关键字传递给函数。

仅位置参数

此处再介绍一些细节，特定形参可以标记为 仅限位置。仅限位置时，形参的顺序很重要，且这些形参不能用关键字传递。仅限位置形参应放在 / （正斜杠）前。/ 用于在逻辑上分割仅限位置形参与其它形参。如果函数定义中没有 /，则表示没有仅限位置形参。

/ 后可以是 位置或关键字或 仅限关键字形参。

仅限关键字参数

把形参标记为 仅限关键字，表明必须以关键字参数形式传递该形参，应在参数列表中第一个 仅限关键字形参前添加 *。

函数示例

请看下面的函数定义示例，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，可以按位置也可以按关键字传递参数：

```
>>> standard_arg(2)
2

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

第二个函数 pos_only_arg 的函数定义中有 /，仅限使用位置形参：

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
```

(下页继续)

(续上页)

```
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword␣
↪arguments: 'arg'
```

第三个函数 kwd_only_args 的函数定义通过 * 表明仅限关键字参数：

```
>>> 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
```

最后一个函数在同一个函数定义中，使用了全部三种调用惯例：

```
>>> 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 some positional-only arguments passed as keyword␣
↪arguments: 'pos_only'
```

下面的函数定义中，kwds 把 name 当作键，因此，可能与位置参数 name 产生潜在冲突：

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

调用该函数不可能返回 True，因为关键字 'name' 总与第一个形参绑定。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'
>>>
```

加上 / （仅限位置参数）后，就可以了。此时，函数定义把 name 当作位置参数，'name' 也可以作为关键字参数的键：

```
>>> 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。

小结

以下用例决定哪些形参可以用于函数定义：

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

说明：

- 使用仅限位置形参，可以让用户无法使用形参名。形参名没有实际意义时，强制调用函数的实参顺序时，或同时接收位置形参和关键字时，这种方式很有用。
- 当形参名有实际意义，且显式名称可以让函数定义更易理解时，阻止用户依赖传递实参的位置时，才使用关键字。
- 对于 API，使用仅限位置形参，可以防止未来修改形参名时造成破坏性的 API 变动。

4.8.4 任意实参列表

调用函数时，使用任意数量的实参是最少见的选项。这些实参包含在元组中（详见元组和序列）。在可变数量的实参之前，可能有若干个普通参数：

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

variadic 参数用于采集传递给函数的所有剩余参数，因此，它们通常在形参列表的末尾。`*args` 形参后的任何形式参数只能是仅限关键字参数，即只能用作关键字参数，不能用作位置参数：

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

4.8.5 解包实参列表

函数调用要求独立的位置参数，但实参在列表或元组里时，要执行相反的操作。例如，内置的 `range()` 函数要求独立的 *start* 和 *stop* 实参。如果这些参数不是独立的，则要在调用函数时，用 `*` 操作符把实参从列表或元组解包出来：

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

同样，字典可以用 `**` 操作符传递关键字参数：

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

4.8.6 Lambda 表达式

lambda 关键字用于创建小巧的匿名函数。lambda a, b: a+b 函数返回两个参数的和。Lambda 函数可用于任何需要函数对象的地方。在语法上，匿名函数只能是单个表达式。在语义上，它只是常规函数定义的语法糖。与嵌套函数定义一样，lambda 函数可以引用包含作用域中的变量：

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

上例用 lambda 表达式返回函数。还可以把匿名函数用作传递的实参：

```
>>> 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')]
```

4.8.7 文档字符串

以下是文档字符串内容和格式的约定。

第一行应为对象用途的简短摘要。为保持简洁，不要在这里显式说明对象名或类型，因为可通过其他方式获取这些信息（除非该名称碰巧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。

文档字符串为多行时，第二行应为空白行，在视觉上将摘要与其余描述分开。后面的行可包含若干段落，描述对象的调用约定、副作用等。

Python 解析器不会删除 Python 中多行字符串字面值的缩进，因此，文档处理工具应在必要时删除缩进。这项操作遵循以下约定：文档字符串第一行 之后的第一个非空行决定了整个文档字符串的缩进量（第一行通常与字符串开头的引号相邻，其缩进在字符串中并不明显，因此，不能用第一行的缩进），然后，删除字符串中所有行开头处与此缩进“等价”的空白符。不能有比此缩进更少的行，但如果出现了缩进更少的行，应删除这些行的所有前导空白符。转化制表符后（通常为 8 个空格），应测试空白符的等效性。

下面是多行文档字符串的一个例子：

```
>>> 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.
```

4.8.8 函数注解

函数注解是可选的用户自定义函数类型的元数据完整信息（详见 **PEP 3107** 和 **PEP 484** ）。

标注 以字典的形式存放在函数的 `__annotations__` 属性中而对函数的其他部分没有影响。形参标注的定义方式是在形参名后加冒号，后面跟一个会被求值为标注的值的表达式。返回值标注的定义方式是加组合符号 `->`，后面跟一个表达式，这样的校注位于形参列表和表示 `def` 语句结束的冒号。下面的示例有一个必须的参数、一个可选的关键字参数以及返回值都带有相应的标注:

```
>>> 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'
```

4.9 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* 了。大多数语言都能以不同的风格被编写（或更准确地说，被格式化）；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。

Python 项目大多都遵循 **PEP 8** 的风格指南；它推行的编码风格易于阅读、赏心悦目。Python 开发者均应抽时间悉心研读；以下是该提案中的核心要点：

- 缩进，用 4 个空格，不要用制表符。

  4 个空格是小缩进（更深嵌套）和大缩进（更易阅读）之间的折中方案。制表符会引起混乱，最好别用。

- 换行，一行不超过 79 个字符。

  这样换行的小屏阅读体验更好，还便于在大屏显示器上并排阅读多个代码文件。

- 用空行分隔函数和类，及函数内较大的代码块。

- 最好把注释放到单独一行。

- 使用文档字符串。

- 运算符前后、逗号后要用空格，但不要直接在括号内使用：`a = f(1, 2) + g(3, 4)`。

- 类和函数的命名要一致；按惯例，命名类用 `UpperCamelCase`，命名函数与方法用 `lowercase_with_underscores`。命名方法中第一个参数总是用 `self` (类和方法详见初探类)。

- 编写用于国际多语环境的代码时，不要用生僻的编码。Python 默认的 UTF-8 或纯 ASCII 可以胜任各种情况。

- 同理，就算多语阅读、维护代码的可能再小，也不要在标识符中使用非 ASCII 字符。

备注

CHAPTER 5

数据结构

本章深入讲解之前学过的一些内容，同时，还增加了新的知识点。

5.1 列表详解

列表数据类型支持很多方法，列表对象的所有方法所示如下：

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

在列表末尾添加一个元素，相当于 `a[len(a):] = [x]` 。

`list.`**`extend`**`(`*iterable*`)`

用可迭代对象的元素扩展列表。相当于 `a[len(a):] = iterable` 。

`list.`**`insert`**`(`*i*, *x*`)`

在指定位置插入元素。第一个参数是插入元素的索引，因此，`a.insert(0, x)` 在列表开头插入元素，`a.insert(len(a), x)` 等同于 `a.append(x)` 。

`list.`**`remove`**`(`*x*`)`

从列表中删除第一个值为 *x* 的元素。未找到指定元素时，触发 `ValueError` 异常。

`list.`**`pop`**`(`[*i*]`)`

删除列表中指定位置的元素，并返回被删除的元素。未指定位置时，`a.pop()` 删除并返回列表的最后一个元素。（方法签名中 *i* 两边的方括号表示该参数是可选的，不是要求输入方括号。这种表示法常见于 Python 参考库）。

`list.`**`clear`**`()`

删除列表里的所有元素，相当于 `del a[:]` 。

`list.`**`index`**`(`*x*[, *start*[, *end*]]`)`

返回列表中第一个值为 *x* 的元素的零基索引。未找到指定元素时，触发 `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*`)`

返回列表中元素 *x* 出现的次数。

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

就地排序列表中的元素（要了解自定义排序参数，详见 sorted()）。

list.**reverse**()

翻转列表中的元素。

list.**copy**()

返回列表的浅拷贝。相当于 a[:] 。

多数列表方法示例：

```
>>> 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 at 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 insert, remove 或 sort 等仅修改列表的方法都不会打印返回值 -- 它们返回默认值 None。[1] 这是适用于 Python 中所有可变数据结构的设计原则。

还有，不是所有数据都可以排序或比较。例如，[None, 'hello', 10] 就不可排序，因为整数不能与字符串对比，而 *None* 不能与其他类型对比。有些类型根本就没有定义顺序关系，例如，3+4j < 5+7j 这种对比操作就是无效的。

5.1.1 用列表实现堆栈

通过列表方法可以非常容易地将列表作为栈来使用，最后添加的元素将最先被提取（“后进先出”）。要向栈顶添加一个条目，请使用 append()。要从栈顶提取一个条目，请使用 pop()，无需显式指定索引。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]
```

[1] 别的语言可能会将可变对象返回，允许方法连续执行，例如 d->insert("a")->remove("b")->sort();。

5.1.2 用列表实现队列

列表也可以用作队列，最先加入的元素，最先取出（“先进先出”）；然而，列表作为队列的效率很低。因为，在列表末尾添加和删除元素非常快，但在列表开头插入或移除元素却很慢（因为所有其他元素都必须移动一位）。

实现队列最好用 `collections.deque`，可以快速从两端添加或删除元素。例如：

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

5.1.3 列表推导式

列表推导式创建列表的方式更简洁。常见的用法为，对序列或可迭代对象中的每个元素应用某种操作，用生成的结果创建新的列表；或用满足特定条件的元素创建子序列。

例如，创建平方值的列表：

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

注意，这段代码创建（或覆盖）变量 `x`，该变量在循环结束后仍然存在。下述方法可以无副作用地计算平方列表：

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

或等价于：

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

上面这种写法更简洁、易读。

列表推导式的方括号内包含以下内容：一个表达式，后面为一个 `for` 子句，然后，是零个或多个 `for` 或 `if` 子句。结果是由表达式依据 `for` 和 `if` 子句求值计算而得出一个新列表。举例来说，以下列表推导式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)]
```

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，上面两段代码中，for 和 if 的顺序相同。

表达式是元组（例如上例的 (x, y)）时，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
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # 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，甚至可以是另一个列表推导式。

下面这个 3x4 矩阵，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],
... ]
```

下面的列表推导式可以转置行列：

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

如我们在之前小节中看到的，内部的列表推导式是在它之后的 for 的上下文中被求值的，所以这个例子等价于：

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
```

（下页继续）

(续上页)

```
...
>>> 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，最好用内置函数替代复杂的流程语句。此时，`zip()` 函数更好用：

```
>>> 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*解包实参列表*。

5.2 del 语句

有一种方式可以按索引而不是值从列表中移除条目：`del` 语句。这与返回一个值的 `pop()` 方法不同。`del` 语句也可用于从列表中移除切片或清空整个列表（我们之前通过将切片赋值为一个空列表实现过此操作）。例如：

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

`del` 也可以用来删除整个变量：

```
>>> del a
```

此后，再引用 `a` 就会报错（直到为它赋与另一个值）。后文会介绍 `del` 的其他用法。

5.3 元组和序列

列表和字符串有很多共性，例如，索引和切片操作。这两种数据类型是 *序列*（参见 typesseq）。随着 Python 语言的发展，其他的序列类型也被加入其中。本节介绍另一种标准序列类型：*元组*。

元组由多个用逗号隔开的值组成，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])
```

输出时，元组都要由圆括号标注，这样才能正确地解释嵌套元组。输入时，圆括号可有可无，不过经常是必须的（如果元组是更大的表达式的一部分）。不允许为元组中的单个元素赋值，当然，可以创建含列表等可变对象的元组。

虽然，元组与列表很像，但使用场景不同，用途也不同。元组是*immutable*（不可变的），一般可包含异质元素序列，通过解包（见本节下文）或索引访问（如果是 `namedtuples`，可以属性访问）。列表是*mutable*（可变的），列表元素一般为同质类型，可迭代访问。

构造 0 个或 1 个元素的元组比较特殊：为了适应这种情况，对句法有一些额外的改变。用一对空圆括号就可以创建空元组；只有一个元素的元组可以通过在这个元素后添加逗号来构建（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* 的例子：值 `12345`, `54321` 和 `'hello!'` 一起被打包进元组。逆操作也可以：

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

称之为 *序列解包* 也是妥妥的，适用于右侧的任何序列。序列解包时，左侧变量与右侧序列元素的数量应相等。注意，多重赋值其实只是元组打包和序列解包的组合。

5.4 集合

Python 还支持 集合这种数据类型。集合是由不重复元素组成的无序容器。基本用法包括成员检测、消除重复元素。集合对象支持合集、交集、差集、对称差分等数学运算。

创建集合用花括号或 `set()` 函数。注意，创建空集合只能用 `set()`，不能用 `{}`，`{}` 创建的是空字典，下一小节介绍数据结构：字典。

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'}
```

与列表推导式 类似，集合也支持推导式：

```
>>> 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 typesmapping)。字典在其他语言中可能会被称为“关联存储”或“关联数组”。不同于以固定范围的数字进行索引的序列，字典是以 键进行索引的，键可以是任何不可变类型；字符串和数字总是可以作为键。如果一个元组只包含字符串、数字或元组则也可以作为键；如果一个元组直接或间接地包含了任何可变对象，则不能作为键。列表不能作为键，因为列表可以使用索引赋值、切片赋值或者 `append()` 和 `extend()` 等方法进行原地修改列表。

可以把字典理解为 键值对的集合，但字典的键必须是唯一的。花括号 `{}` 用于创建空字典。另一种初始化字典的方式是，在花括号里输入逗号分隔的键值对，这也是字典的输出方式。

字典的主要用途是通过关键字存储、提取值。用 `del` 可以删除键值对。用已存在的关键字存储值，与该关键字关联的旧值会被取代。通过不存在的键提取值，则会报错。

对字典执行 `list(d)` 操作，返回该字典中所有键的列表，按插入次序排列（如需排序，请使用 `sorted(d)`）。检查字典里是否存在某个键，使用关键字 `in`。

以下是一些字典的简单示例：

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

(下页继续)

(续上页)

```
>>> 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()` 构造函数可以直接用键值对序列创建字典：

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

字典推导式可以用任意键值表达式创建字典：

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

关键字是比较简单的字符串时，直接用关键字参数指定键值对更便捷：

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

5.6 循环的技巧

当对字典执行循环时，可以使用 `items()` 方法同时提取键及其对应的值。

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

在序列中循环时，用 `enumerate()` 函数可以同时取出位置索引和对应的值：

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

同时循环两个或多个序列时，用 `zip()` 函数可以将其内的元素一一匹配：

```
>>> 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))
```

(下页继续)

(续上页)

```
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

为了逆向对序列进行循环，可以求出欲循环的正向序列，然后调用 `reversed()` 函数：

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

按指定顺序循环序列，可以用 `sorted()` 函数，在不改动原序列的基础上，返回一个重新的序列：

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

使用 `set()` 去除序列中的重复元素。使用 `sorted()` 加 `set()` 则按排序后的顺序，循环遍历序列中的唯一元素：

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

一般来说，在循环中修改列表的内容时，创建新列表比较简单，且安全：

```
>>> 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 条件句不只可以进行比较，还可以使用任意运算符。

比较运算符 in 和 not in 用于执行确定一个值是否存在（或不存在）于某个容器中的成员检测。运算符 is 和 is not 用于比较两个对象是否是同一个对象。所有比较运算符的优先级都一样，且低于任何数值运算符。

比较操作支持链式操作。例如，a < b == c 校验 a 是否小于 b，且 b 是否等于 c。

比较操作可以用布尔运算符 and 和 or 组合，并且，比较操作（或其他布尔运算）的结果都可以用 not 取反。这些操作符的优先级低于比较操作符；not 的优先级最高，or 的优先级最低，因此，A and not B or C 等价于 (A and (not B)) or C。与其他运算符操作一样，此处也可以用圆括号表示想要的组合。

布尔运算符 and 和 or 是所谓的 短路运算符：其参数从左至右求值，一旦可以确定结果，求值就会停止。例如，如果 A 和 C 为真，B 为假，那么 A and B and C 不会对 C 求值。用作普通值而不是布尔值时，短路运算符的返回值通常是最后一个求了值的参数。

还可以把比较运算或其它布尔表达式的结果赋值给变量，例如：

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

注意，Python 与 C 不同，在表达式内部赋值必须显式使用 海象运算符 :=。这避免了 C 程序中常见的问题：要在表达式中写 == 时，却写成了 =。

5.8 序列和其他类型的比较

序列对象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，比较前两个对应元素，如果不相等，则可确定比较结果；如果相等，则比较之后的两个元素，以此类推，直到其中一个序列结束。如果要比较的两个元素本身是相同类型的序列，则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，则两个序列相等。如果一个序列是另一个的初始子序列，则较短的序列可被视为较小（较少）的序列。对于字符串来说，字典式顺序使用 Unicode 码位序号排序单个字符。下面列出了一些比较相同类型序列的例子：

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

注意，当比较不同类型的对象时，只要待比较的对象提供了合适的比较方法，就可以使用 < 和 > 进行比较。例如，混合的数字类型通过数字值进行比较，所以，0 等于 0.0，等等。如果没有提供合适的比较方法，解释器不会随便给出一个比较结果，而是引发 TypeError 异常。

备注

CHAPTER 6

模块

退出 Python 解释器后，再次进入时，之前在 Python 解释器中定义的函数和变量就丢失了。因此，编写较长程序时，最好用文本编辑器代替解释器，执行文件中的输入内容，这就是编写 *脚本*。随着程序越来越长，为了方便维护，最好把脚本拆分成多个文件。编写脚本还一个好处，不同程序调用同一个函数时，不用把函数定义复制到各个程序。

为实现这些需求，Python 把各种定义存入一个文件，在脚本或解释器的交互式实例中使用。这个文件就是 *模块*；模块中的定义可以 *导入* 到其他模块或 *主模块*（在顶层和计算器模式下，执行脚本中可访问的变量集）。

模块是包含 Python 定义和语句的文件。其文件名是模块名加后缀名 `.py` 。在模块内部，通过全局变量 `__name__` 可以获取模块名（即字符串）。例如，用文本编辑器在当前目录下创建 `fibo.py` 文件，输入以下内容：

```
# 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，用以下命令导入该模块：

```
>>> import fibo
```

此操作不会直接把 `fibo` 中定义的函数名称添加到当前*namespace* 中（请参阅*Python 作用域和命名空间* 了解详情）；它只是将模块名称 `fibo` 添加到那里。使用该模块名称你可以访问其中的函数：

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

(下页继续)

(续上页)

```
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

如果经常使用某个函数，可以把它赋值给局部变量：

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

6.1 模块详解

模块包含可执行语句及函数定义。这些语句用于初始化模块，且仅在 import 语句 *第一次*遇到模块名时执行。[1] (文件作为脚本运行时，也会执行这些语句。)

每个模块都有自己的私有命名空间，它会被用作模块中定义的所有函数的全局命名空间。因此，模块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，如果你知道要怎么做就可以通过与引用模块函数一样的标记法 `modname.itemname` 来访问一个模块的全局变量。

模块可以导入其他模块。根据惯例可以将所有 `import` 语句都放在模块（或者也可以说是脚本）的开头但这并非强制要求。如果被放置于一个模块的最高层级，则被导入的模块名称会被添加到该模块的全局命名空间。

还有一种 `import` 语句的变化形式可以将来自某个模块的名称直接导入到导入方模块的命名空间中。例如:

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

这条语句不会将所导入的模块的名称引入到局部命名空间中（因此在本示例中，`fibo` 将是未定义的名称）。

还有一种变体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
```

这种方式会导入所有不以下划线（`_`）开头的名称。大多数情况下，不要用这个功能，这种方式向解释器导入了一批未知的名称，可能会覆盖已经定义的名称。

注意，一般情况下，不建议从模块或包内导入 `*`，因为，这项操作经常让代码变得难以理解。不过，为了在交互式编译器中少打几个字，这么用也没问题。

模块名后使用 `as` 时，直接把 `as` 后的名称与导入模块绑定。

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

与 `import fibo` 一样，这种方式也可以有效地导入模块，唯一的区别是，导入的名称是 `fib`。

`from` 中也可以使用这种方式，效果类似：

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

[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会将函数名称添加到模块的全局命名空间。

备注： 为了保证运行效率，每次解释器会话只导入一次模块。如果更改了模块内容，必须重启解释器；仅交互测试一个模块时，也可以使用 `importlib.reload()`，例如 `import importlib; importlib.reload(modulename)`。

6.1.1 以脚本方式执行模块

可以用以下方式运行 Python 模块：

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

这项操作将执行模块里的代码，和导入模块一样，但会把 `__name__` 赋值为 `"__main__"`。也就是把下列代码添加到模块末尾：

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

这个文件既能被用作脚本，又能被用作一个可供导入的模块，因为解析命令行参数的那两行代码只有在模块作为“main”文件执行时才会运行：

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

当这个模块被导入到其它模块时，那两行代码不运行：

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

这常用于为模块提供一个便捷的用户接口，或用于测试（把模块作为执行测试套件的脚本运行）。

6.1.2 模块搜索路径

当导入一个名为 `spam` 的模块时，解释器首先会搜索具有该名称的内置模块。这些模块的名称在 `sys.builtin_module_names` 中列出。如果未找到，它将在变量 `sys.path` 所给出的目录列表中搜索名为 `spam.py` 的文件。`sys.path` 是从这些位置初始化的:

- 被命令行直接运行的脚本所在的目录（或未指定文件时的当前目录）。
- `PYTHONPATH`（目录列表，与 shell 变量 `PATH` 的语法一样）。
- 依赖于安装的默认值（按照惯例包括一个 `site-packages` 目录，由 `site` 模块处理）。

更多细节请参阅 sys-path-init。

备注： 在支持符号链接的文件系统中，“被命令行直接运行的脚本所在的目录”是符号链接最终指向的目录。换句话说，符号链接所在的目录并 **没有**被添加至模块搜索路径。

初始化后，Python 程序可以更改 `sys.path`。脚本所在的目录先于标准库所在的路径被搜索。这意味着，脚本所在的目录如果有和标准库同名的文件，那么加载的是该目录里的，而不是标准库的。这一般是一个错误，除非这样的替换是你有意为之。详见标准模块。

6.1.3 “已编译的” Python 文件

为了快速加载模块，Python 把模块的编译版本缓存在 __pycache__ 目录中，文件名为 module.*version*.pyc，version 对编译文件格式进行编码，一般是 Python 的版本号。例如，CPython 的 3.3 发行版中，spam.py 的编译版本缓存为 __pycache__/spam.cpython-33.pyc。这种命名惯例让不同 Python 版本编译的模块可以共存。

Python 对比编译版与源码的修改日期，查看编译版是否已过期，是否要重新编译。此进程完全是自动的。此外，编译模块与平台无关，因此，可在不同架构的系统之间共享相同的库。

Python 在两种情况下不检查缓存。一，从命令行直接载入的模块，每次都会重新编译，且不储存编译结果；二，没有源模块，就不会检查缓存。为了让一个库能以隐藏源代码的形式分发（通过将所有源代码变为编译后的版本），编译后的模块必须放在源目录而非缓存目录中，并且源目录绝不能包含同名的未编译的源模块。

给专业人士的一些小建议：

- 在 Python 命令中使用 -O 或 -OO 开关，可以减小编译模块的大小。-O 去除断言语句，-OO 去除断言语句和 __doc__ 字符串。有些程序可能依赖于这些内容，因此，没有十足的把握，不要使用这两个选项。“优化过的” 模块带有 opt- 标签，并且文件通常会一小些。将来的发行版或许会改进优化的效果。
- 从 .pyc 文件读取的程序不比从 .py 读取的执行速度快，.pyc 文件只是加载速度更快。
- compileall 模块可以为一个目录下的所有模块创建.pyc 文件。
- 本过程的细节及决策流程图，详见 **PEP 3147**。

6.2 标准模块

Python 自带一个标准模块的库，它在 Python 库参考（此处以下称为” 库参考” ）里另外描述。一些模块是内嵌到编译器里面的，它们给一些虽并非语言核心但却内嵌的操作提供接口，要么是为了效率，要么是给操作系统基础操作例如系统调入提供接口。这些模块集是一个配置选项，并且还依赖于底层的操作系统。例如，winreg 模块只在 Windows 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 编译器中。sys.ps1 和 sys.ps2 变量定义了一些字符，它们可以用作主提示符和辅助提示符:

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

只有解释器用于交互模式时，才定义这两个变量。

变量 sys.path 是字符串列表，用于确定解释器的模块搜索路径。该变量以环境变量 PYTHONPATH 提取的默认路径进行初始化，如未设置 PYTHONPATH，则使用内置的默认路径。可以用标准列表操作修改该变量:

```
>>> 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。返回结果是经过排序的字符串列表：

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', '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', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

没有参数时，dir() 列出当前已定义的名称：

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

注意它列出所有类型的名称：变量，模块，函数，……。

dir() 不会列出内置函数和变量的名称。这些内容的定义在标准模块 builtins 中：

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
```

(下页继续)

(续上页)

```
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
'zip']
```

6.4 包

包是通过使用“带点号模块名”来构造 Python 模块命名空间的一种方式。例如，模块名 `A.B` 表示名为 `A` 的包中名为 `B` 的子模块。就像使用模块可以让不同模块的作者不必担心彼此的全局变量名一样，使用带点号模块名也可以让 NumPy 或 Pillow 等多模块包的作者也不必担心彼此的模块名冲突。

假设要为统一处理声音文件与声音数据设计一个模块集（“包”）。声音文件的格式很多（通常以扩展名来识别，例如：`.wav`，`.aiff`，`.au`），因此，为了不同文件格式之间的转换，需要创建和维护一个不断增长的模块集合。为了实现对声音数据的不同处理（例如，混声、添加回声、均衡器功能、创造人工立体声效果），还要编写无穷无尽的模块流。下面这个分级文件树展示了这个包的架构：

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

导入包时，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来处理。这可以防止具有通用名称的目录如 `string` 在无意中屏蔽后续出现在模块搜索路径中的有效模块。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，这将在后面详细描述。

还可以从包中导入单个模块，例如：

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

这将加载子模块 `sound.effects.echo`。它必须通过其全名来引用。

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

另一种导入子模块的方法是：

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

这也会加载子模块 echo，并使其不必加包前缀，因此可按如下方式使用：

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

Import 语句的另一种变体是直接导入所需的函数或变量：

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

同样，这将加载子模块 echo，但这使其函数 echofilter() 直接可用：

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

注意，使用 from package import item 时，item 可以是包的子模块（或子包），也可以是包中定义的函数、类或变量等其他名称。import 语句首先测试包中是否定义了 item；如果未在包中定义，则假定 item 是模块，并尝试加载。如果找不到 item，则触发 ImportError 异常。

相反，使用 import item.subitem.subsubitem 句法时，除最后一项外，每个 item 都必须是包；最后一项可以是模块或包，但不能是上一项中定义的类、函数或变量。

6.4.1 从包中导入 *

使用 from sound.effects import * 时会发生什么？你可能希望它会查找并导入包的所有子模块，但事实并非如此。因为这将花费很长的时间，并且可能会产生你不想要的副作用，如果这种副作用被你设计为只有在导入某个特定的子模块时才应该发生。

唯一的解决办法是提供包的显式索引。import 语句使用如下惯例：如果包的 __init__.py 代码定义了列表 __all__，运行 from package import * 时，它就是被导入的模块名列表。发布包的新版本时，包的作者应更新此列表。如果包的作者认为没有必要在包中执行导入 * 操作，也可以不提供此列表。例如，sound/effects/__init__.py 文件可以包含以下代码：

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

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

请注意子模块可能会受到本地定义名称的影响。例如，如果你在 sound/effects/__init__.py 文件中添加了一个 reverse 函数，from sound.effects import * 将只导入 echo 和 surround 这两个子模块，但 **不会**导入 reverse 子模块，因为它被本地定义的 reverse 函数所遮挡：

```
__all__ = [
    "echo",      # refers to the 'echo.py' file
    "surround",  # refers to the 'surround.py' file
    "reverse",   # !!! refers to the 'reverse' function now !!!
]

def reverse(msg: str):  # <-- this name shadows the 'reverse.py' submodule
    return msg[::-1]    #     in the case of a 'from sound.effects import *'
```

如果没有定义 __all__，from sound.effects import * 语句 不会把包 sound.effects 中的所有子模块都导入到当前命名空间；它只是确保包 sound.effects 已被导入（可能还会运行 __init__.py 中的任何初始化代码），然后再导入包中定义的任何名称。这包括由 __init__.py 定义的任何名称（以及显式加载的子模块）。它还包括先前 import 语句显式加载的包里的任何子模块。请看以下代码：

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

在本例中，echo 和 surround 模块被导入到当前命名空间，因为在执行 from...import 语句时它们已在 sound.effects 包中定义了。（当定义了 __all__ 时也是如此）。

虽然，可以把模块设计为用 import * 时只导出遵循指定模式的名称，但仍不提倡在生产代码中使用这种做法。

记住，使用 from package import specific_submodule 没有任何问题！实际上，除了导入模块使用不同包的同名子模块之外，这种方式是推荐用法。

6.4.2 相对导入

当包由多个子包构成（如示例中的 sound 包）时，可以使用绝对导入来引用同级包的子模块。例如，如果 sound.filters.vocoder 模块需要使用 sound.effects 包中的 echo 模块，它可以使用 from sound.effects import echo。

你还可以编写相对导入代码，即使用 from module import name 形式的 import 语句。这些导入使用前导点号来表示相对导入所涉及的当前包和上级包。例如对于 surround 模块，可以使用:

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

注意，相对导入基于当前模块名。因为主模块名永远是 "__main__"，所以如果计划将一个模块用作 Python 应用程序的主模块，那么该模块内的导入语句必须始终使用绝对导入。

6.4.3 多目录中的包

包还支持一个特殊属性 __path__。在包的 __init__.py 中的代码被执行前，该属性被初始化为一个只含一项的列表，该项是一个字符串，是 __init__.py 所在目录的名称。可以修改此变量；这样做会改变在此包中搜索模块和子包的方式。

这个功能虽然不常用，但可用于扩展包中的模块集。

备注

CHAPTER 7

输入与输出

程序输出有几种显示方式；数据既可以输出供人阅读的形式，也可以写入文件备用。本章探讨一些可用的方式。

7.1 更复杂的输出格式

到目前为止我们已遇到过两种写入值的方式: *表达式语句*和 `print()` 函数。（第三种方式是使用文件对象的 `write()` 方法；标准输出文件可以被引用为 `sys.stdout`。更多相关信息请参阅标准库参考）。

对输出格式的控制不只是打印空格分隔的值，还需要更多方式。格式化输出包括以下几种方法。

- 使用*格式化字符串字面值*，要在字符串开头的引号/三引号前添加 `f` 或 `F`。在这种字符串中，可以在 `{` 和 `}` 字符之间输入引用的变量，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()` 方法需要更多手动操作。该方法也用 `{` 和 `}` 标记替换变量的位置，虽然这种方法支持详细的格式化指令，但需要提供格式化信息。

```
>>> 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，还可以用字符串切片和合并操作完成字符串处理操作，创建任何排版布局。字符串类型还支持将字符串按给定列宽进行填充，这些方法也很有用。

如果不需要花哨的输出，只想快速显示变量进行调试，可以用 `repr()` 或 `str()` 函数把值转化为字符串。

`str()` 函数返回供人阅读的值，`repr()` 则生成适于解释器读取的值（如果没有等效的语法，则强制执行 `SyntaxError`）。对于没有支持供人阅读展示结果的对象，`str()` 返回与 `repr()` 相同的值。一般情况下，数字、列表或字典等结构的值，使用这两个函数输出的表现形式是一样的。字符串有两种不同的表现形式。

示例如下：

```
>>> 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 `Template` 类，提供了将值替换为字符串的另一种方法。该类使用 `$x` 占位符，并用字典的值进行替换，但对格式控制的支持比较有限。

7.1.1 格式化字符串字面值

格式化字符串字面值（简称为 f-字符串）在字符串前加前缀 `f` 或 `F`，通过 `{expression}` 表达式，把 Python 表达式的值添加到字符串内。

格式说明符是可选的，写在表达式后面，可以更好地控制格式化值的方式。下例将 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 `':'` 后传递整数，为该字段设置最小字符宽度，常用于列对齐：

```
>>> 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
```

还有一些修饰符可以在格式化前转换值。`'!a'` 应用 `ascii()`，`'!s'` 应用 `str()`，`'!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'.
```

= 说明符可被用于将一个表达式扩展为表达式文本、等号再加表达式求值结果的形式。

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
```

(下页继续)

(续上页)

```
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

请参阅 自说明型表达式以了解 = 说明符的更多信息。有关这些格式说明的详情，请查看针对 formatspec 的参考指南。

7.1.2 字符串 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!"
```

花括号及之内的字符（称为格式字段）被替换为传递给 `str.format()` 方法的对象。花括号中的数字表示传递给 `str.format()` 方法的对象所在的位置。

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

`str.format()` 方法中使用关键字参数名引用值。

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

位置参数和关键字参数可以任意组合：

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

如果不想分拆较长的格式字符串，最好按名称引用变量进行格式化，不要按位置。这项操作可以通过传递字典，并用方括号 `'[]'` 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
```

这也可以通过将 `table` 字典作为采用 `**` 标记的关键字参数传入来实现。

```
>>> 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
```

与内置函数 `vars()` 结合使用时，这种方式非常实用，可以返回包含所有局部变量的字典。

举个例子，以下几行代码将产生一组整齐的数据列，包含给定的整数及其平方与立方：

```
>>> 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
```

(下页继续)

(续上页)

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

str.format() 进行字符串格式化的完整概述详见 formatstrings 。

7.1.3 手动格式化字符串

下面是使用手动格式化方式实现的同一个平方和立方的表：

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

（注意，每列之间的空格是通过使用 print() 添加的：它总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，对给定宽度字段中的字符串进行右对齐。同类方法还有 str.ljust() 和 str.center() 。这些方法不写入任何内容，只返回一个新字符串，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；虽然这种方式会弄乱列布局，但也比另一种方法好，后者在显示值时可能不准确（如果真的想截断字符串，可以使用 x.ljust(n)[:n] 这样的切片操作。）

另一种方法是 str.zfill() ，该方法在数字字符串左边填充零，且能识别正负号：

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

7.1.4 旧式字符串格式化方法

% 运算符（求余符）也可用于字符串格式化。给定 'string' % values，则 string 中的 % 实例会以零个或多个 values 元素替换。此操作被称为字符串插值。例如：

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

old-string-formatting 小节介绍更多相关内容。

7.2 读写文件

`open()` 返回一个*file object*，最常使用的是两个位置参数和一个关键字参数：`open(filename, mode, encoding=None)`

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

第一个实参是文件名字符串。第二个实参是包含描述文件使用方式字符的字符串。*mode* 的值包括 `'r'`，表示文件只能读取；`'w'` 表示只能写入（现有同名文件会被覆盖）；`'a'` 表示打开文件并追加内容，任何写入的数据会自动添加到文件末尾。`'r+'` 表示打开文件进行读写。*mode* 实参是可选的，省略时的默认值为 `'r'`。

通常情况下，文件是以 *text mode* 打开的，也就是说，你从文件中读写字符串，这些字符串是以特定的 *encoding* 编码的。如果没有指定 *encoding*，默认的是与平台有关的（见 `open()`）。因为 UTF-8 是现代事实上的标准，除非你知道你需要使用一个不同的编码，否则建议使用 `encoding="utf-8"`。在模式后面加上一个 `'b'`，可以用 *binary mode* 打开文件。二进制模式的数据是以 `bytes` 对象的形式读写的。在二进制模式下打开文件时，你不能指定 *encoding*。

在文本模式下读取文件时，默认把平台特定的行结束符（Unix 上为 `\n`, Windows 上为 `\r\n`）转换为 `\n`。在文本模式下写入数据时，默认把 `\n` 转换回平台特定结束符。这种操作方式在后台修改文件数据对文本文件来说没有问题，但会破坏 JPEG 或 EXE 等二进制文件中的数据。注意，在读写此类文件时，一定要使用二进制模式。

在处理文件对象时，最好使用 `with` 关键字。优点是，子句体结束后，文件会正确关闭，即便触发异常也可以。而且，使用 `with` 相比等效的 `try-finally` 代码块要简短得多：

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

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

如果没有使用 `with` 关键字，则应调用 `f.close()` 关闭文件，即可释放文件占用的系统资源。

> **警告：** 调用 `f.write()` 时，未使用 `with` 关键字，或未调用 `f.close()`，即使程序正常退出，也 ** 可能 ** 导致 `f.write()` 的参数没有完全写入磁盘。

通过 `with` 语句，或调用 `f.close()` 关闭文件对象后，再次使用该文件对象将会失败。

```
>>> 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

本节下文中的例子假定已创建 `f` 文件对象。

`f.read(size)` 可用于读取文件内容，它会读取一些数据，并返回字符串（文本模式），或字节串对象（在二进制模式下）。*size* 是可选的数值参数。省略 *size* 或 *size* 为负数时，读取并返回整个文件的内容；文件大小是内存的两倍时，会出现问题。*size* 取其他值时，读取并返回最多 *size* 个字符（文本模式）或 *size* 个字节（二进制模式）。如已到达文件末尾，`f.read()` 返回空字符串（`''`）。

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

（下页继续）

(续上页)

```
>>> f.read()
''
```

`f.readline()` 从文件中读取单行数据；字符串末尾保留换行符（`\n`），只有在文件不以换行符结尾时，文件的最后一行才会省略换行符。这种方式让返回值清晰明确；只要 `f.readline()` 返回空字符串，就表示已经到达了文件末尾，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()
''
```

从文件中读取多行时，可以用循环遍历整个文件对象。这种操作能高效利用内存，快速，且代码简单：

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

如需以列表形式读取文件中的所有行，可以用 `list(f)` 或 `f.readlines()`。

`f.write(string)` 把 *string* 的内容写入文件，并返回写入的字符数。

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

写入其他类型的对象前，要先把它们转化为字符串（文本模式）或字节对象（二进制模式）：

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

`f.tell()` 返回整数，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，以及文本模式下的意义不明的数字。

`f.seek(offset, whence)` 可以改变文件对象的位置。通过向参考点添加 *offset* 计算位置；参考点由 *whence* 参数指定。*whence* 值为 0 时，表示从文件开头计算，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。省略 *whence* 时，其默认值为 0，即使用文件开头作为参考点。

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

在文本文件（模式字符串未使用 `b` 时打开的文件）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），唯一有效的 *offset* 值是能从 `f.tell()` 中返回的，或 0。其他 *offset* 值都会产生未定义的行为。

文件对象还有一些额外的方法，如使用频率较低的 `isatty()` 和 `truncate()` 等；有关文件对象的完整指南请查阅标准库参考。

7.2.2 使用 json 保存结构化数据

字符串可以很容易地写入文件或从文件中读取。数字则更麻烦一些，因为 `read()` 方法只返回字符串，而字符串必须传给 `int()` 这样的函数，它接受 `'123'` 这样的字符串并返回其数值 123。当你想要保存嵌套列表和字典等更复杂的数据类型时，手动执行解析和序列化操作将会变得非常复杂。

Python 允许你使用流行的数据交换格式 JSON (JavaScript Object Notation)，而不是让用户持续编写和调试代码来将复杂的数据类型存入文件中。标准库模块 `json` 可以接受带有层级结构的 Python 数据，并将其转换为字符串表示形式；这个过程称为 *serializing*。根据字符串表示形式重建数据则称为 *deserializing*。在序列化和反序列化之间，用于代表对象的字符串可以存储在文件或数据库中，或者通过网络连接发送到远端主机。

备注： JSON 格式通常用于现代应用程序的数据交换。程序员早已对它耳熟能详，可谓是交互操作的不二之选。

只需一行简单的代码即可查看某个对象的 JSON 字符串表现形式：

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

`dumps()` 函数还有一个变体，`dump()`，它只将对象序列化为*text file* 。因此，如果 `f` 是*text file* 对象，可以这样做：

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

要再次解码对象，如果 `f` 是已打开、供读取的*binary file* 或*text file* 对象：

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

备注： JSON 文件必须以 UTF-8 编码。当打开 JSON 文件作为一个*text file* 用于读写时，使用 `encoding="utf-8"` 。

这种简单的序列化技术可以处理列表和字典，但在 JSON 中序列化任意类的实例，则需要付出额外努力。`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：如果解序化的数据是由手段高明的攻击者精心设计的，这种不受信任来源的 pickle 数据可以执行任意代码。

CHAPTER 8

错误和异常

至此，本教程还未深入介绍错误信息，但如果您尝试过本教程前文中的例子，应该已经看到过一些错误信息。错误可（至少）被分为两种：*语法错误*和 *异常*。

8.1 语法错误

语法错误又称解析错误，是学习 Python 时最常见的错误：

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

解析器会复现出现句法错误的代码行，并用小“箭头”指向行里检测到的第一个错误。错误是由箭头上方的 token 触发的（至少是在这里检测出的）：本例中，在 `print()` 函数中检测到错误，因为，在它前面缺少冒号（`':'`）。错误信息还输出文件名与行号，在使用脚本文件时，就可以知道去哪里查错。

8.2 异常

即使语句或表达式使用了正确的语法，执行时仍可能触发错误。执行时检测到的错误称为 *异常*，异常不一定导致严重的后果：很快我们就能学会如何处理 Python 的异常。大多数异常不会被程序处理，而是显示下列错误信息：

```
>>> 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):
```

(下页继续)

(续上页)

```
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

错误信息的最后一行说明程序遇到了什么类型的错误。异常有不同的类型，而类型名称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（虽然这种规范很有用）。标准的异常类型是内置的标识符（不是保留关键字）。

此行其余部分根据异常类型，结合出错原因，说明错误细节。

错误信息开头用堆栈回溯形式展示发生异常的语境。一般会列出源代码行的堆栈回溯；但不会显示从标准输入读取的行。

bltin-exceptions 列出了内置异常及其含义。

8.3 异常的处理

可以编写程序处理选定的异常。下例会要求用户一直输入内容，直到输入有效的整数，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）；注意，用户中断程序会触发 KeyboardInterrupt 异常。

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

try 语句的工作原理如下：

- 首先，执行 *try* 子句（try 和 except 关键字之间的（多行）语句）。
- 如果没有触发异常，则跳过 *except* 子句，try 语句执行完毕。
- 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。如果异常的类型与 except 关键字后指定的异常相匹配，则会执行 *except* 子句，然后跳到 try/except 代码块之后继续执行。
- 如果发生的异常与 *except* 子句中指定的异常不匹配，则它会被传递到外层的 try 语句中；如果没有找到处理句柄，则它是一个 *未处理异常* 且执行将停止并输出一条错误消息。

try 语句可以有多个 *except* 子句来为不同的异常指定处理程序。但最多只有一个处理程序会被执行。处理程序只处理对应的 *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。*except* 子句可以用带圆括号的元组来指定多个异常，例如:

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

如果发生的异常与 except 子句中的类是同一个类或是它的基类时，则该类与该异常相兼容（反之则不成立 --- 列出派生类的 *except* 子句与基类不兼容）。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
```

(下页继续)

(续上页)

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

请注意如果颠倒 *except* 子句的顺序（把 except B 放在最前），则会输出 B, B, B --- 即触发了第一个匹配的 *except* 子句。

发生异常时，它可能具有关联值，即异常 *参数*。是否需要参数，以及参数的类型取决于异常的类型。

except 子句可能会在异常名称后面指定一个变量。这个变量将被绑定到异常实例，该实例通常会有一个存储参数的 args 属性。为了方便起见，内置异常类型定义了 __str__() 来打印所有参数而不必显式地访问 .args。

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception type
...     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
```

未处理异常的 __str__() 输出会被打印为该异常消息的最后部分 ('detail')。

BaseException 是所有异常的共同基类。它的一个子类，Exception，是所有非致命异常的基类。不是 Exception 的子类的异常通常不被处理，因为它们被用来指示程序应该终止。它们包括由 sys.exit() 引发的 SystemExit，以及当用户希望中断程序时引发的 KeyboardInterrupt。

Exception 可以被用作通配符，捕获（几乎）一切。然而，好的做法是，尽可能具体地说明我们打算处理的异常类型，并允许任何意外的异常传播下去。

处理 Exception 最常见的模式是打印或记录异常，然后重新提出（允许调用者也处理异常）：

```
import sys

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

try ... except 语句具有可选的 *else* 子句，该子句如果存在，它必须放在所有 *except* 子句之后。它适用于 *try* 子句没有引发异常但又必须要执行的代码。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，可以避免意外捕获非 try ... except 语句保护的代码触发的异常。

异常处理程序不仅会处理在 *try* 子句中立刻发生的异常，还会处理在 *try* 子句中调用（包括间接调用）的函数。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 触发异常

raise 语句支持强制触发指定的异常。例如:

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

raise 唯一的参数就是要触发的异常。这个参数必须是异常实例或异常类（派生自 BaseException 类，例如 Exception 或其子类）。如果传递的是异常类，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()'
```

如果只想判断是否触发了异常，但并不打算处理该异常，则可以使用更简单的 raise 语句重新触发异常:

```
>>> 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 异常链

如果一个未处理的异常发生在 except 部分内，它将会有被处理的异常附加到它上面，并包括在错误信息中:

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: unable to handle error
```

为了表明一个异常是另一个异常的直接后果，raise 语句允许一个可选的 from 子句:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

转换异常时，这种方式很有用。例如:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

它还允许使用 from None 表达禁用自动异常链:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

异常链机制详见 bltin-exceptions。

8.6 用户自定义异常

程序可以通过创建新的异常类命名自己的异常（Python 类的内容详见类）。不论是以直接还是间接的方式，异常都应从 Exception 类派生。

异常类可以被定义成能做其他类所能做的任何事，但通常应当保持简单，它往往只提供一些属性，允许相应的异常处理程序提取有关错误的信息。

大多数异常命名都以“Error”结尾，类似标准异常的命名。

许多标准模块定义了自己的异常，以报告他们定义的函数中可能出现的错误。

8.7 定义清理操作

try 语句还有一个可选子句，用于定义在所有情况下都必须要执行的清理操作。例如：

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

如果存在 finally 子句，则 finally 子句是 try 语句结束前执行的最后一项任务。不论 try 语句是否触发异常，都会执行 finally 子句。以下内容介绍了几种比较复杂的触发异常情景：

- 如果执行 try 子句期间触发了某个异常，则某个 except 子句应处理该异常。如果该异常没有 except 子句处理，在 finally 子句执行后会被重新触发。
- except 或 else 子句执行期间也会触发异常。同样，该异常会在 finally 子句执行之后被重新触发。
- 如果 finally 子句中包含 break、continue 或 return 等语句，异常将不会被重新引发。
- 如果执行 try 语句时遇到 break,、continue 或 return 语句，则 finally 子句在执行 break、continue 或 return 语句之前执行。
- 如果 finally 子句中包含 return 语句，则返回值来自 finally 子句的某个 return 语句的返回值，而不是来自 try 子句的 return 语句的返回值。

例如：

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

这是一个比较复杂的例子：

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
```

(下页继续)

(续上页)

```
...         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'
```

如上所示，任何情况下都会执行 `finally` 子句。except 子句不处理两个字符串相除触发的 `TypeError`，因此会在 `finally` 子句执行后被重新触发。

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。

8.8 预定义的清理操作

某些对象定义了不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否成功，都会执行清理操作。比如，下例要打开一个文件，并输出文件内容：

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

这个代码的问题在于，执行完代码后，文件在一段不确定的时间内处于打开状态。在简单脚本中这没有问题，但对于较大的应用程序来说可能会出问题。`with` 语句支持以及时、正确的清理的方式使用文件对象：

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

语句执行完毕后，即使在处理行时遇到问题，都会关闭文件 *f*。和文件一样，支持预定义清理操作的对象会在文档中指出这一点。

8.9 引发和处理多个不相关的异常

在有些情况下，有必要报告几个已经发生的异常。这通常是在并发框架中当几个任务并行失败时的情况，但也有其他的用例，有时需要是继续执行并收集多个错误而不是引发第一个异常。

内置的 `ExceptionGroup` 打包了一个异常实例的列表，这样它们就可以一起被引发。它本身就是一个异常，所以它可以像其他异常一样被捕获。

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> f()
  + Exception Group Traceback (most recent call last):
```

(下页继续)

(续上页)

```
  |   File "<stdin>", line 1, in <module>
  |   File "<stdin>", line 3, in f
  | ExceptionGroup: there were problems
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

通过使用 except* 代替 except ，我们可以有选择地只处理组中符合某种类型的异常。在下面的例子中，显示了一个嵌套的异常组，每个 except* 子句都从组中提取了某种类型的异常，而让所有其他的异常传播到其他子句，并最终被重新引发。

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |   File "<stdin>", line 2, in f
  | ExceptionGroup: group1
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
>>>
```

注意，嵌套在一个异常组中的异常必须是实例，而不是类型。这是因为在实践中，这些异常通常是那些已经被程序提出并捕获的异常，其模式如下:

```
>>> excs = []
... for test in tests:
```

(下页继续)

(续上页)

```
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...     raise ExceptionGroup("Test Failures", excs)
...
```

8.10 用注释细化异常情况

当一个异常被创建以引发时，它通常被初始化为描述所发生错误的信息。在有些情况下，在异常被捕获后添加信息是很有用的。为了这个目的，异常有一个 `add_note(note)` 方法接受一个字符串，并将其添加到异常的注释列表。标准的回溯在异常之后按照它们被添加的顺序呈现包括所有的注释。

```
>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: bad type
Add some information
Add some more information
>>>
```

例如，当把异常收集到一个异常组时，我们可能想为各个错误添加上下文信息。在下文中，组中的每个异常都有一个说明，指出这个错误是什么时候发生的。

```
>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  | ExceptionGroup: We have some problems (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |   File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 1
    +---------------- 2 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |   File "<stdin>", line 2, in f
    | OSError: operation failed
```

(下页继续)

(续上页)

```
    | Happened in Iteration 2
    +---------------- 3 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |   File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 3
    +------------------------------------
>>>
```

CHAPTER 9

类

类提供了把数据和功能绑定在一起的方法。创建新类时创建了新的对象 *类型*，从而能够创建该类型的新 *实例*。实例具有能维持自身状态的属性，还具有能修改自身状态的方法（由其所属的类来定义）。

和其他编程语言相比，Python 的类只使用了很少的新语法和语义。Python 的类有点类似于 C++ 和 Modula-3 中类的结合体，而且支持面向对象编程（OOP）的所有标准特性：类的继承机制支持多个基类、派生的类能覆盖基类的方法、类的方法能调用基类中的同名方法。对象可包含任意数量和类型的数据。和模块一样，类也支持 Python 动态特性：在运行时创建，创建后还可以修改。

如果用 C++ 术语来描述的话，类成员（包括数据成员）通常为 *public*（例外的情况见下文私有变量），所有成员函数都为 *virtual* 。与 Modula-3 中一样，没有用于从对象的方法中引用本对象成员的简写形式：方法函数在声明时，有一个显式的第一个参数代表本对象，该参数由方法调用隐式提供。与在 Smalltalk 中一样，Python 的类也是对象，这为导入和重命名提供了语义支持。与 C++ 和 Modula-3 不同，Python 的内置类型可以用作基类，供用户扩展。此外，与 C++ 一样，具有特殊语法的内置运算符（算术运算符、下标等）都可以为类实例重新定义。

由于缺乏关于类的公认术语，本章中偶尔会使用 Smalltalk 和 C++ 的术语。本章还会使用 Modula-3 的术语，Modula-3 的面向对象语义比 C++ 更接近 Python，但估计听说过这门语言的读者很少。

9.1 名称和对象

对象之间相互独立，多个名称（甚至是多个作用域内的多个名称）可以绑定到同一对象。这在其他语言中通常被称为别名。Python 初学者通常不容易理解这个概念，处理数字、字符串、元组等不可变基本类型时，可以不必理会。但是，对于涉及可变对象（如列表、字典，以及大多数其他类型）的 Python 代码的语义，别名可能会产生意料之外的效果。这样做，通常是为了让程序受益，因为别名在某些方面就像指针。例如，传递对象的代价很小，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，调用者就可以看到更改——无需像 Pascal 那样用两个不同的机制来传参。

9.2 Python 作用域和命名空间

在介绍类前，首先要介绍 Python 的作用域规则。类定义对命名空间有一些巧妙的技巧，了解作用域和命名空间的工作机制有利于加强对类的理解。并且，即便对于高级 Python 程序员，这方面的知识也很有用。

接下来，我们先了解一些定义。

namespace（命名空间）是从名称到对象的映射。现在，大多数命名空间都使用 Python 字典实现，但除非涉及到性能优化，我们一般不会关注这方面的事情，而且将来也可能会改变这种方式。命名空间的例子有：内置名称集合（包括 `abs()` 函数以及内置异常的名称等）；一个模块的全局名称；一个函数调用中的局部名称。对象的属性集合也是命名空间的一种形式。关于命名空间的一个重要知识点是，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 `maximize` 函数，且不会造成混淆。用户使用函数时必须要在函数名前面加上模块名。

点号之后的名称是 **属性**。例如，表达式 `z.real` 中，`real` 是对象 `z` 的属性。严格来说，对模块中名称的引用是属性引用：表达式 `modname.funcname` 中，`modname` 是模块对象，`funcname` 是模块的属性。模块属性和模块中定义的全局名称之间存在直接的映射：它们共享相同的命名空间![1]

属性可以是只读的或者可写的。在后一种情况下，可以对属性进行赋值。模块属性是可写的：你可以写入 `modname.the_answer = 42` 。也可以使用 `del` 语句删除可写属性。例如，`del modname.the_answer` 将从名为 `modname` 对象中移除属性 `the_answer`。

命名空间是在不同时刻创建的，且拥有不同的生命周期。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读取模块定义时创建；通常，模块的命名空间也会持续到解释器退出。从脚本文件读取或交互式读取的，由解释器顶层调用执行的语句是 `__main__` 模块调用的一部分，也拥有自己的全局命名空间。内置名称实际上也在模块里，即 `builtins` 。

函数的局部命名空间在函数被调用时被创建，并在函数返回或抛出未在函数内被处理的异常时，被删除。（实际上，用“遗忘”来描述实际发生的情况会更好一些。）当然，每次递归调用都有自己的局部命名空间。

一个命名空间的 *作用域* 是 Python 代码中的一段文本区域，从这个区域可直接访问该命名空间。“可直接访问”的意思是，该文本区域内的名称在被非限定引用时，查找名称的范围，是包括该命名空间在内的。

作用域虽然是被静态确定的，但会被动态使用。执行期间的任何时刻，都会有 3 或 4 个“命名空间可直接访问”的嵌套作用域：

- 最内层作用域，包含局部名称，并首先在其中进行搜索
- 那些外层闭包函数的作用域，包含“非局部、非全局”的名称，从最靠内层的那个作用域开始，逐层向外搜索。
- 倒数第二层作用域，包含当前模块的全局名称
- 最外层（最后搜索）的作用域，是内置名称的命名空间

如果一个名称被声明为全局，则所有引用和赋值都将直接指向“倒数第二层作用域”，即包含模块的全局名称的作用域。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；如果未使用 nonlocal 声明，这些变量将为只读（尝试写入这样的变量将在最内层作用域中创建一个 *新的* 局部变量，而使得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。在函数之外，局部作用域引用与全局作用域一致的命名空间：模块的命名空间。类定义在局部命名空间内再放置另一个命名空间。

划重点，作用域是按字面文本确定的：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，Python 正在朝着“编译时静态名称解析”的方向发展，因此不要过于依赖动态名称解析！（局部变量已经是被静态确定了。）

Python 有一个特殊规定。如果不存在生效的 `global` 或 `nonlocal` 语句，则对名称的赋值总是会进入最内层作用域。赋值不会复制数据，只是将名称绑定到对象。删除也是如此：语句 `del x` 从局部作用域引用的命名空间中移除对 `x` 的绑定。所有引入新名称的操作都是使用局部作用域：尤其是 `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。

[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。

global 语句用于表明特定变量在全局作用域里，并应在全局作用域中重新绑定；nonlocal 语句表明特定变量在外层作用域中，并应在外层作用域中重新绑定。

9.2.1 作用域和命名空间示例

下例演示了如何引用不同作用域和名称空间，以及 global 和 nonlocal 对变量绑定的影响：

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

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

    def do_global():
        global spam
        spam = "global spam"

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

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

示例代码的输出是：

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

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。

而且，global 赋值前没有 *spam* 的绑定。

9.3 初探类

类引入了一点新语法，三种新的对象类型和一些新语义。

9.3.1 类定义语法

最简单的类定义形式如下：

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

与函数定义 (def 语句) 一样，类定义必须先执行才能生效。把类定义放在 if 语句的分支里或函数内部试试。

在实践中，类定义内的语句通常都是函数定义，但也可以是其他语句。这部分内容稍后再讨论。类里的函数定义一般是特殊的参数列表，这是由方法调用的约定规范所指明的 --- 同样，稍后再解释。

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*。这基本上是一个围绕类定义所创建的命名空间的包装器；我们将在下一节中了解有关类对象的更多信息。原始的 (在进入类定义之前有效的) 作用域将重新生效，类对象将在这里与类定义头所给出的类名称进行绑定 (在这个示例中为 ClassName)。

9.3.2 Class 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):
        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)：

```
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。许多类都希望创建的对象实例是根据特定初始状态定制的。因此一个类可能会定义名为 __init__() 的特殊方法，就像这样:

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

当一个类定义了 __init__() 方法时，类的实例化会自动为新创建的类实例发起调用 __init__()。因此在这个例子中，可以通过以下语句获得一个已初始化的新实例:

```
x = MyClass()
```

当然，__init__() 方法还有一些参数用于实现更高的灵活性。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，

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

9.3.3 实例对象

现在我们能用实例对象做什么？实例对象所能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；就像局部变量一样，它们将在首次被赋值时产生。举例来说，如果 `x` 是上面创建的 `MyClass` 的实例，则以下代码将打印数值 `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，方法并不是必须立即调用: `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())
```

将持续打印 `hello world`，直到结束。

当一个方法被调用时究竟会发生什么？你可能已经注意到尽管 `f()` 的函数定义指定了一个参数，但上面调用 `x.f()` 时却没有带参数。这个参数发生了什么事？当一个需要参数的函数在不附带任何参数的情况下被调用时 Python 肯定会引发异常 --- 即使参数实际上没有被使用...

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，那么看了实现细节可能会弄清楚问题。当对实例对象进行属性引用时，如果该属性在实例中无法找到，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，会打包两者（实例对象和查找到的函数对象）的指针到一个抽象对象，这个抽象对象就是方法对象。当用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 `C` 类的指向函数对象的属性，因此它们都是 `C` 实例的方法 --- 其中 `h` 与 `g` 完全等价。但请注意这种做法通常只会使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该方法的定义语句的模块。（类永远不会被用作全局作用域。）尽管一个人很少会有好的理由在方法中使用全局作用域中的数据，全局作用域依然存在许多合理的使用场景：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，定义在全局作用域中的函数和类也一样。通常，包含该方法的类本身就定义在全局作用域中，而在下一节中我们将会发现，为何有些时候方法需要引用其所属类。

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` 必须定义于可从包含所派生的类的定义的作用域访问的命名空间中。作为基类名称的替代，也允许使用其他任意表达式。例如，当基类定义在另一个模块中时，这就会很有用处:

```
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。

派生类可能会重写其基类的方法。因为方法在调用同一对象的其他方法时没有特殊权限，所以基类方法在尝试调用调用同一基类中定义的另一方法时，可能实际上调用是该基类的派生类中定义的方法。（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 也支持一种多重继承。带有多个基类的类定义语句如下所示:

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

对于多数目的来说，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、从左到右的，当层次结构存在重叠时不会在同一个类中搜索两次。因此，如果某个属性在 DerivedClassName 中找不到，就会在 Base1 中搜索它，然后（递归地）在 Base1 的基类中搜索，如果在那里也找不到，就将在 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 的非公有部分 (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

有时具有类似于 Pascal "record" 或 C "struct" 的数据类型是很有用的，将一些带名称的数据项捆绑在一起。实现这一目标的理想方式是使用 dataclasses:

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

一段期望使用特定抽象数据类型的 Python 代码通常可以通过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() 方法以便从字典串缓冲区获取数据的类，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__ 就是该方法所对应的 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 语句会在容器对象上调用 iter()。该函数返回一个定义了 __next__() 方法的迭代器对象，此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
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
```

(下页继续)

(续上页)

```
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

了解了迭代器协议背后的机制后，就可以轻松地为你的类添加迭代器行为了。定义 __iter__() 方法用于返回一个带有 __next__() 方法的对象。如果类已定义了 __next__()，那么 __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)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

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

9.9 生成器

生成器 是一个用于创建迭代器的简单而强大的工具。它们的写法类似于标准的函数，但当它们要返回数据时会使用 yield 语句。每次在生成器上调用 next() 时，它会从上次离开的位置恢复执行（它会记住上次执行语句时的所有数据值）。一个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任何功能同样可以通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，但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。

示例:

```
>>> 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']
```

备注

CHAPTER 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:\\Python312'
>>> 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，因为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

一般的工具脚本常常需要处理命令行参数。这些参数以列表形式存储在 sys 模块的 *argv* 属性中。举例来说，让我们查看下面的 demo.py 文件:

```
# File demo.py
import sys
print(sys.argv)
```

以下是在命令行中运行 python demo.py one two three 输出的结果:

```
['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://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()             # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())         # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> 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
```

(下页继续)

(续上页)

```
>>> 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` 模块使得实现远程过程调用变成了小菜一碟。尽管存在于模块名称中，但用户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 和其他符合 **RFC 2822** 规范的邮件文档。与 `smtplib` 和 `poplib` 不同（它们实际上做的是发送和接收消息），电子邮件包提供完整的工具集，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，这种格式通常为数据库和电子表格所支持。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。

CHAPTER 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_string("%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 的子类可以自定义分隔符。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))
```

(下页继续)

(续上页)

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

模板的另一个应用是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端字节序:

```
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()
```

(下页继续)

(续上页)

```
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所有对资源的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 日志记录

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

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

这会产生以下输出:

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

默认情况下，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()
```

(下页继续)

(续上页)

```
>>> 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:/python312/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度更高。下面演示了一个用双字节无符号整数数组来储存整数的例子（类型码为 `"H"`），而通常的用 Python 的 int 对象来储存整数的列表，每个表项通常要使用 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` 模块具有用于操作有序列表的函数：

```
>>> 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
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 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')
```

CHAPTER 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:

```
python -m venv tutorial-env
```

这将创建 `tutorial-env` 目录，如果它不存在的话，并在其中创建包含 Python 解释器副本和各种支持文件的目录。

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:

```
tutorial-env\Scripts\activate
```

在 Unix 或 MacOS 上，运行:

```
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。）

激活虚拟环境将改变你所用终端的提示符，以显示你正在使用的虚拟环境，并修改环境以使 `python` 命令所运行的将是已安装的特定 Python 版本。例如：

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

要撤销激活一个虚拟环境，请输入：

```
deactivate
```

到终端。

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 安装软件包。你可以在你的 web 浏览器中查看 Python Package Index。

`pip` 有许多子命令: "install", "uninstall", "freeze" 等等。（请在 installing-index 指南页查看完整的 `pip` 文档。）

您可以通过指定包的名称来安装最新版本的包：

```
(tutorial-env) $ python -m 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) $ python -m 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因而不做任何事。你可以提供不同的版本号来获取相应版本，或者你可以运行 `python -m pip install --upgrade` 以将软件包升级到最新版本：

```
(tutorial-env) $ python -m 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
```

python -m pip uninstall 后跟一个或多个要从虚拟环境中删除的包所对应的名称。

python -m pip show 将显示有关某个特定包的信息:

```
(tutorial-env) $ python -m 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:
```

python -m pip list 将显示所有在虚拟环境中安装的包:

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

python -m pip freeze 将产生一个类似的已安装包列表，但其输出会使用 python -m pip install 所期望的格式。一个常见的约定是将此列表放在 requirements.txt 文件中:

```
(tutorial-env) $ python -m 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) $ python -m 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 有更多的选项。有关 pip 的完整文档请查阅 installing-index 指南。当你编写了一个软件包并希望将其放在 Python Package Index 中时，请查阅 Python packaging user guide。

CHAPTER 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:

- library-index:

  你应当浏览一下本手册，其中提供了有关标准库中的类型、函数和模块的完整（但简洁）的参考资料。标准 Python 分发版包括 许多附加代码。这些模块可以完成读取 Unix 邮箱，通过 HTTP 获取文档，生成随机数，解析命令行选项，压缩数据以及许多其他任务。浏览标准库参考将使你了解有哪些可用的功能。

- installing-index 解释了怎么安装由其他 Python 开发者编写的模块。
- reference-index: 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: Python 主网站。它包含代码、文档和指向全网 Python 相关网页的链接。
- 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。特别值得一看的贡献收集在一本名为 Python Cookbook（O'Reilly & Associates, ISBN 0-596-00797-3）的书中。
- https://pyvideo.org 收集了来自研讨会和用户组会议的 Python 相关视频的链接。
- https://scipy.org： Scientific Python 项目包含用于快速矩阵计算和操作的模块，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被提出的问题及其答案，所以可能已经包含了您的问题解决方案。

[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。

备注

CHAPTER 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变量和模块名补全功能将 自动启用以便在按下 `Tab` 键时发起调用补全函数；它会查找 Python 语句名称、当前局部变量和可用的模块名称。对于带点号的表达式如 `string.a`，它会对该表达式最后一个 `'.'` 之前的部分求值然后根据结果对象的属性给出补全建议。请注意如果具有 `__getattr__()` 方法的对象是该表达式的一部分这可能会执行应用程序定义的代码。默认配置还会将你的编辑历史保存到你的用户目录下名为 `.python_history` 的文件。该历史在下一次交互式解释器会话期间将继续可用。

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。

CHAPTER 15

浮点算术：争议和限制

浮点数在计算机硬件中是以基数为 2 (二进制) 的小数来表示的。例如，**十进制**小数 `0.625` 的值为 6/10 + 2/100 + 5/1000，而同样的 **二进制**小数 `0.101` 的值为 1/2 + 0/4 + 1/8。这两个小数具有相同的值，唯一的区别在于第一个写成了基数为 10 的小数形式，而第二个则写成的基数为 2 的小数形式。

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

```
0.0001100110011001100110011001100110011001100110011001100110011...
```

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 。

由于值的显示方式大多数用户都不会意识到这个差异的存在。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 要把 0.1 的二进制值对应的准确的十进制值打印出来，将会显示成这样:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

这比大多数人认为有用的数位更多，因此 Python 通过改为显示舍入值来保留可管理的数位:

```
>>> 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` 、`0.1000000000000001` 、`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 的值相加也无法恰好得到 0.3:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

而且，由于这个 0.1 无法精确表示 1/10 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(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

虽然这些数字无法精确表示其所要代表的实际值，但是可以使用 `math.isclose()` 函数来进行不精确的值比较:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

或者，也可以使用 `round()` 函数来大致地比较近似程度：

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```

二进制浮点运算会有许多这样令人惊讶的情况。有关”0.1” 的问题会在下面” 表示性错误” 一节中更精确详细地描述。请参阅 Examples of Floating Point Problems 获取针对二进制浮点运算机制及在实践中各种常见问题的概要说明。还可参阅 The Perils of Floating Point 获取其他常见意外现象的更完整介绍。

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 formatstrings 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那么你应当了解一下 NumPy 包以及由 SciPy 项目所提供的许多其他数学和统计运算包。参见 <https://scipy.org>。

Python 还提供了一些工具可能在你确实想要知道一个浮点数的精确值的少数情况下提供帮助。例如 `float.as_integer_ratio()` 方法会将浮点数值表示为一个分数:

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

由于这个比值是精确的，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形式可被用来精确地重建浮点数值:

```
>>> 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 `sum()` 函数，它能够帮助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为中间舍入步骤使用扩展的精度。这可以更好地保持总体精确度，使得错误不会积累到能够影响最终总计值的程度:

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

`math.fsum()` 更进一步地会在数值被添加到总计值的时候跟踪“丢失的数位”以使得结果只执行一次舍入。此函数要比 `sum()` 慢但在大量输入几乎相互抵销导致最终总计值接近零的少见场景中将会更为精确:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...         -143401161400469.7, 266262841.31058735, -0.003244936839808227]
>>> float(sum(map(Fraction, arr)))   # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr)                   # Single rounding
8.042173697819788e-13
>>> sum(arr)                         # Multiple roundings in extended precision
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
...     total += x                   # Multiple roundings in standard precision
```

(下页继续)

(续上页)

```
...
>>> total                                  # Straight addition has no correct digits!
-0.0051575902860057365
```

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。至少从 2000 年起，几乎所有机器都使用 IEEE 754 二进制浮点运算标准，而几乎所有系统平台都将 Python 浮点数映射为 IEEE 754 binary64 ” 双精度” 值。IEEE 754 binary64 值包含 53 位精度，因此在输入时计算机会尽量将 0.1 转换为以 *J*/2**N* 形式所能表示的最接近的小数，其中 *J* 为恰好包含 53 比特位的整数。重新将

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

写为

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

并且由于 *J* 恰好有 53 位 (即 >= 2**52 但 < 2**53)，*N* 的最佳值为 56:

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

也就是说，56 是唯一能使 *J* 恰好有 53 位的 *N* 值。这样 *J* 可能的最佳就是舍入之后的商:

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

由于余数超于 10 的一半，所以最佳近似值可通过向上舍入获得:

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

因此在 IEEE 754 双精度下可能达到的 1/10 的最佳近似值为:

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

分子和分母都除以二则结果小数为:

```
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!

因此计算机永远不会” 看到” 1/10: 它实际看到的就是上面所给出的小数，即它能达到的最佳 IEEE 754 双精度近似值:

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

如果我们将该小数乘以 10**55，我们可以看到该值输出 55 个十进制数位:

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

这意味着存储在计算机中的确切数字等于十进制数值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，而是将结果舍入到 17 位有效数字:

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

`fractions` 和 `decimal` 模块使得这样的计算更为容易:

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

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

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

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

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

CHAPTER 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栈回溯。在交互模式下，将回到主命令提示符，而如果输入内容来自文件，在打印栈回溯之后，程序会以非零的退出状态码退出。（注：这个上下文中所说的错误不包括 `try` 语句中已经被 `except` 子句所捕获的异常。）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某些情况下的内存耗尽。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符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 脚本可以像 shell 脚本一样直接执行，通过在第一行添加：

```
#!/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。注意，“散列字符”，或者说“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。

```
$ 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。

[1] GNU Readline 包的问题可能会阻止这种情况。

16.1.3 交互式启动文件

当您以交互模式使用 Python 时，您可能会希望在每次启动解释器时，解释器先执行几条您预先编写的命令，然后您再以交互模式继续使用。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的文件名来实现。这类似于 Unix shell 的 `.profile` 功能。

Python 只有在交互模式时，才会读取此文件，而非在从脚本读指令或是将 `/dev/tty` 显式作为被运行的 Python 脚本的文件名时（后者反而表现得像一个交互式会话）。这个文件与交互式指令共享相同的命名空间，所以它定义或导入的对象可以在交互式会话中直接使用。您也可以在该文件中更改提示符 `sys.ps1` 和 `sys.ps2`。

如果您想 从当前目录中读取一个额外的启动文件，您可以在上文所说的全局启动文件中编写像 `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())` 这样的代码。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：

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4 定制模块

Python 提供了两个钩子供你进行自定义: sitecustomize 和 usercustomize。要了解它是如何工作的，首先需要找到用户 site-packages 目录的位置。启动 Python 并运行以下代码：

```
>>> import site
>>> site.getusersitepackages()
'/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 的工作方式相同，但通常由计算机管理员在全局 site-packages 目录中创建，并在 usercustomize 之前导入。更多细节请参阅 `site` 模块的文档。

备注

APPENDIX A

术语对照表

>>> 交互式终端中默认的 Python 提示符。往往会显示于能以交互方式在解释器里执行的样例代码之前。

... 具有以下含义：

- 交互式终端中输入特殊代码行时默认的 Python 提示符，包括：缩进的代码块，成对的分隔符之内（圆括号、方括号、花括号或三重引号），或是指定一个装饰器之后。
- `Ellipsis` 内置常量。

2to3 把 Python 2.x 代码转换为 Python 3.x 代码的工具，通过解析源码，遍历解析树，处理绝大多数检测到的不兼容问题。

2to3 包含在标准库中，模块名为 `lib2to3`；提供了独立入口点 `Tools/scripts/2to3`。详见 2to3-reference。

abstract base class -- 抽象基类 抽象基类简称 ABC，是对*duck-typing* 的补充，它提供了一种定义接口的新方式，相比之下其他技巧例如 `hasattr()` 显得过于笨拙或有微妙错误（例如使用 魔术方法）。ABC 引入了虚拟子类，这种类并非继承自其他类，但却仍能被 `isinstance()` 和 `issubclass()` 所认可；详见 `abc` 模块文档。Python 自带许多内置的 ABC 用于实现数据结构（在 `collections.abc` 模块中）、数字（在 `numbers` 模块中）、流（在 `io` 模块中）、导入查找器和加载器（在 `importlib.abc` 模块中）。你可以使用 `abc` 模块来创建自己的 ABC。

annotation -- 标注 关联到某个变量、类属性、函数形参或返回值的标签，被约定作为类型注解 来使用。

局部变量的标注在运行时不可访问，但全局变量、类属性和函数的标注会分别存放模块、类和函数的 `__annotations__` 特殊属性中。

参见*variable annotation*, *function annotation*, **PEP 484** 和 **PEP 526**，对此功能均有介绍。另请参见 annotations-howto 了解使用标注的最佳实践。

argument -- 参数 在调用函数时传给*function* （或*method* ）的值。参数分为两种：

- *关键字参数*: 在函数调用中前面带有标识符（例如 `name=`）或者作为包含在前面带有 `**` 的字典里的值传入。举例来说，`3` 和 `5` 在以下对 `complex()` 的调用中均属于关键字参数:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *位置参数*: 不属于关键字参数的参数。位置参数可出现于参数列表的开头以及/或者作为前面带有 `*` 的*iterable* 里的元素被传入。举例来说，`3` 和 `5` 在以下调用中均属于位置参数:

```
complex(3, 5)
complex(*(3, 5))
```

参数会被赋值给函数体中对应的局部变量。有关赋值规则参见 calls 一节。根据语法，任何表达式都可用来表示一个参数；最终算出的值会被赋给对应的局部变量。

另参见*parameter* 术语表条目，常见问题中 参数与形参的区别以及 **PEP 362**。

asynchronous context manager -- 异步上下文管理器 此种对象通过定义 `__aenter__()` 和 `__aexit__()` 方法来对 `async with` 语句中的环境进行控制。由 **PEP 492** 引入。

asynchronous generator -- 异步生成器 返回值为*asynchronous generator iterator* 的函数。它与使用 `async def` 定义的协程函数很相似，不同之处在于它包含 `yield` 表达式以产生一系列可在 `async for` 循环中使用的值。

此术语通常是指异步生成器函数，但在某些情况下则可能是指 异步生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

一个异步生成器函数可能包含 `await` 表达式或者 `async for` 以及 `async with` 语句。

asynchronous generator iterator -- 异步生成器迭代器 *asynchronous generator* 函数所创建的对象。

此对象属于*asynchronous iterator*，当使用 `__anext__()` 方法调用时会返回一个可等待对象来执行异步生成器函数的函数体直到下一个 `yield` 表达式。

每个 `yield` 会临时暂停处理，记住当前位置执行状态（包括局部变量和挂起的 try 语句）。当该 异步生成器迭代器通过 `__anext__()` 所返回的其他可等待对象有效恢复时，它会从离开位置继续执行。参见 **PEP 492** 和 **PEP 525**。

asynchronous iterable -- 异步可迭代对象 一个可以在 `async for` 语句中使用的对象。必须通过它的 `__aiter__()` 方法返回一个*asynchronous iterator*。由 **PEP 492** 引入。

asynchronous iterator -- 异步迭代器 一个实现了 `__aiter__()` 和 `__anext__()` 方法的对象。`__anext__()` 必须返回一个*awaitable* 对象。`async for` 会处理异步迭代器的 `__anext__()` 方法所返回的可等待对象直到其引发一个 `StopAsyncIteration` 异常。由 **PEP 492** 引入。

attribute -- 属性 关联到一个对象的值，通常使用点号表达式按名称来引用。举例来说，如果对象 *o* 具有属性 *a* 则可以用 *o.a* 来引用它。

如果对象允许，将未被定义为 identifiers 的非标识名称用作一个对象的属性也是可以的，例如使用 `setattr()`。这样的属性将无法使用点号表达式来访问，而是必须通过 `getattr()` 来获取。

awaitable -- 可等待对象 一个可在 `await` 表达式中使用的对象。可以是*coroutine* 或是具有 `__await__()` 方法的对象。参见 **PEP 492**。

BDFL “终身仁慈独裁者”的英文缩写，即 Guido van Rossum，Python 的创造者。

binary file -- 二进制文件 *file object* 能够读写字节类对象。二进制文件的例子包括以二进制模式 (`'rb'`, `'wb'` 或 `'rb+'`) 打开的文件、`sys.stdin.buffer`、`sys.stdout.buffer` 以及 `io.BytesIO` 和 `gzip.GzipFile` 的实例。

另请参见*text file* 了解能够读写 `str` 对象的文件对象。

borrowed reference -- 借入引用 在 Python 的 C API 中，借用引用是指一种对象引用，使用该对象的代码并不持有该引用。如果对象被销毁则它就会变成一个悬空指针。例如，垃圾回收器可以移除对象的最后一个*strong reference* 来销毁它。

推荐在*borrowed reference* 上调用 `Py_INCREF()` 以将其原地转换为*strong reference*，除非是当该对象无法在借入引用的最后一次使用之前被销毁。`Py_NewRef()` 函数可以被用来创建一个新的*strong reference*。

bytes-like object -- 字节类对象 支持 bufferobjects 并且能导出 C-*contiguous* 缓冲的对象。这包括所有 `bytes`、`bytearray` 和 `array.array` 对象，以及许多普通 `memoryview` 对象。字节类对象可在多种二进制数据操作中使用；这些操作包括压缩、保存为二进制文件以及通过套接字发送等。

某些操作需要可变的二进制数据。这种对象在文档中常被称为“可读写字节类对象”。可变缓冲对象的例子包括 `bytearray` 以及 `bytearray` 的 `memoryview`。其他操作要求二进制数据存放于不可变对象 (” 只读字节类对象”)；这种对象的例子包括 `bytes` 以及 `bytes` 对象的 `memoryview`。

bytecode -- 字节码 Python 源代码会被编译为字节码，即 CPython 解释器中表示 Python 程序的内部代码。字节码还会缓存在 `.pyc` 文件中，这样第二次执行同一文件时速度更快（可以免去将源码重新编译为字节码）。这种" 中间语言" 运行在根据字节码执行相应机器码的*virtual machine* 之上。请注意不同 Python 虚拟机上的字节码不一定通用，也不一定能在不同 Python 版本上兼容。

字节码指令列表可以在 dis 模块的文档中查看。

callable -- 可调用对象 可调用对象就是可以执行调用运算的对象，并可能附带一组参数 (参见*argument*)，使用以下语法:

```
callable(argument1, argument2, argumentN)
```

function，还可扩展到*method* 等，就属于可调用对象。实现了 `__call__()` 方法的类的实例也属于可调用对象。

callback -- 回调 一个作为参数被传入以用以在未来的某个时刻被调用的子例程函数。

class -- 类 用来创建用户定义对象的模板。类定义通常包含对该类的实例进行操作的方法定义。

class variable -- 类变量 在类中定义的变量，并且仅限在类的层级上修改 (而不是在类的实例中修改)。

complex number -- 复数 对普通实数系统的扩展，其中所有数字都被表示为一个实部和一个虚部的和。虚数是虚数单位（`-1` 的平方根）的实倍数，通常在数学中写为 `i`，在工程学中写为 `j`。Python 内置了对复数的支持，采用工程学标记方式；虚部带有一个 `j` 后缀，例如 `3+1j`。如果需要 `math` 模块内对象的对应复数版本，请使用 `cmath`，复数的使用是一个比较高级的数学特性。如果你感觉没有必要，忽略它们也几乎不会有任何问题。

context manager -- 上下文管理器 在 `with` 语句中通过定义 `__enter__()` 和 `__exit__()` 方法来控制环境状态的对象。参见 **PEP 343**。

context variable -- 上下文变量 一种根据其所属的上下文可以具有不同的值的变量。这类似于在线程局部存储中每个执行线程可以具有不同的变量值。不过，对于上下文变量来说，一个执行线程中可能会有多个上下文，而上下文变量的主要用途是对并发异步任务中变量进行追踪。参见 `contextvars`。

contiguous -- 连续 一个缓冲如果是 *C 连续*或 *Fortran 连续*就会被认为是连续的。零维缓冲是 C 和 Fortran 连续的。在一维数组中，所有条目必须在内存中彼此相邻地排列，采用从零开始的递增索引顺序。在多维 C-连续数组中，当按内存地址排列时用最后一个索引访问条目时速度最快。但是在 Fortran 连续数组中则是用第一个索引最快。

coroutine -- 协程 协程是子例程的更一般形式。子例程可以在某一点进入并在另一点退出。协程则可以在许多不同的点上进入、退出和恢复。它们可通过 `async def` 语句来实现。参见 **PEP 492**。

coroutine function -- 协程函数 返回一个*coroutine* 对象的函数。协程函数可通过 `async def` 语句来定义，并可能包含 `await`、`async for` 和 `async with` 关键字。这些特性是由 **PEP 492** 引入的。

CPython Python 编程语言的规范实现，在 python.org 上发布。"CPython" 一词用于在必要时将此实现与其他实现例如 Jython 或 IronPython 相区别。

decorator -- 装饰器 返回值为另一个函数的函数，通常使用 `@wrapper` 语法形式来进行函数变换。装饰器的常见例子包括 `classmethod()` 和 `staticmethod()`。

装饰器语法只是一种语法糖，以下两个函数定义在语义上完全等价:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

同样的概念也适用于类，但通常较少这样使用。有关装饰器的详情可参见 函数定义和 类定义的文档。

descriptor -- 描述器 任何定义了 `__get__()`，`__set__()` 或 `__delete__()` 方法的对象。当一个类属性为描述器时，它的特殊绑定行为就会在属性查找时被触发。通常情况下，使用 *a.b* 来获取、设置或删除一个属性时会在 *a* 类的字典中查找名称为 *b* 的对象，但如果 *b* 是一个描述器，则会调用对应

的描述器方法。理解描述器的概念是更深层次理解 Python 的关键，因为这是许多重要特性的基础，包括函数、方法、特征属性、类方法、静态方法以及对超类的引用等等。

有关描述器的方法的更多信息，请参阅 descriptors 或 描述器使用指南。

dictionary -- 字典 一个关联数组，其中的任意键都映射到相应的值。键可以是任何具有 `__hash__()` 和 `__eq__()` 方法的对象。在 Perl 中称为 hash。

dictionary comprehension -- 字典推导式 处理一个可迭代对象中的所有或部分元素并返回结果字典的一种紧凑写法。`results = {n: n ** 2 for n in range(10)}` 将生成一个由键 `n` 到值 `n ** 2` 的映射构成的字典。参见 comprehensions。

dictionary view -- 字典视图 从 `dict.keys()`, `dict.values()` 和 `dict.items()` 返回的对象被称为字典视图。它们提供了字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。要将字典视图强制转换为真正的列表，可使用 `list(dictview)`。参见 dict-views。

docstring -- 文档字符串 作为类、函数或模块之内的第一个表达式出现的字符串字面值。它在代码执行时会被忽略，但会被解释器识别并放入所在类、函数或模块的 `__doc__` 属性中。由于它可用于代码内省，因此是对象存放文档的规范位置。

duck-typing -- 鸭子类型 指一种编程风格，它并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法或属性（“看起来像鸭子，叫起来也像鸭子，那么肯定就是鸭子。”）由于强调接口而非特定类型，设计良好的代码可通过允许多态替代来提升灵活性。鸭子类型避免使用 `type()` 或 `isinstance()` 检测。(但要注意鸭子类型可以使用抽象基类 作为补充。) 而往往会采用 `hasattr()` 检测或是*EAFP* 编程。

EAFP “求原谅比求许可更容易”的英文缩写。这种 Python 常用代码编写风格会假定所需的键或属性存在，并在假定错误时捕获异常。这种简洁快速风格的特点就是大量运用 `try` 和 `except` 语句。于其相对的则是所谓*LBYL* 风格，常见于 C 等许多其他语言。

expression -- 表达式 可以求出某个值的语法单元。换句话说，一个表达式就是表达元素例如字面值、名称、属性访问、运算符或函数调用的汇总，它们最终都会返回一个值。与许多其他语言不同，并非所有语言构件都是表达式。还存在不能被用作表达式的*statement*，例如 `while`。赋值也是属于语句而非表达式。

extension module -- 扩展模块 以 C 或 C++ 编写的模块，使用 Python 的 C API 来与语言核心以及用户代码进行交互。

f-string -- f-字符串 带有 `'f'` 或 `'F'` 前缀的字符串字面值通常被称为“f-字符串”即 格式化字符串字面值的简写。参见 **PEP 498**。

file object -- 文件对象 对外公开面向文件的 API（带有 `read()` 或 `write()` 等方法）以使用下层资源的对象。根据其创建方式的不同，文件对象可以处理对真实磁盘文件、其他类型的存储或通信设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等）。文件对象也被称为 *文件类对象* 或 *流*。

实际上共有三种类别的文件对象: 原始二进制文件, 缓冲二进制文件 以及文本文件。它们的接口定义均在 `io` 模块中。创建文件对象的规范方式是使用 `open()` 函数。

file-like object -- 文件类对象 *file object* 的同义词。

filesystem encoding and error handler -- 文件系统编码格式与错误处理句柄 Python 用来从操作系统解码字节串和向操作系统编码 Unicode 的编码格式与错误处理句柄。

文件系统编码格式必须保证能成功解码长度在 128 以下的所有字节串。如果文件系统编码格式无法提供此保证，则 API 函数可能会引发 `UnicodeError`。

`sys.getfilesystemencoding()` 和 `sys.getfilesystemencodeerrors()` 函数可被用来获取文件系统编码格式与错误处理句柄。

filesystem encoding and error handler 是在 Python 启动时通过 `PyConfig_Read()` 函数来配置的：请参阅 `PyConfig` 的 `filesystem_encoding` 和 `filesystem_errors` 等成员。

另请参见*locale encoding*。

finder -- 查找器 一种会尝试查找被导入模块的*loader* 的对象。

从 Python 3.3 起存在两种类型的查找器: 元路径查找器 配合 `sys.meta_path` 使用，以及*path entry finders* 配合 `sys.path_hooks` 使用。

更多详情可参见 **PEP 302**, **PEP 420** 和 **PEP 451**。

floor division -- 向下取整除法 向下舍入到最接近的整数的数学除法。向下取整除法的运算符是 `//` 。例如，表达式 `11 // 4` 的计算结果是 `2` ，而与之相反的是浮点数的真正除法返回 `2.75` 。注意 `(-11) // 4` 会返回 `-3` 因为这是 `-2.75` 向下舍入得到的结果。见 **PEP 238** 。

function -- 函数 可以向调用者返回某个值的一组语句。还可以向其传入零个或多个参数 并在函数体执行中被使用。另见*parameter*, *method* 和 function 等节。

function annotation -- 函数标注 即针对函数形参或返回值的*annotation* 。

函数标注通常用于类型提示：例如以下函数预期接受两个 `int` 参数并预期返回一个 `int` 值:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

函数标注语法的详解见 function 一节。

参见*variable annotation* 和 **PEP 484**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标的最佳实践。

__future__ future 语句, `from __future__ import <feature>` 指示编译器使用将在未来的 Python 发布版中成为标准的语法和语义来编译当前模块。`__future__` 模块文档记录了可能的 *feature* 取值。通过导入此模块并对其变量求值，你可以看到每项新特性在何时被首次加入到该语言中以及它将（或已）在何时成为默认:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection -- 垃圾回收 释放不再被使用的内存空间的过程。Python 是通过引用计数和一个能够检测和打破循环引用的循环垃圾回收器来执行垃圾回收的。可以使用 `gc` 模块来控制垃圾回收器。

generator -- 生成器 返回一个*generator iterator* 的函数。它看起来很像普通函数，不同点在于其包含 `yield` 表达式以便产生一系列值供给 for-循环使用或是通过 `next()` 函数逐一获取。

通常是指生成器函数，但在某些情况下也可能是指 生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

generator iterator -- 生成器迭代器 *generator* 函数所创建的对象。

每个 `yield` 会临时暂停处理，记住当前位置执行状态（包括局部变量和挂起的 try 语句）。当该 生成器迭代器恢复时，它会从离开位置继续执行（这与每次调用都从新开始的普通函数差别很大）。

generator expression -- 生成器表达式 返回一个迭代器的表达式。它看起来很像普通表达式后面带有定义了一个循环变量、范围的 `for` 子句，以及一个可选的 `if` 子句。以下复合表达式会为外层函数生成一系列值:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function -- 泛型函数 为不同的类型实现相同操作的多个函数所组成的函数。在调用时会由调度算法来确定应该使用哪个实现。

另请参见*single dispatch* 术语表条目、`functools.singledispatch()` 装饰器以及 **PEP 443**。

generic type -- 泛型 可参数化的*type*；通常为 `list` 或 `dict` 这样的 容器类。用于类型提示 和注解。

更多细节参见 泛型别名类型、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模块。

GIL 参见*global interpreter lock*。

global interpreter lock -- 全局解释器锁 *CPython* 解释器所采用的一种机制，它确保同一时刻只有一个线程在执行 Python *bytecode*。此机制通过设置对象模型（包括 `dict` 等重要内置类型）针对并发访问的隐式安全简化了 CPython 实现。给整个解释器加锁使得解释器多线程运行更方便，其代价则是牺牲了在多处理器上的并行性。

不过，某些标准库或第三方库的扩展模块被设计为在执行计算密集型任务如压缩或哈希时释放 GIL。此外，在执行 I/O 操作时也总是会释放 GIL。

创建一个（以更精细粒度来锁定共享数据的）“自由线程”解释器的努力从未获得成功，因为这会牺牲在普通单处理器情况下的性能。据信克服这种性能问题的措施将导致实现变得更复杂，从而更难以维护。

hash-based pyc -- 基于哈希的 pyc 使用对应源文件的哈希值而非最后修改时间来确定其有效性的字节码缓存文件。参见 pyc-invalidation。

hashable -- 可哈希 一个对象如果具有在其生命期内绝不改变的哈希值（它需要有 `__hash__()` 方法），并可以同其他对象进行比较（它需要有 `__eq__()` 方法）就被称为 *可哈希* 对象。可哈希对象必须具有相同的哈希值比较结果才会相等。

可哈希性使得对象能够作为字典键或集合成员使用，因为这些数据结构要在内部使用哈希值。

大多数 Python 中的不可变内置对象都是可哈希的；可变容器（例如列表或字典）都不可哈希；不可变容器（例如元组和 frozenset）仅当它们的元素均为可哈希时才是可哈希的。用户定义类的实例对象默认是可哈希的。它们在比较时一定不相同（除非是与自己比较），它们的哈希值的生成是基于它们的 `id()`。

IDLE Python 的集成开发与学习环境。idle 是 Python 标准发行版附带的基本编辑器和解释器环境。

immutable -- 不可变对象 具有固定值的对象。不可变对象包括数字、字符串和元组。这样的对象不能被改变。如果必须存储一个不同的值，则必须创建新的对象。它们在需要常量哈希值的地方起着重要作用，例如作为字典中的键。

import path -- 导入路径 由多个位置（或*路径条目*）组成的列表，会被模块的*path based finder* 用来查找导入目标。在导入时，此位置列表通常来自 `sys.path`，但对次级包来说也可能来自上级包的 `__path__` 属性。

importing -- 导入 令一个模块中的 Python 代码能为另一个模块中的 Python 代码所使用的过程。

importer -- 导入器 查找并加载模块的对象；此对象既属于*finder* 又属于*loader*。

interactive -- 交互 Python 带有一个交互式解释器，即你可以在解释器提示符后输入语句和表达式，立即执行并查看其结果。只需不带参数地启动 `python` 命令（也可以在你的计算机开始菜单中选择相应菜单项）。在测试新想法或检验模块和包的时候用这种方式会非常方便（请记得使用 `help(x)`）。

interpreted -- 解释型 Python 一是种解释型语言，与之相对的是编译型语言，虽然两者的区别由于字节码编译器的存在而会有所模糊。这意味着源文件可以直接运行而不必显式地创建可执行文件再运行。解释型语言通常具有比编译型语言更短的开发/调试周期，但是其程序往往运行得更慢。参见*interactive*。

interpreter shutdown -- 解释器关闭 当被要求关闭时，Python 解释器将进入一个特殊运行阶段并逐步释放所有已分配资源，例如模块和各种关键内部结构等。它还会多次调用*垃圾回收器*。这会触发用户定义析构器或弱引用回调中的代码执行。在关闭阶段执行的代码可能会遇到各种异常，因为其所依赖的资源已不再有效（常见的例子有库模块或警告机制等）。

解释器需要关闭的主要原因有 `__main__` 模块或所运行的脚本已完成执行。

iterable -- 可迭代对象 一种能够逐个返回其成员项的对象。可迭代对象的例子包括所有序列类型（如 `list`, `str` 和 `tuple` 等）以及某些非序列类型如 `dict`, *文件对象* 以及任何定义了 `__iter__()` 方法或实现了*sequence* 语义的 `__getitem__()` 方法的自定义类的对象。

可迭代对象可被用于 `for` 循环以及许多其他需要一个序列的地方 (`zip()`, `map()`, ...)。当一个可迭代对象作为参数被传给内置函数 `iter()` 时，它会返回该对象的迭代器。这种迭代器适用于对值集合的一次性遍历。在使用可迭代对象时，你通常不需要调用 `iter()` 或者自己处理迭代器对象。`for` 语句会自动为你处理那些操作，创建一个临时的未命名变量用来在循环期间保存迭代器。参见*iterator*, *sequence* 和*generator*。

iterator -- 迭代器 用来表示一连串数据流的对象。重复调用迭代器的 `__next__()` 方法 (或将其传给内置函数 `next()`) 将逐个返回流中的项。当没有数据可用时则将引发 `StopIteration` 异常。到这时迭代器对象中的数据项已耗尽，继续调用其 `__next__()` 方法只会再次引发 `StopIteration`。迭代器必须具有 `__iter__()` 方法用来返回该迭代器对象自身，因此迭代器必定也是可迭代对象，可被用于其他可迭代对象适用的大部分场合。一个显著的例外是那些会多次重复访问迭代项的代码。容器对象 (例如 `list`) 在你每次将其传入 `iter()` 函数或是在 `for` 循环中使用时都会产生一

个新的迭代器。如果在此情况下你尝试用迭代器则会返回在之前迭代过程中被耗尽的同一迭代器对象，使其看起来就像是一个空容器。

更多信息可查看 typeiter。

CPython 实现细节： CPython 没有统一应用迭代器定义 `__iter__()` 的要求。

key function -- 键函数 键函数或称整理函数，是能够返回用于排序或排位的值的可调用对象。例如，`locale.strxfrm()` 可用于生成一个符合特定区域排序约定的排序键。

Python 中有许多工具都允许用键函数来控制元素的排位或分组方式。其中包括 `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` 以及 `itertools.groupby()`。

有多种方式可以创建一个键函数。例如，`str.lower()` 方法可以用作忽略大小写排序的键函数。或者，键函数也可通过 `lambda` 表达式来创建例如 `lambda r: (r[0], r[2])`。此外，`operator.attrgetter()`, `operator.itemgetter()` 和 `operator.methodcaller()` 是键函数的三个构造器。请查看 排序指引来获取创建和使用键函数的示例。

keyword argument -- 关键字参数 参见*argument*。

lambda 由一个单独*expression* 构成的匿名内联函数，表达式会在调用时被求值。创建 lambda 函数的句法为 `lambda [parameters]: expression`

LBYL “先查看后跳跃”的英文缩写。这种代码编写风格会在进行调用或查找之前显式地检查前提条件。此风格与*EAFP* 方式恰成对比，其特点是大量使用 `if` 语句。

在多线程环境中，LBYL 方式会导致“查看”和“跳跃”之间发生条件竞争风险。例如，以下代码 `if key in mapping: return mapping[key]` 可能由于在检查操作之后其他线程从 *mapping* 中移除了 *key* 而出错。这种问题可通过加锁或使用 EAFP 方式来解决。

locale encoding -- 语言区域编码格式 在 Unix 上，它是 LC_CTYPE 语言区域的编码格式。它可以通过 `locale.setlocale(locale.LC_CTYPE, new_locale)` 来设置。

在 Windows 上，它是 ANSI 代码页 (如: `"cp1252"`)。

在 Android 和 VxWorks 上，Python 使用 `"utf-8"` 作为语言区域编码格式。

`locale.getencoding()` 可被用来获取语言区域编码格式。

另请参阅*filesystem encoding and error handler*。

list -- 列表 Python 内置的一种*sequence*。虽然名为列表，但更类似于其他语言中的数组而非链接列表，因为访问元素的时间复杂度为 O(1)。

list comprehension -- 列表推导式 处理一个序列中的所有或部分元素并返回结果列表的一种紧凑写法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 将生成一个 0 到 255 范围内的十六进制偶数对应字符串（0x..）的列表。其中 `if` 子句是可选的，如果省略则 `range(256)` 中的所有元素都会被处理。

loader -- 加载器 负责加载模块的对象。它必须定义名为 `load_module()` 的方法。加载器通常由一个*finder* 返回。详情参见 **PEP 302**，对于*abstract base class* 可参见 `importlib.abc.Loader`。

magic method -- 魔术方法 *special method* 的非正式同义词。

mapping -- 映射 一种支持任意键查找并实现了 `collections.abc.Mapping` 或 `collections.abc.MutableMapping` 抽象基类所规定方法的容器对象。此类对象的例子包括 `dict`, `collections.defaultdict`, `collections.OrderedDict` 以及 `collections.Counter`。

meta path finder -- 元路径查找器 `sys.meta_path` 的搜索所返回的*finder*。元路径查找器与*path entry finders* 存在关联但并不相同。

请查看 `importlib.abc.MetaPathFinder` 了解元路径查找器所实现的方法。

metaclass -- 元类 一种用于创建类的类。类定义包含类名、类字典和基类列表。元类负责接受上述三个参数并创建相应的类。大部分面向对象的编程语言都会提供一个默认实现。Python 的特别之处在于可以创建自定义元类。大部分用户永远不需要这个工具，但当需要出现时，元类可提供强大而优雅的解决方案。它们已被用于记录属性访问日志、添加线程安全性、跟踪对象创建、实现单例，以及其他许多任务。

更多详情参见 metaclasses。

method -- 方法 在类内部定义的函数。如果作为该类的实例的一个属性来调用，方法将会获取实例对象作为其第一个*argument* (通常命名为 `self`)。参见*function* 和*nested scope*。

method resolution order -- 方法解析顺序 方法解析顺序就是在查找成员时搜索全部基类所用的先后顺序。请查看 Python 2.3 方法解析顺序 了解自 2.3 版起 Python 解析器所用相关算法的详情。

module -- 模块 此对象是 Python 代码的一种组织单位。各模块具有独立的命名空间，可包含任意 Python 对象。模块可通过*importing* 操作被加载到 Python 中。

另见*package*。

module spec -- 模块规格 一个命名空间，其中包含用于加载模块的相关导入信息。是 `importlib.machinery.ModuleSpec` 的实例。

MRO 参见*method resolution order*。

mutable -- 可变对象 可变对象可以在其 `id()` 保持固定的情况下改变其取值。另请参见*immutable*。

named tuple -- 具名元组 术语“具名元组”可用于任何继承自元组，并且其中的可索引元素还能使用名称属性来访问的类型或类。这样的类型或类还可能拥有其他特性。

有些内置类型属于具名元组，包括 `time.localtime()` 和 `os.stat()` 的返回值。另一个例子是 `sys.float_info`:

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

有些具名元组是内置类型（例如上面的例子）。此外，具名元组还可通过常规类定义从 `tuple` 继承并定义名称字段的方式来创建。这样的类可以手工编写，或者使用工厂函数 `collections.namedtuple()` 创建。后一种方式还会添加一些手工编写或内置具名元组所没有的额外方法。

namespace -- 命名空间 命名空间是存放变量的场所。命名空间有局部、全局和内置的，还有对象中的嵌套命名空间（在方法之内）。命名空间通过防止命名冲突来支持模块化。例如，函数 `builtins.open` 与 `os.open()` 可通过各自的命名空间来区分。命名空间还通过明确哪个模块实现那个函数来帮助提高可读性和可维护性。例如，`random.seed()` 或 `itertools.islice()` 这种写法明确了这些函数是由 `random` 与 `itertools` 模块分别实现的。

namespace package -- 命名空间包 **PEP 420** 所引入的一种仅被用作子包的容器的*package*，命名空间包可以没有实体表示物，其描述方式与*regular package* 不同，因为它们没有 `__init__.py` 文件。

另可参见*module*。

nested scope -- 嵌套作用域 在一个定义范围内引用变量的能力。例如，在另一函数之内定义的函数可以引用前者的变量。请注意嵌套作用域默认只对引用有效而对赋值无效。局部变量的读写都受限于最内层作用域。类似的，全局变量的读写则作用于全局命名空间。通过 `nonlocal` 关键字可允许写入外层作用域。

new-style class -- 新式类 对目前已被应用于所有类对象的类形式的旧称谓。在较早的 Python 版本中，只有新式类能够使用 Python 新增的更灵活我，如 `__slots__`、描述器、特征属性、`__getattribute__()`、类方法和静态方法等。

object -- 对象 任何具有状态（属性或值）以及预定义行为（方法）的数据。object 也是任何*new-style class* 的最顶层基类名。

package -- 包 一种可包含子模块或递归地包含子包的 Python *module*。从技术上说，包是具有 `__path__` 属性的 Python 模块。

另参见*regular package* 和*namespace package*。

parameter -- 形参 *function* （或方法）定义中的命名实体，它指定函数可以接受的一个*argument* （或在某些情况下，多个实参）。有五种形参：

- *positional-or-keyword*：位置或关键字，指定一个可以作为位置参数 传入也可以作为关键字参数 传入的实参。这是默认的形参类型，例如下面的 *foo* 和 *bar*:

```
def func(foo, bar=None): ...
```

- *positional-only*：仅限位置，指定一个只能通过位置传入的参数。仅限位置形参可通过在函数定义的形参列表中它们之后包含一个 / 字符来定义，例如下面的 *posonly1* 和 *posonly2*:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*：仅限关键字，指定一个只能通过关键字传入的参数。仅限关键字形参可通过在函数定义的形参列表中包含单个可变位置形参或者在多个可变位置形参之前放一个 * 来定义，例如下面的 *kw_only1* 和 *kw_only2*:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*：可变位置，指定可以提供由一个任意数量的位置参数构成的序列（附加在其他形参已接受的位置参数之后）。这种形参可通过在形参名称前加缀 * 来定义，例如下面的 *args*:

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

- *var-keyword*：可变关键字，指定可以提供任意数量的关键字参数（附加在其他形参已接受的关键字参数之后）。这种形参可通过在形参名称前加缀 ** 来定义，例如上面的 *kwargs*。

形参可以同时指定可选和必选参数，也可以为某些可选参数指定默认值。

另参见*argument* 术语表条目、参数与形参的区别中的常见问题、`inspect.Parameter` 类、function 一节以及 **PEP 362**。

path entry -- 路径入口 *import path* 中的一个单独位置，会被*path based finder* 用来查找要导入的模块。

path entry finder -- 路径入口查找器 任一可调用对象使用 `sys.path_hooks` (即*path entry hook*) 返回的*finder*，此种对象能通过*path entry* 来定位模块。

请参看 `importlib.abc.PathEntryFinder` 以了解路径入口查找器所实现的各个方法。

path entry hook -- 路径入口钩子 一种可调用对象，它在知道如何查找特定*path entry* 中的模块的情况下能够使用 `sys.path_hooks` 列表返回一个*path entry finder*。

path based finder -- 基于路径的查找器 默认的一种元路径查找器，可在一个*import path* 中查找模块。

path-like object -- 路径类对象 代表一个文件系统路径的对象。路径类对象可以是一个表示路径的 `str` 或者 `bytes` 对象，还可以是一个实现了 `os.PathLike` 协议的对象。一个支持 `os.PathLike` 协议的对象可通过调用 `os.fspath()` 函数转换为 `str` 或者 `bytes` 类型的文件系统路径；`os.fsdecode()` 和 `os.fsencode()` 可被分别用来确保获得 `str` 或 `bytes` 类型的结果。此对象是由 **PEP 519** 引入的。

PEP “Python 增强提议”的英文缩写。一个 PEP 就是一份设计文档，用来向 Python 社区提供信息，或描述一个 Python 的新增特性及其进度或环境。PEP 应当提供精确的技术规格和所提议特性的原理说明。

PEP 应被作为提出主要新特性建议、收集社区对特定问题反馈以及为必须加入 Python 的设计决策编写文档的首选机制。PEP 的作者有责任在社区内部建立共识，并应将不同意见也记入文档。

参见 **PEP 1**。

portion -- 部分 构成一个命名空间包的单个目录内文件集合（也可能存放于一个 zip 文件内），具体定义见 **PEP 420**。

positional argument -- 位置参数 参见*argument*。

provisional API -- 暂定 API 暂定 API 是指被有意排除在标准库的向后兼容性保证之外的应用编程接口。虽然此类接口通常不会再有重大改变，但只要其被标记为暂定，就可能在核心开发者确定有必要的情况下进行向后不兼容的更改（甚至包括移除该接口）。此种更改并不会随意进行 -- 仅在 API 被加入之前未考虑到的严重基础性缺陷被发现时才可能会这样做。

即便是对暂定 API 来说，向后不兼容的更改也会被视为“最后的解决方案”——任何问题被确认时都会尽可能先尝试找到一种向后兼容的解决方案。

这种处理过程允许标准库持续不断地演进，不至于被有问题的长期性设计缺陷所困。详情见 **PEP 411**。

provisional package -- 暂定包 参见*provisional API*。

Python 3000 Python 3.x 发布路线的昵称（这个名字在版本 3 的发布还遥遥无期的时候就已出现了）。有时也被缩写为“Py3k”。

Pythonic 指一个思路或一段代码紧密遵循了 Python 语言最常用的风格和理念，而不是使用其他语言中通用的概念来实现代码。例如，Python 的常用风格是使用 `for` 语句循环来遍历一个可迭代对象中的所有元素。许多其他语言没有这样的结构，因此不熟悉 Python 的人有时会选择使用一个数字计数器:

```
for i in range(len(food)):
    print(food[i])
```

而相应的更简洁更 Pythonic 的方法是这样的:

```
for piece in food:
    print(piece)
```

qualified name -- 限定名称 一个以点号分隔的名称，显示从模块的全局作用域到该模块中定义的某个类、函数或方法的“路径”，相关定义见 **PEP 3155**。对于最高层级的函数和类，限定名称与对象名称一致:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

当被用于引用模块时，*完整限定名称*意为标示该模块的以点号分隔的整个路径，其中包含其所有的父包，例如 `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count -- 引用计数 对象的引用次数。当对象的引用计数降为零时，它就会被释放。有些对象是“永生”的其引用计数永远不会被修改，因此这种对象永远不会被释放。引用计数对 Python 代码来说通常是不可见的，但它是*CPython* 实现的一个关键元素。程序员可以调用 `sys.getrefcount()` 函数来返回特定对象的引用计数。

regular package -- 常规包 传统型的*package*，例如包含有一个 `__init__.py` 文件的目录。

另参见*namespace package*。

__slots__ 一种写在类内部的声明，通过预先声明实例属性等对象并移除实例字典来节省内存。虽然这种技巧很流行，但想要用好却并不容易，最好是只保留在少数情况下采用，例如极耗内存的应用程序，并且其中包含大量实例。

sequence -- 序列 一种*iterable*，它支持通过 `__getitem__()` 特殊方法来使用整数索引进行高效的元素访问，并定义了一个返回序列长度的 `__len__()` 方法。内置的序列类型有 `list`, `str`, `tuple` 和 `bytes` 等。请注意虽然 `dict` 也支持 `__getitem__()` 和 `__len__()`，但它被归类为映射而非序列，因为它使用任意*immutable* 键而不是整数来查找元素。

collections.abc.Sequence 抽象基类定义了一个更丰富的接口，它在 __getitem__() 和 __len__() 之外，还添加了 count(), index(), __contains__() 和 __reversed__()。实现此扩展接口的类型可以使用 register() 来显式地注册。

set comprehension -- 集合推导式 处理一个可迭代对象中的所有或部分元素并返回结果集合的一种紧凑写法。results = {c for c in 'abracadabra' if c not in 'abc'} 将生成字符串集合 {'r', 'd'}。参见 comprehensions。

single dispatch -- 单分派 一种*generic function* 分派形式，其实现是基于单个参数的类型来选择的。

slice -- 切片 通常只包含了特定*sequence* 的一部分的对象。切片是通过使用下标标记来创建的，在 [] 中给出几个以冒号分隔的数字，例如 variable_name[1:3:5]。方括号（下标）标记在内部使用 slice 对象。

special method -- 特殊方法 一种由 Python 隐式调用的方法，用来对某个类型执行特定操作例如相加等等。这种方法的名称的首尾都为双下划线。特殊方法的文档参见 specialnames。

statement -- 语句 语句是程序段（一个代码“块”）的组成单位。一条语句可以是一个*expression* 或某个带有关键字的结构，例如 if、while 或 for。

static type checker -- 静态类型检查器 读取 Python 代码并进行分析，以查找问题例如拼写错误的外部工具。另请参阅类型提示 以及 typing 模块。

strong reference -- 强引用 在 Python 的 C API 中，强引用是指为持有引用的代码所拥有的对象的引用。在创建引用时可通过调用 Py_INCREF() 来获取强引用而在删除引用时可通过 Py_DECREF() 来释放它。

Py_NewRef() 函数可被用于创建一个对象的强引用。通常，必须在退出某个强引用的作用域时在该强引用上调用 Py_DECREF() 函数，以避免引用的泄漏。

另请参阅*borrowed reference*。

text encoding -- 文本编码格式 在 Python 中，一个字符串是一串 Unicode 代码点（范围为 U+0000--U+10FFFF）。为了存储或传输一个字符串，它需要被序列化为一串字节。

将一个字符串序列化为一个字节序列被称为” 编码”，而从字节序列中重新创建字符串被称为” 解码”。

有各种不同的文本序列化 编码器，它们被统称为” 文本编码格式”。

text file -- 文本文件 一种能够读写 str 对象的*file object*。通常一个文本文件实际是访问一个面向字节的数据流并自动处理*text encoding*。文本文件的例子包括以文本模式（'r' 或 'w'）打开的文件、sys.stdin、sys.stdout 以及 io.StringIO 的实例。

另请参看*binary file* 了解能够读写字节类对象 的文件对象。

triple-quoted string -- 三引号字符串 首尾各带三个连续双引号（”）或者单引号（’）的字符串。它们在功能上与首尾各用一个引号标注的字符串没有什么不同，但是有多种用处。它们允许你在字符串内包含未经转义的单引号和双引号，并且可以跨越多行而无需使用连接符，在编写文档字符串时特别好用。

type -- 类型 类型决定一个 Python 对象属于什么种类；每个对象都具有一种类型。要知道对象的类型，可以访问它的 __class__ 属性，或是通过 type(obj) 来获取。

type alias -- 类型别名 一个类型的同义词，创建方式是把类型赋值给特定的标识符。

类型别名的作用是简化类型注解。例如:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以这样提高可读性:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

参见 typing 和 **PEP 484**，其中有对此功能的详细描述。

type hint -- 类型注解 *annotation* 为变量、类属性、函数的形参或返回值指定预期的类型。

类型提示是可选的而不是 Python 的强制要求，但它们对静态类型检查器 很有用处。它们还能协助 IDE 实现代码补全与重构。

全局变量、类属性和函数的类型注解可以使用 typing.get_type_hints() 来访问，但局部变量则不可以。

参见 typing 和 **PEP 484**，其中有对此功能的详细描述。

universal newlines -- 通用换行 一种解读文本流的方式，将以下所有符号都识别为行结束标志：Unix 的行结束约定 '\n'、Windows 的约定 '\r\n' 以及旧版 Macintosh 的约定 '\r'。参见 **PEP 278** 和 **PEP 3116** 和 bytes.splitlines() 了解更多用法说明。

variable annotation -- 变量标注 对变量或类属性的*annotation*。

在标注变量或类属性时，还可选择为其赋值:

```
class C:
    field: 'annotation'
```

变量标注通常被用作类型注解：例如以下变量预期接受 int 类型的值:

```
count: int = 0
```

变量标注语法的详细解释见 annassign 一节。

参见*function annotation*, **PEP 484** 和 **PEP 526**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标注的最佳实践。

virtual environment -- 虚拟环境 一种采用协作式隔离的运行时环境，允许 Python 用户和应用程序在安装和升级 Python 分发包时不会干扰到同一系统上运行的其他 Python 应用程序的行为。

另参见 venv。

virtual machine -- 虚拟机 一台完全通过软件定义的计算机。Python 虚拟机可执行字节码编译器所生成的*bytecode*。

Zen of Python -- Python 之禅 列出 Python 设计的原则与哲学，有助于理解与使用这种语言。查看其具体内容可在交互模式提示符中输入”import this”。

APPENDIX B

文档说明

这些文档是用 Sphinx 从 reStructuredText 源生成的，*Sphinx* 是一个专为处理 Python 文档而编写的文档生成器。

本文档及其工具链之开发，皆在于志愿者之努力，亦恰如 Python 本身。如果您想为此作出贡献，请阅读 reporting-bugs 了解如何参与。我们随时欢迎新的志愿者！

特别鸣谢：

- Fred L. Drake, Jr.，原始 Python 文档工具集之创造者，众多文档之作者；
- 用于创建 reStructuredText 和 Docutils 套件的 Docutils 项目；
- Fredrik Lundh 的 Alternative Python Reference 项目，Sphinx 从中得到了许多好的想法。

B.1 Python 文档的贡献者

有很多对 Python 语言，Python 标准库和 Python 文档有贡献的人，随 Python 源代码分发的 Misc/ACKS 文件列出了部分贡献者。

有了 Python 社区的输入和贡献，Python 才有了如此出色的文档——谢谢你们！

APPENDIX C

历史和许可证

C.1 该软件的历史

Python 由荷兰数学和计算机科学研究学会（CWI，见 https://www.cwi.nl/ ）的 Guido van Rossum 于 1990 年代初设计，作为一门叫做 ABC 的语言的替代品。尽管 Python 包含了许多来自其他人的贡献，Guido 仍是其主要作者。

1995 年，Guido 在弗吉尼亚州的国家创新研究公司（CNRI，见 https://www.cnri.reston.va.us/ ）继续他在 Python 上的工作，并在那里发布了该软件的多个版本。

2000 年五月，Guido 和 Python 核心开发团队转到 BeOpen.com 并组建了 BeOpen PythonLabs 团队。同年十月，PythonLabs 团队转到 Digital Creations (现为 Zope 公司；见 https://www.zope.org/)。2001 年，Python 软件基金会 (PSF，见 https://www.python.org/psf/) 成立，这是一个专为拥有 Python 相关知识产权而创建的非营利组织。Zope 公司现在是 Python 软件基金会的赞助成员。

所有的 Python 版本都是开源的（有关开源的定义参阅 https://opensource.org/ ）。历史上，绝大多数 Python 版本是 GPL 兼容的；下表总结了各个版本情况。

发布版本	源自	年份	所有者	GPL 兼容？
0.9.0 至 1.2	n/a	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	否
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 及更高	2.1.1	2001 至今	PSF	是

备注： GPL 兼容并不意味着 Python 在 GPL 下发布。与 GPL 不同，所有 Python 许可证都允许您分发修改后的版本，而无需开源所做的更改。GPL 兼容的许可证使得 Python 可以与其它在 GPL 下发布的软件结合使用；但其它的许可证则不行。

感谢众多在 Guido 指导下工作的外部志愿者，使得这些发布成为可能。

C.2 获取或以其他方式使用 Python 的条款和条件

Python 软件和文档的使用许可均基于*PSF 许可协议*。

从 Python 3.8.6 开始，文档中的示例、操作指导和其他代码采用的是 PSF 许可协议和零条款 *BSD 许可* 的双重使用许可。

某些包含在 Python 中的软件基于不同的许可。这些许可会与相应许可之下的代码一同列出。有关这些许可的不完整列表请参阅收录软件的许可与鸣谢。

C.2.1 用于 PYTHON 3.12.1 的 PSF 许可协议

```
1. This LICENSE AGREEMENT is between the Python Software Foundation␣
↪("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise␣
↪using Python
   3.12.1 software in source or binary form and its associated␣
↪documentation.

2. Subject to the terms and conditions of this License Agreement, PSF␣
↪hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative␣
↪works,
   distribute, and otherwise use Python 3.12.1 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's␣
↪notice of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.12.1 alone or in any derivative␣
↪version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.12.1 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made␣
↪to Python
   3.12.1.

4. PSF is making Python 3.12.1 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY␣
↪OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY␣
↪REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR␣
↪THAT THE
   USE OF PYTHON 3.12.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.12.1
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
```

```
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.12.1, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

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

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee. ␣
↪This License
   Agreement does not grant permission to use PSF trademarks or trade name␣
↪in a
   trademark sense to endorse or promote products or services of Licensee,␣
↪or any
   third party.

8. By copying, installing or otherwise using Python 3.12.1, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议

BEOPEN PYTHON 开源许可协议第 1 版

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

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

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

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

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

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
```

(下页继续)

(续上页)

```
   granted on that web page.

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

C.2.3 用于 PYTHON 1.6.1 的 CNRI 许可协议

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

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

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

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

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

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

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

(下页继续)

(续上页)

```
   sense to endorse or promote products or services of Licensee, or any third
   party.

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

C.2.4 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议

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

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

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

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.12.1 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

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

C.3 收录软件的许可与鸣谢

本节是 Python 发行版中收录的第三方软件的许可和致谢清单，该清单是不完整且不断增长的。

C.3.1 Mersenne Twister

作为 random 模块下层的 _random C 扩展包括基于从 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 下载的代码。以下是原始代码的完整注释:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 套接字

socket 使用了 getaddrinfo() 和 getnameinfo() WIDE 项目的不同源文件中: https://www.wide.ad.jp/

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
```

(下页继续)

(续上页)

```
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 异步套接字服务

`test.support.asynchat` 和 `test.support.asyncore` 模块包含以下说明。:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

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

C.3.4 Cookie 管理

`http.cookies` 模块包含以下声明:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

                All Rights Reserved

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

(下页继续)

(续上页)

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

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

C.3.5 执行追踪

trace 模块包含以下声明:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 与 UUdecode 函数

uu 模块包含以下声明:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
```

(下页继续)

(续上页)

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

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 远程过程调用

`xmlrpc.client` 模块包含以下声明:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test.test_epoll` 模块包含以下说明:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
```

(下页继续)

(续上页)

```
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 模块关于 kqueue 的接口包含以下声明:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

`Python/pyhash.c` 文件包含 Marek Majkowski' 对 Dan Bernstein 的 SipHash24 算法的实现。它包含以下声明:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
```

(下页继续)

(续上页)

```
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 和 dtoa

Python/dtoa.c 文件提供了 C 函数 dtoa 和 strtod，用于 C 双精度数值和字符串之间的转换，它派生自由 David M. Gay 编写的同名文件。目前该文件可在 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 访问。在 2009 年 3 月 16 日检索到的原始文件包含以下版权和许可声明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

如果操作系统提供支持，则 hashlib, posix, ssl, crypt 模块会使用 OpenSSL 库来提高性能。此外，Python 的 Windows 和 macOS 安装程序可能会包括 OpenSSL 库的副本，所以我们也在此包括一份 OpenSSL 许可证的副本。对于 OpenSSL 3.0 版及其后续衍生版本，均使用 Apache 许可证 v2:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
```

(下页继续)

(续上页)

```
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
```

(下页继续)

(续上页)

```
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.
```

(下页继续)

(续上页)

```
6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

pyexpat 扩展是使用所包括的 expat 源副本来构建的，除非配置了 --with-system-expat:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
```

(下页继续)

(续上页)

```
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

作为 ctypes 模块下层的 _ctypes C 扩展是使用包括了 libffi 源的副本构建的，除非构建时配置了 --with-system-libffi:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

如果系统上找到的 zlib 版本太旧而无法用于构建，则使用包含 zlib 源代码的拷贝来构建 zlib 扩展:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.
```

(下页继续)

(续上页)

```
Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

tracemalloc 使用的哈希表的实现基于 cfuhash 项目:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

作为 decimal 模块下层的 _decimal C 扩展是使用包括了 libmpdec 库的副本构建的，除非构建时配置了 --with-system-libmpdec:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
```

(下页继续)

(续上页)

```
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 测试套件

`test` 包中的 C14N 2.0 测试集 (`Lib/test/xmltestdata/c14n-20/`) 提取自 W3C 网站 https://www.w3.org/TR/xml-c14n2-testcases/ 并根据 3 条款版 BSD 许可证发行:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 audioop

audioop 模块使用 SoX 项目的 g771.c 文件中的代码为基础:

```
Programming the AdLib/Sound Blaster
FM Music Chips
Version 2.0 (24 Feb 1992)
Copyright (c) 1991, 1992 by Jeffrey S. Lee
jlee@smylex.uucp
Warranty and Copyright Policy
This document is provided on an "as-is" basis, and its author makes
no warranty or representation, express or implied, with respect to
its quality performance or fitness for a particular purpose.  In no
event will the author of this document be liable for direct, indirect,
special, incidental, or consequential damages arising out of the use
or inability to use the information contained within.  Use of this
document is at your own risk.
This file may be used and copied freely so long as the applicable
copyright notices are retained, and no modifications are made to the
text of the document.  No money shall be charged for its distribution
beyond reasonable shipping, handling and duplication costs, nor shall
proprietary changes be made to this document so that it cannot be
distributed freely.  This document may not be included in published
material or commercial packages without the written consent of its
author.
```

APPENDIX D

版权所有

Python 与这份文档：

有关完整的许可证和许可信息，请参见历史和许可证。

索引

非字母

. . ., **111**
(*hash*)
    注释, 9
* (星号)
    在函数调用中, 29
**
    在函数调用中, 29
2to3, **111**
: (冒号)
    函数标注, 31
->
    函数标注, 31
>>>, **111**
__all__, 51
__future__, **115**
__slots__, **120**
特殊
    method -- 方法, 121
环境变量
    PATH, 47, 109
    PYTHONPATH, 47, 48
    PYTHONSTARTUP, 110
编码
    style, 31
魔术
    method -- 方法, 117

A

abstract base class -- 抽象基类, **111**
annotation -- 标注, **111**
annotations
    function -- 函数, 31
argument -- 参数, **111**
asynchronous context manager -- 异步上下文管理器, **112**
asynchronous generator -- 异步生成器, **112**
asynchronous generator iterator -- 异步生成器迭代器, **112**
asynchronous iterable -- 异步可迭代对象, **112**
asynchronous iterator -- 异步迭代器, **112**
attribute -- 属性, **112**
awaitable -- 可等待对象, **112**

B

BDFL, **112**
binary file -- 二进制文件, **112**
borrowed reference -- 借入引用, **112**
builtins
    module, 49
bytecode -- 字节码, **113**
bytes-like object -- 字节类对象, **112**

C

C 连续, 113
callable -- 可调用对象, **113**
callback -- 回调, **113**
class -- 类, **113**
class variable -- 类变量, **113**
complex number -- 复数, **113**
context manager -- 上下文管理器, **113**
context variable -- 上下文变量, **113**
contiguous -- 连续, **113**
coroutine -- 协程, **113**
coroutine function -- 协程函数, **113**
CPython, **113**

D

decorator -- 装饰器, **113**
descriptor -- 描述器, **113**
dictionary -- 字典, **114**
dictionary comprehension -- 字典推导式, **114**
dictionary view -- 字典视图, **114**
docstring -- 文档字符串, **114**
duck-typing -- 鸭子类型, **114**

E

EAFP, **114**
expression -- 表达式, **114**
extension module -- 扩展模块, **114**

F

f-string -- f-字符串, **114**

file object -- 文件对象, **114**
file-like object -- 文件类对象, **114**
filesystem encoding and error handler -- 文件系统编码格式与错误处理句柄, **114**
finder -- 查找器, **114**
floor division -- 向下取整除法, **115**
for
    statement -- 语句, 18
Fortran 连续, 113
function -- 函数
    annotations, 31
function -- 函数, **115**
function annotation -- 函数标注, **115**

G

garbage collection -- 垃圾回收, **115**
generator -- 生成器, 115
generator -- 生成器, **115**
generator expression -- 生成器表达式, 115
generator expression -- 生成器表达式, **115**
generator iterator -- 生成器迭代器, **115**
generic function -- 泛型函数, **115**
generic type -- 泛型, **115**
GIL, **115**
global interpreter lock -- 全局解释器锁, **115**

H

hash-based pyc -- 基于哈希的 pyc, **116**
hashable -- 可哈希, **116**
help
    内置函数, 83

I

IDLE, **116**
immutable -- 不可变对象, **116**
import path -- 导入路径, **116**
importer -- 导入器, **116**
importing -- 导入, **116**
interactive -- 交互, **116**
interpreted -- 解释型, **116**
interpreter shutdown -- 解释器关闭, **116**
iterable -- 可迭代对象, **116**
iterator -- 迭代器, **116**

J

json
    module, 59

K

key function -- 键函数, **117**
keyword argument -- 关键字参数, **117**

L

lambda, **117**
LBYL, **117**
list -- 列表, **117**
list comprehension -- 列表推导式, **117**
loader -- 加载器, **117**
locale encoding -- 语言区域编码格式, **117**

M

magic method -- 魔术方法, **117**
mapping -- 映射, **117**
meta path finder -- 元路径查找器, **117**
metaclass -- 元类, **117**
method -- 方法
    object -- 对象, 75
    特殊, 121
    魔术, 117
method -- 方法, **118**
method resolution order -- 方法解析顺序, **118**
module
    builtins, 49
    json, 59
    sys, 48
    搜索 path, 47
module -- 模块, **118**
module spec -- 模块规格, **118**
MRO, **118**
mutable -- 可变对象, **118**

N

name
    扭曲, 79
named tuple -- 具名元组, **118**
namespace -- 命名空间, **118**
namespace package -- 命名空间包, **118**
nested scope -- 嵌套作用域, **118**
new-style class -- 新式类, **118**

O

object -- 对象
    method -- 方法, 75
    文件, 57
object -- 对象, **118**
open
    内置函数, 57

P

package -- 包, **118**
parameter -- 形参, **118**
PATH, 47, 109
path
    module 搜索, 47
path based finder -- 基于路径的查找器, **119**
path entry -- 路径入口, **119**
path entry finder -- 路径入口查找器, **119**
path entry hook -- 路径入口钩子, **119**
path-like object -- 路径类对象, **119**
PEP, **119**

portion -- 部分, **119**
positional argument -- 位置参数, **119**
provisional API -- 暂定 API, **119**
provisional package -- 暂定包, **120**
Python 3000, **120**
Python 提高建议
    PEP 1, 119
    PEP 8, 31
    PEP 238, 115
    PEP 278, 122
    PEP 302, 115, 117
    PEP 343, 113
    PEP 362, 112, 119
    PEP 411, 120
    PEP 420, 115, 118, 119
    PEP 443, 115
    PEP 451, 115
    PEP 483, 115
    PEP 484, 31, 111, 115, 122
    PEP 492, 112, 113
    PEP 498, 114
    PEP 519, 119
    PEP 525, 112
    PEP 526, 111, 122
    PEP 585, 115
    PEP 636, 23
    PEP 3107, 31
    PEP 3116, 122
    PEP 3147, 48
    PEP 3155, 120
Pythonic, **120**
PYTHONPATH, 47, 48
PYTHONSTARTUP, 110

Q

qualified name -- 限定名称, **120**

R

reference count -- 引用计数, **120**
regular package -- 常规包, **120**
RFC
    RFC 2822, 88

S

sequence -- 序列, **120**
set comprehension -- 集合推导式, **121**
single dispatch -- 单分派, **121**
sitecustomize, 110
slice -- 切片, **121**
special method -- 特殊方法, **121**
statement -- 语句
    for, 18
statement -- 语句, **121**
static type checker -- 静态类型检查器, **121**
strong reference -- 强引用, **121**
style
    编码, 31
sys
    module, 48

T

text encoding -- 文本编码格式, **121**
text file -- 文本文件, **121**
triple-quoted string -- 三引号字符串, **121**
type -- 类型, **121**
type alias -- 类型别名, **121**
type hint -- 类型注解, **122**

U

universal newlines -- 通用换行, **122**
usercustomize, 110

V

variable annotation -- 变量标注, **122**
内置函数
    help, 83
    open, 57
virtual environment -- 虚拟环境, **122**
virtual machine -- 虚拟机, **122**
字符串, 文档, 23, 30

W

扭曲
    name, 79
搜索
    path, module, 47
文件
    object -- 对象, 57
文档字符串, 23, 30

Z

Zen of Python -- Python 之禅, **122**