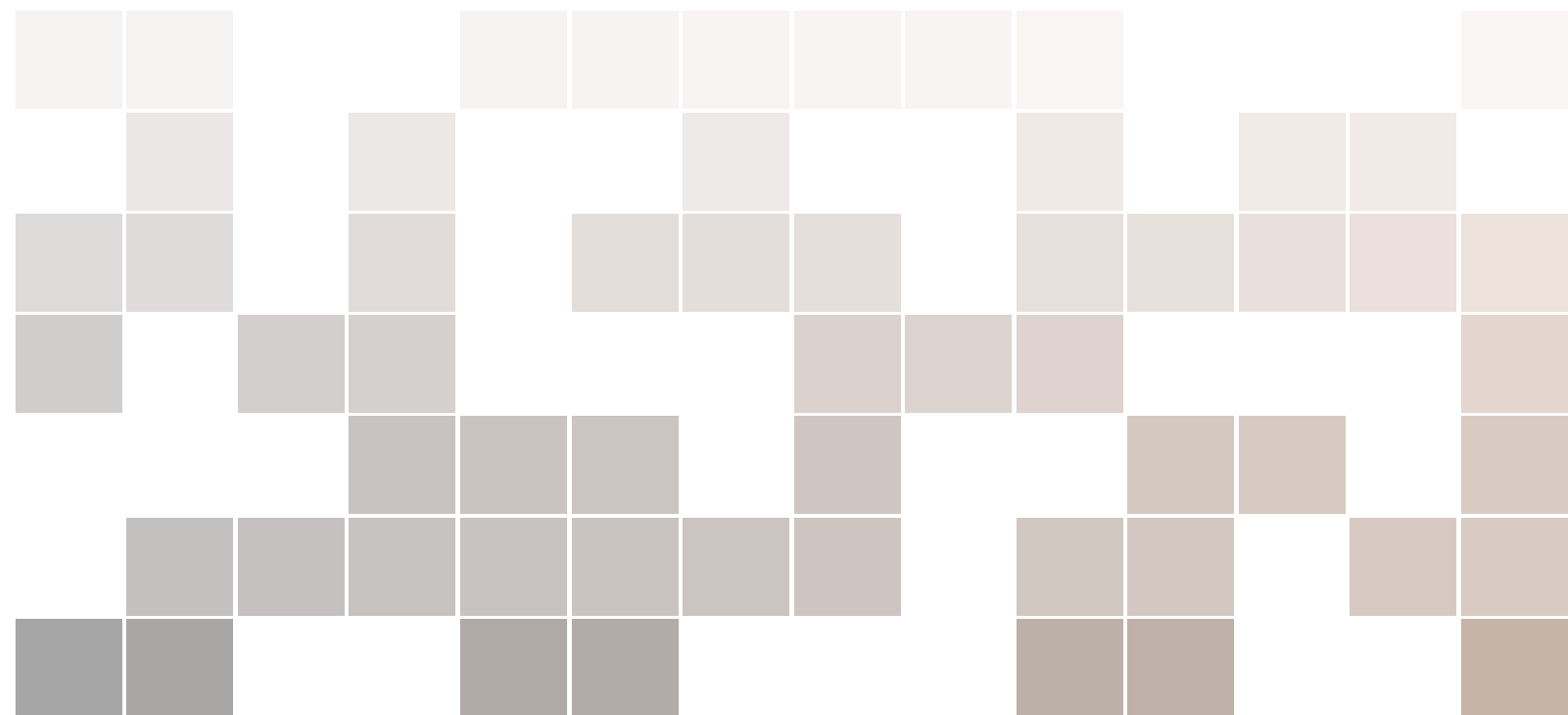


Python 入门教程

Release: 3.2.2

docspy3zh.readthedocs.org

2013 年 10 月 28 日



目录

第一章 开胃菜	4	4.4 break 和 continue 语句, 以及循环中的 else 子句	29
第二章 使用 Python 解释器	7	4.5 pass 语句	30
2.1 调用 Python 解释器	7	4.6 定义函式	30
2.1.1 参数传递	8	4.7 深入函式定义	33
2.1.2 交互模式	9	4.7.1 默认参数	33
2.2 解释器及其环境	10	4.7.2 关键字参数	34
2.2.1 错误处理	10	4.7.3 任意参数表	36
2.2.2 可执行的 Python 脚本	10	4.7.4 释放参数列表	37
2.2.3 源程序编码	11	4.7.5 Lambda 形式	38
2.2.4 交互式启动文件	11	4.7.6 文档字符串	38
2.2.5 定制模块	12	4.8 插曲: 代码风格	39
第三章 非正式介绍 Python	12	第五章 数据结构	40
3.1 把 Python 当计算器使用	13	5.1 深入列表	40
3.1.1 数值	13	5.1.1 把列表当成堆栈用	42
3.1.2 字符串	16	5.1.2 把列表当队列使用	43
3.1.3 关于 Unicode	21	5.1.3 列表推导式	43
3.1.4 列表	22	5.1.4 嵌套列表推导式	45
3.2 编程第一步	24	5.2 del 语句	46
第四章 深入流程控制	26	5.3 元组和序列	46
4.1 if 语句	26	5.4 集合 (Set)	48
4.2 for 语句	26	5.5 字典	49
4.3 range() 函式	27	5.6 遍历技巧	50
		5.7 深入条件控制	52

5.8 序列和其它类型的比较	53	8.6 定义清理动作	81
第六章 模块	54	8.7 预定义的清理动作 . . .	83
6.1 深入模块	55	第九章 类	83
6.1.1 把模块当脚本		9.1 关于名称和对象的讨论	84
执行	56	9.2 Python 的作用域和命名空间	84
6.1.2 模块搜索路径 .	57	9.2.1 域和命名空间的例子	87
6.1.3 “已编译”的 Python 文件 .	57	9.3 类的初印象	88
6.2 标准模块	59	9.3.1 类定义的语法 .	88
6.3 dir() 函数	59	9.3.2 类对象	88
6.4 包	61	9.3.3 实例对象 . . .	90
6.4.1 从包中导入 * .	64	9.3.4 方法对象 . . .	90
6.4.2 内部包参考 . .	65	9.4 随机备注	91
6.4.3 多目录的包 . .	65	9.5 继承	93
第七章 输入和输出	66	9.5.1 多重继承 . . .	94
7.1 美化输出格式	66	9.6 私有变量	95
7.1.1 旧式字符串格式化	70	9.7 杂物	96
7.2 读和写文件	71	9.8 异常也是类	96
7.2.1 文件对象的方法	71	9.9 迭代器	97
7.2.2 pickle 模块 .	74	9.10 发生器	99
第八章 错误和异常	75	9.11 生成器表达式	100
8.1 语法错误	75	第十章 标准库简明介绍	101
8.2 异常	75	10.1 与操作系统的接口 . .	101
8.3 处理异常	76	10.2 文件的通配符	102
8.4 抛出异常	79	10.3 命令行参数	102
8.5 自定义异常	80	10.4 错误的重定向输出和程序的终止	102

10.5 字符串模式的区配 . . .	103	11.6 弱引用	113
10.6 数学处理	103	11.7 处理列表的工具	114
10.7 访问互联网	104	11.8 十进制浮点数的运算 . . .	116
10.8 日期和时间	104		
10.9 数据的压缩	105	第十二章 现在干什么?	117
10.10 性能测试	105	第十三章 交互式输入编辑及历史替代	119
10.11 质量控制	106	13.1 行编辑	119
10.12 充电区	107	13.2 历史替代	119
		13.3 按键绑定	120
第十一章 标准库简明介绍 (第二部分)	108	13.4 交互式解释器的替代品 . . .	122
11.1 格式化输出	108		
11.2 模板化	109	第十四章 浮点算数: 问题和限制	122
11.3 Working with Binary Data Record Layouts . . .	111	14.1 表示错误	127
11.4 多线程	111		
11.5 日志	113	A 使用 Sphinx 输出包含中文支持的 L^AT_EX 源文件	131

Python 是种易学而强大的编程语言. 它包含了高效的高级数据结构, 能够用简单而高效的方式进行面向对象编程. Python 优雅的语法和动态类型, 以及它天然的解释能力, 使其成为了大多数平台上能广泛适用于各领域的理想脚本语言和开发环境.

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

Python 的解释器可以很容易的通过 C 或者 C++ (或者其它可以通过 C 调用的语言) 扩展新的函式和数据类型. Python 也可以作为定制应用的扩展语言.

本教程向读者介绍 Python 语言及其体系的基本知识与概念. 配合 Python 解释器学习会很有帮助, 因为文中已包含所有的完整例子, 所以这本手册也可以离线阅读.

需要有关标准对象和模块的详细介绍的话, 请参阅[库参考手册](#). 而[语言手册](#)提供了更多关于语言本身的正式说明. 需要编写 C 或 C++ 扩展, 请阅读[扩展和嵌入](#)以及对 [C 接口](#). 这几部分涵盖了 Python 各领域的深入知识.

本教程没有涵盖 Python 的所有功能, 也不准备解释涉及的所有相关知识. 相反的, 只介绍 Python 中最引人注目的功能, 这对读者掌握这门语言的风格大有帮助. 读完后, 你应该已能阅读和编写 Python 模块和程序, 接下去就可以从 Python [库参考手册](#)中进一步学习 Python 丰富库和模块.

[术语表](#)也值得仔细阅读.

第一章 开胃菜¹

假如你用计算机做许多工作, 最终你会发现有些工作你会希望计算机能自动完成. 例如: 以复杂的方式重命名并整理大量图片. 又或想要编写一个小型的定制数据库, 又或一个特殊的图形界面程序, 甚或一个小型的游

¹初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

戏。

如果你是名专业的软件开发者, 你可能不得不动用很多 C/C++/Java 库, 结果编写/编译/测试/重编译周期越来越长. 可能你需要给每个库编写对应的测试代码, 但发现这是个烦人的活儿. 或者你在编写的程序能使用一种扩展语言, 但你又不想整个重新设计和实现一回.

这时, Python 正是你要的语言.

或许你可以写一个 Unix shell 脚本抑或是 Windows 批处理文件来完成上述的一些工作, 但是脚本语言最擅长的是移动文件和对文本文件进行处理, 而在图形界面程序或游戏方面并不擅长. 你能写 C/C++/Java 程序, 但是这些技术即使开发最简单的程序也要用去大量的时间. 无论在 Windows、MacOS X 或 Unix 操作系统上, Python 都非常易于使用, 可以帮助你更快的完成任务.

虽说 Python 很容易上手, 但它毕竟是一门真正的编程语言, 相对于 Shell 脚本或是批处理文件, 对大型程序的和数据结构的支持, Python 要多的多. 另一方面, 它提供了比 C 更多的错误检查, 而且, 作为一门非常高阶的语言 (*very-high-level language*), 它拥有内置高级数据结构类型, 例如可变数组和字典. 因为 Python 拥有更多的通用数据类型, 因此它相较 Awk 甚至是 Perl, 在更广泛的问题领域内有用武之地, 而且在许多事情上 Python 表现的至少不比那些语言复杂.

Python 允许你把自己的程序分隔成不同的模块, 以便在其它的 Python 程序中重用. Python 自带一个很大的标准模块集, 你应该把它们作为自己程序的基础——或者把它们做为开始学习 Python 时的编程实例. 其中一些模块中提供了诸如文件 I/O, 系统调用, sockets 甚至类似 TK 这样的图形接口.

Python 是一门解释型语言, 因为不需要编译和链接的时间, 它可以帮你省下一些开发时间. 解释器可以交互式的使用, 这使你很容易实验用各种语言特征, 写可抛弃的程序, 或在自下而上的开发期间测试功能. 它也是一个随手可得的计算器.

Python 能让程序紧凑, 可读性增强. 用 Python 写的程序通常比同样



的 C, C++ 或 Java 程序要短得多, 这是因为以下几个原因:

- 高级数据结构使你可以在单独的语句中也能表述复杂的操作;
- 语句的组织依赖于缩进而不是开始/结束符 (类似 C 族语言的 {} 符号或 Pascal 的 begin/end 关键字);
- 参数或变量不需要声明.



Note:

有关 Python 使用缩进来进行语法结构控制的特性, 这是在技术社区中经常引发争论的一点, 习惯用标识符的程序猿有诸多怨辞; 从译者看来这正是 Python 最可爱的一点:

- 精确体现出了内置的简单就是美的精神追求:
- 不得写出难以看懂的代码!
- 因为使用了空间上的缩进, 所以:
 - 超过 3 层的编辑结构, 会导致代码行超出屏幕, 难以阅读
 - 团队中各自使用不同的缩进方式时, 难以阅读其它人的代码
 - 超过一屏高度的代码, 将难以理解层次关系
 - ...
- 那么这也意味着:
 - 你忽然开始享受人类最优雅和 NB 的编辑器了
 - 你的所有函式都小于 50 行, 简洁明了
 - 你所在的团队有稳定统一的代码规约了, 你看任何人的代码都没有反胃的感觉了
 - ...

Python 是可扩展的: 如果你会用 C 写程序, 就可以很容易的为解释器添加新的内建函式或模块, 或者优化性能瓶颈, 使其达到最大速度, 或者使 Python 能够链接到必要的二进制架构 (比如某个专用的商业图形库). 一旦

你真正掌握了, 你可以将 Python 集成进由 C 写成的程序, 把 Python 当做是这个程序的扩展或命令行语言.

顺便说一下, 这个语言的名字来自于 BBC 的 “Monty Python’s Flying Circus” 节目, 和凶猛的爬行类生物没有任何关系. 在文档中引用 Monty Python 的典故不仅可行, 而且值得鼓励!

现在我们已经了解了 Python 中所有激动人心的东西, 大概你想详细尝试一下了. 的确, 学习一门语言最好的办法就是使用它, 如你所读到的, 教程将邀请你在 Python 解释环境中进行试练.

下一节, 将先说明解释器的用法, 这没有什么神秘的内容, 不过有助于我们练习后面展示的例子.

本教程其它部分通过示例介绍了 Python 语言和系统的各种功能, 开始是简单表达式, 语法和数据类型, 接下来是函式和模块, 最后是诸如异常和自定义类这样的高级内容.

第二章 使用 Python 解释器²

2.1 调用 Python 解释器

Python 解释器通常安装在目标机器的 `/usr/local/bin/` 目录下. 将 `/usr/local/bin` 目录放进你的 Unix Shell 的搜索路径里, 确保它可以通过输入:

```
1 python3.2
```

启动.³ 因为安装路径是可选的, 所以也可能安装在其它位置, 具体的你可以咨询当地 Python 导师或系统管理员. (例如, `/usr/local/python` 就是一个很常见的选择)

在 Windows 机器中, Python 通常安装在 `C:\Python32` 目录. 当然, 我们可以在运行安装程序的时候改变它. 需要把这个目录加入到我们的 Path


²初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

³在 Unix, Python 3.x 解释器默认不使用可执行文件名 `python` 安装, 所以同时安装 Python 2.x 并不会发生冲突.

中的话, 可以像下面这样在 DOS 窗中输入命令行:

```
set path=%path%;C:\python32
```

输入一个文件结束符 (UNIX 上是 Control-D , Windows 上是 Control-Z) 解释器会以 0 值退出. 如果没有起作用, 你可以输入以下命令退出: `quit()`.

解释器的行编辑功能并不复杂, 装在 UNIX 上的解释器可能需要 `G`  `readline` 库支持, 这样就可以额外得到精巧的交互编辑和历史记录功能. 确认命令行编辑器支持能力最方便的方式可能是在主提示符下输入 Control-P, 如果有嘟嘟声 (计算机扬声器), 说明你可以使用命令行编辑功能 (在交互式输入编辑及历史替代在快捷键的介绍). 如果什么也没有发声, 或者显示 ^P, 说明命令行编辑功能不可用; 你将只能用退格键删除当前行的字符.

解释器的操作有些像 UNIX Shell: 使用终端设备作为标准输入来调用它时, 解释器交互地解读和执行命令, 通过文件名参数或以文件作为标准输入时, 它从文件中解读并执行脚本.

启动解释器的第二个方法是 `python -c command [arg] ...`, 这种方法会执行 `command` 中的语句, 等同于 Shell 的 `-c` 选项. 因为 Python 语句中通常会包括空格之类的对 shell 有特殊含义的字符, 所以最好把整个 `command` 用单引号包起来.

有一些 Python 模块也可以当作脚本使用. 它们可以通过 `python -m module [arg] ...` 调用, 这如同在命令行中给出其完整文件名来运行一样.

使用脚本文件时, 经常会运行脚本然后进入交互模式. 这也可以通过在脚本之前加上 `-i` 参数来实现. (如果脚本来自标准输入, 就不能这样执行, 与前述提及原因一样.)

2.1.1 参数传递

调用解释器时, 脚本名和附加参数传入一个名为 `sys.argv` 的字符串列表.

- 没有给定脚本和参数时, 它至少有一个元素: `sys.argv[0]`, 此时它是一个空字符串,
- 脚本名指定为 `'-'` (表示标准输入) 时, `sys.argv[0]` 被设为 `'-'`.
- 使用 `-c` 命令时, `sys.argv[0]` 被设定为 `'-c'`.
- 使用 `-m` 模块时, `sys.argv[0]` 被设定为模块的全名.
- `-c command` 或 `-m module` 之后的参数不会被 Python 解释器的处理机制所截获, 而是留在 `sys.argv` 中, 供命令或模块操作.



2.1.2 交互模式

从 `tty` 读取命令时, 我们称解释器工作于交互模式 (*interactive mode*). 这种模式下它通过主提示符 (*primary prompt*) 提示下一条命令, 主提示符通常为三个大于号 (`>>>`); 而通过从提示符由 (三个点标识 `...` 组成) 提示一条命令的续行. 在第一条命令之前, 解释器会打印欢迎信息, 版本号和授权提示:

```
$ python3.2
Python 3.2 (py3k, Sep 12 2011, 12:21:02)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

输入多行结构时就需要从属提示符了, 例如, 下面这个 `if` 语句:

```
1 >>> the_world_is_flat = 1
2 >>> if the_world_is_flat:
3 ...     print("Be careful not to fall off!")
4 ...
5 Be careful not to fall off!
```

2.2 解释器及其环境

2.2.1 错误处理

有错误发生时, 解释器会输出错误信息和栈跟踪. 交互模式下, 它返回到主提示符, 如果从文件输入执行, 它在打印栈跟踪后以非零状态退出. (在 `try` 语句中抛出并被 `except` 从句处理的异常不是这里所讲的错误). 一些常致命的错误会导致非零状态下退出, 这通常由内部问题或内存溢出造成, 所有的错误信息都写入标准错误流; 命令中执行的普通输出写入标准输出.

在主提示符或从属提示符后输入中断符 (通常是 Control-C 或者 DEL) 就会取消当前输入, 回到主提示符.⁴ 执行命令时输入一个中断符会抛出一个 `KeyboardInterrupt` 异常, 它可以被 `try` 语句截获.

2.2.2 可执行的 Python 脚本

类 BSD 的 UNIX 系统中, Python 脚本可以像 Shell 脚本那样直接执行, 只要在脚本文件开头加一行文本来声明模式:

```
1 #!/usr/bin/env python3.2
```

(要先确认 Python 解释器存在于用户的 **PATH** 环境变量中). `#!` 这两个字符必须是文件的头两个字符. 在某些平台上, 第一行必须以 UNIX 风格的行结束符 (`'\n'`) 结束, 不能用 Windows (`'\r\n'`) 的行结束符. 注意, `#` 用于 Python 一行注释的开始.

脚本可以用 `chmod` 命令指定可执行模式或权限:

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

在 Windows 系统下, 没有“可执行模式 (executable mode)”的概念. Python 安装器会自动地把 `.py` 后缀的文件与 `python.exe` 绑定, 因此双击一个 Python 文件, 就可以把它作为脚本来运行. 扩展名也可以是 `.pyw`, 这时工作台窗口会隐藏不被打开.

⁴一个 GNU Readline 包的问题可能会禁止这个功能.

2.2.3 源程序编码

默认情况下, Python 源码文件以 UTF-8 编码. 在这种编码下, 世界上大多数语言的字符都可以用于, 字符串常量, 标识符, 以及注释——尽管标准库遵循一个所有可移植代码都应遵守的约定: 仅使用 ASCII 字符作为标识符. 要正确地显示所有这些字符, 你的编辑器一定要有能力辨认出 UTF-8 编码, 还要使用一个支持所有文件中字符的字体.

也可以为源码文件指定不同的编码. 为此, 要在 `#!` 行后面指定一个特殊的注释行, 以定义源码文件的编码:

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

有了这样的声明, 源文件中的所有字符都会被以 `encoding` 的编码来解读, 而非是 UTF-8. 在 Python 库参考的 `codecs` 一节可以找到所有可用的编码.

例如, 如果你使用的编辑器不支持 UTF-8 编码, 但是支持另一种称为 Windows-1252 的编码, 你可以在源码中写上:

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

这样就可以在源码文件中使用 Windows-1252 字符集. 这个特殊的编码注释必须在代码文件的第一或第二行.

2.2.4 交互式启动文件

交互式地使用 Python 解释器时, 我们可能需要在每次启动时执行一些命令. 为了做到这点, 你可以设置一个名为 `PYTHONSTARTUP` 的变量, 指向包含启动命令的文件. 这类似于 Unix Shell 的 `.profile` 文件.

这个文件只在交互式会话中才被读取, 当 Python 从脚本中读取命令或显式地以 `/dev/tty` 作为命令源时 (尽管它的行为很像是处在交互会话期) 则不会如此. 它与解释器执行的命令处在同一个命名空间, 所以由它定义或引用的一切可以在解释器中不受限制的使用. 你也可以在这个文件中改变 `sys.ps1` and `sys.ps2` 的值.

如果你想要在当前目录中执行额外的启动文件, 可以在全局启动文件

中加入类似以下的代码: `if os.path.isfile('.pythonrc.py'):exec(open('.pythonrc.py').read())`. 如果你想要在某个脚本中使用启动文件, 必须要在脚本中写入这样的语句:

```
1 import os
2 filename = os.environ.get('PYTHONSTARTUP')
3 if filename and os.path.isfile(filename):
4     exec(open(filename).read())
```

2.2.5 定制模块

Python 为你提供两个钩子 (hook) 来定制交互环境: `sitecustomize` 和 `usercustomize`. 要知道它如何工作, 你需要先找到你的 user site-package 目录的位置. 打开 Python 并运行这段代码:

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

现在你可以在那个目录下创建一个名为 `usercustomize.py` 的文件, 并在里面放置任何你想放的东西. 它将影响到每一次 Python 的调用, 除非使用了 `-s` 选项来禁用了自动导入功能.

`sitecustomize` 以同样的方式工作, 但通常由该计算机的管理员在全局 site-packages 目录下创建, 并且在 `usercustomize` 之前被导入. 参看 `site` 模块的文档获取更多细节.

第三章 非正式介绍 Python⁵

在以下的例子中, 输入和输出通过提示符 (`>>>` 和 `...`) 来区分: 要运行示例, 你必须在提示符出现后键入提示符后面的所有内容; 不以提示符开头的行是解释器的输出.

⁵初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

注意, 在例子中有以一个次提示符独占一行时意味着你必须加入一个空行; 用来表示多行命令的结束.

这里有许多例子, 都是在交互式的提示符后中输入, 包括注释. Python 中的注释以一个井号, # 开头, 一直延伸到该物理行的最后. 注释既可以出现在一行的开头, 也可以跟着空白或代码后面, 但不能在字符串里面. 在字符串里面的井号只是一个井号字符. 因为注释使用来使代码清晰的, 而不是被 Python 解释, 所以在键入例子是可以省略它们.

例如:

```
1 # 这是第一个注释
2 SPAM = 1                # 这是第二个注释
3                          # ... 而现在是第三个!
4 STRING = "#_这不是注释."
```

3.1 把 Python 当计算器使用

让我们尝试一些简单的 Python 命令. 打开解释器, 等待主提示符, >>>, 出现. (这不会很久)

3.1.1 数值

解释器可作为简单的计算器: 输入表达式给它, 它将输出表达式的值. 表达式语法很直白: 操作符 +, -, *, / 就像大多数语言一样工作 (例如, Pascal 和 C); 圆括号用以分组. 例如:

```
1 >>> 2+2
2 4
3 >>> # 这是注释
4 ... 2+2
5 4
6 >>> 2+2 # 代码同一行的注释
7 4
8 >>> (50-5*6)/4
9 5.0
10 >>> 8/5 # 整数相除时并不会丢失小数部分
11 1.6
```

注意: 在你那儿, 可能结果并不完全相同; 因为不同机器上的浮点数结果可能不同, 待会我们会讲如何控制浮点数输出地显示. 具体参见浮点算术: 问题和限制中关于浮点数的细节和表示法的完整讨论.

要从整数相除中得到一个整数, 丢弃任何小数部分, 可以使用另一个操作符, `//`:

```
1 >>> # 整数相除返回地板数:
2 ... 7//3
3 2
4 >>> 7//-3
5 -3
```

等号 (`=`) 用于把一个值分配给一个变量. 其后不会输出任何结果, 而是下一个交互提示符:

```
1 >>> width = 20
2 >>> height = 5*9
3 >>> width * height
4 900
```

一个值可以同时被赋给多个变量:

```
1 >>> x = y = z = 0 # 给 x, y 和 z 赋值 0
2 >>> x
3 0
4 >>> y
5 0
6 >>> z
7 0
```

变量在使用之前必须要被“定义”(分配一个值), 否则会产生一个错误:

```
1 >>> # 尝试访问未定义的变量
2 ... n
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 NameError: name 'n' is not defined
```

Python 完全支持浮点数; 在混合计算时, Python 会把整型转换成为浮点数:

```
1 >>> 3 * 3.75 / 1.5
2 7.5
3 >>> 7.0 / 2
4 3.5
```

复数也有支持; 虚数部分写得时候要加上后缀, `j` 或 `i`. 实部非零的复数被写作 `(real+imagj)`, 也可以通过函数 `complex(real,imag)` 生成.

复数总是表达为两个浮点数, 实部和虚部. 要从复数 z 中抽取这些部分, 使用 `z.real` 和 `z.imag`

```
1 >>> a=1.5+0.5j
2 >>> a.real
3 1.5
4 >>> a.imag
5 0.5
```

浮点数和整数的转换函数 (`float()`, `int()`) 不能用于复数——没有正确的方法能把一个复数转换为一个实数. 使用 `abs(z)` 得到它的模 (以一个浮点数), 使用 `z.real` 得到他的实部:

```
1 >>> a=3.0+4.0j
2 >>> float(a)
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in ?
5   TypeError: can't convert complex to float; use abs(z)
6 >>> a.real
7 3.0
8 >>> a.imag
9 4.0
10 >>> abs(a) # sqrt(a.real**2 + a.imag**2)
11 5.0
```

在交互模式下, 最后一个表达式的值被分配给变量 `_`. 这意味着当你把 Python 用作桌面计算器时, 可以方便的进行连续计算, 例如:

```
1 >>> tax = 12.5 / 100
2 >>> price = 100.50
3 >>> price * tax
4 12.5625
```



```
5 >>> price + _  
6 113.0625  
7 >>> round(_, 2)  
8 113.06
```

我们应该将这个变量视作只读的. 不要试图给它赋值——否则你会创建一个同名的局部变量, 而隐藏原本内置变量的魔术效果.



3.1.2 字符串

除了数字, Python 也可以通过几种不同的方式来操作字符串. 字符串是用单引号或双引号包裹起来的:

```
1 >>> 'spam_eggs'  
2 'spam_eggs'  
3 >>> 'doesn\'t'  
4 "doesn't"  
5 >>> "doesn't"  
6 "doesn't"  
7 >>> '"Yes,"_he_said.'  
8 '"Yes,"_he_said.'  
9 >>> "\"Yes,\"_he_said."  
10 "\"Yes,\"_he_said.'  
11 >>> '"Isn\'t,"_she_said.'  
12 '"Isn\'t,"_she_said.'
```

解释器以字符串键入时相同的方式打印它们: 在引号里面, 使用引号或其它使用反斜杠的转义字符以说明精确的值. 当字符串包含单引号而没有双引号时, 就使用双引号包围它, 否则, 使用单引号. `print()` 函数为如此的字符串提供了一个更可读的输出.

字符串有几种方法来跨越多行. 继续行可以被使用, 在一行最后加上一个反斜杠以表明下一行是这行的逻辑延续:

```
1 hello = "这是一个相当长的字符串包含\n\  
2 几行文本,就像你在_C_里做的一样.\n\  
3 _ _ _ _ _ 注意开通的空白是\  
4 _ 有意义的."  
5  
6 print(hello)
```

注意, 换行依旧需要在字符串里嵌入 `\n`——在后面的反斜杠后面的换行被丢弃了. 该示例会打印如下内容:

```
1 这是一个相当长的字符串包含
2 几行文本, 就像你在 C 里做的一样.
3     注意开通的空白是 有意义的.
```

另一种方法, 字符串可以使用一对匹配的三引号对包围: `"""` 或 `'''`. 使用三引号时, 回车不需要被舍弃, 他们会包含在字符串里. 于是下面的例子使用了一个反斜杠来避免最初不想要的空行.

```
1 print("""\
2 用途: thingy [OPTIONS]
3     -h                显示用途信息
4     -H hostname       连接到的主机名
5 """)
```

产生如下输入:

```
1 用途: thingy [OPTIONS]
2     -h                显示用途信息
3     -H hostname       连接到的主机名
```

如果我们把字符串变为一个“未处理”字符串, `\n` 序列不会转义成回车, 但是行末的反斜杠, 以及源中的回车符, 都会当成数据包含在字符串里. 因此, 这个例子:

```
1 hello = r"这是一个相当长的字符串包含\n\
2 几行文本, \就像你在 \C\里做的一样."
3
4 print(hello)
```

将会打印:

```
1 这是一个相当长的字符串包含\n\
2 几行文本, 就像你在 C 里做的一样.
```

字符串可以使用 `+` 操作符来连接 (粘在一起), 使用 `*` 操作符重复:

```
1 >>> word = 'Help' + 'A'
2 >>> word
```

```

3 'HelpA'
4 >>> '<' + word*5 + '>'
5 '<HelpAHelpAHelpAHelpAHelpA>'

```

两个靠着一一起的字符串会自动的连接; 上面例子的第一行也可以写成 `word = 'HelpA'`; 这只能用于两个字符串常量, 而不能用于任意字符串表达式:

```

1 >>> 'str' 'ing' # <- 可以
2 'string'
3 >>> 'str'.strip() + 'ing' # <- 可以
4 'string'
5 >>> 'str'.strip() 'ing' # <- 不正确
6 File "<stdin>", line 1, in ?
7     'str'.strip() 'ing'
8         ^
9 SyntaxError: invalid syntax

```

字符串可以使用下标 (索引); 就像 C 一样, 字符串的第一个字符的下标 (索引) 为 0. 没有独立的字符串类型; 一个字符串就是一个大小为一的字符串. 就像 Icon 程序语言一样, 子字符串可以通过切片符号指定: 冒号分隔的两个索引.

```

1 >>> word[4]
2 'A'
3 >>> word[0:2]
4 'He'
5 >>> word[2:4]
6 'lp'

```

切片索引有一些有用的默认值; 省略的第一个索引默认为零, 省略的第二个索引默认为字符串的大小.

```

1 >>> word[:2] # 头两个字符
2 'He'
3 >>> word[2:] # 除了头两个字符
4 'lpA'

```

与 C 字符串不一样, Python 字符串不可以改变. 给字符串的索引位置赋值会产生一个错误:

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

然而, 使用内容组合创建新字符串是简单和有效的:

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

这有一个有用的切片操作的恒等式: $s[:i] + s[i:]$ 等于 s .

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

退化的切片索引被处理地很优雅: 太大的索引会被字符串大小所代替, 上界比下界小就返回空字符串.

```
1 >>> word[1:100]
2 'elpA'
3 >>> word[10:]
4 ''
5 >>> word[2:1]
6 ''
```

索引可以是负数, 那样就会从右边开始算起. 例如:

```
1 >>> word[-1]      # 最后一个字符
2 'A'
3 >>> word[-2]      # 倒数第二个字符
4 'p'
5 >>> word[-2:]     # 最后两个字符
6 'pA'
7 >>> word[:-2]     # 除最后两个字符的其他字符
```

```
8 | 'Hel'
```

但是要注意, -0 与 0 是完全一样的, 因此它不会从右边开始数!

```
1 | >>> word[-0]      # (因为 -0 等于 0)
2 | 'H'
```

越界的负切片索引会被截断, 当不要对单元素 (非切片) 索引使用越界索引.

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

记忆切片工作方式的一个方法是把索引看作是字符之间的点, 第一个字符的左边记作 0 . 包含 n 个字符的字符串最后一个字符的右边的索引就是 n , 例如:

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

第一行给出了索引 $0\dots5$ 在字符串里的位置; 第二行给出了相应的负索引. i 到 j 的切片由标号为 i 和 j 的边缘中间的字符所构成.

对于没有越界非负索引, 切片的长度就是两个索引之差. 例如, `word[1:3]` 的长度是 2.

内建函数 `len()` 返回字符串的长度:

```
1 | >>> s = 'supercalifragilisticexpialidocious'
2 | >>> len(s)
3 | 34
```

See more:

Sequence Types—`str`, `bytes`, `bytearray`, `list`, `tuple`, `range`

字符串是序列类型的例子, 支持该类型的一般操作.

String Methods

字符串支持大量用与基本变换和搜索的方法.

String Formatting

在这描述了使用 `str.format()` 格式字符串的信息.

Old String Formatting Operations

当字符串和 Unicode 字符串为 `%` 操作符的左操作数时, 老的格式操作就会被调用, 在这里描述了更多细节.



3.1.3 关于 Unicode

自 Python 3.0 开始, 所有字符串都支持 Unicode (参见 <http://www.unicode.org/>). Unicode 的益处在于它为自古至今所有文本中使用的每个字符提供了一个序号. 在以前, 只有 256 个序号表示文字字符. 一般地, 文本被一个映射序号到文本字符的编码页所限制. 尤其在软件的国际化 (internationalization, 通常被写作 `i18n` — '`i`' + 18 个字符 + '`n`') 时尤其混乱. Unicode 通过为所有文本定义一个编码页解决了这些难题.

如果你想在字符串里加入特殊字符, 可以使用 *Unicode-Escape* 编码. 下面的例子说明了如何做到这点:

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

转义序列 `\u0020` 表明在给出的位置, 使用序号值 `0x0020` (空格字符), 插入这个 Unicode 字符.

其它字符通过直接地使用它们各自的序号值作为 Unicode 序号而被解释. 如果你有使用标准 Latin-1 编码, 在很多西方国家里使用, 的字符串, 你会方便地发现 Unicode 的前 256 个字符与 Latin-1 的一样.

除这些标准编码以外, Python 还提供了整套其它方法, 通过一个已知编码的基础来创建 Unicode 字符串.

字符串对象提供了一个 **encode()** 方法, 用于使用一个特殊编码转换字符串到字节序列, 该方法带有一个参数, 编码的名字. 优先选择编码的小写名字.

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



3.1.4 列表

Python 有一些复合数据类型, 用来把其它值分组. 最全能的就是 *list*, 它可以写为在方括号中的通过逗号分隔的一系列值 (项). 列表的项并不需要是同一类型.

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

就像字符串索引, 列表的索引从 0 开始, 列表也可以切片, 连接等等:

```
1 >>> a[0]
2 'spam'
3 >>> a[3]
4 1234
5 >>> a[-2]
6 100
7 >>> a[1:-1]
8 ['eggs', 100]
9 >>> a[:2] + ['bacon', 2*2]
10 ['spam', 'eggs', 'bacon', 4]
11 >>> 3*a[:3] + ['Boo!']
12 ['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs',
    100, 'Boo!']
```

所有的切片操作返回一个包含请求元素的新列表. 这意味着, 下面的切片返回列表 *a* 的一个浅复制:

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

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

不像不可变的字符串, 改变列表中单个元素是可能的.

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

为切片赋值同样可能, 这甚至能改变字符串的大小, 或者完全的清除它:

```
1 >>> # 替代一些项:
2 ... a[0:2] = [1, 12]
3 >>> a
4 [1, 12, 123, 1234]
5 >>> # 移除一些:
6 ... a[0:2] = []
7 >>> a
8 [123, 1234]
9 >>> # 插入一些:
10 ... a[1:1] = ['bletch', 'xyzzy']
11 >>> a
12 [123, 'bletch', 'xyzzy', 1234]
13 >>> # 在开始处插入自身 (的一个拷贝)
14 >>> a[:0] = a
15 >>> a
16 [123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
17 >>> # 清除列表: 用空列表替代所有的项
18 >>> a[:] = []
19 >>> a
20 []
```

内建函数 `len()` 同样对列表有效:

```
1 >>> a = ['a', 'b', 'c', 'd']
2 >>> len(a)
3 4
```

嵌套列表 (创建包含其它列表的列表) 是可能的, 例如:

```
1 >>> q = [2, 3]
```



```
2 >>> p = [1, q, 4]
3 >>> len(p)
4 3
5 >>> p[1]
6 [2, 3]
7 >>> p[1][0]
8 2
```

你可以在列表末尾加入一些东西:

```
1 >>> p[1].append('extra')
2 >>> p
3 [1, [2, 3, 'extra'], 4]
4 >>> q
5 [2, 3, 'extra']
```

注意在最后的例子里, `p[1]` 和 `q` 确实指向同一个对象! 我们在以后会回到对象语义.

3.2 编程第一步

当然, 我们可以使用 Python 做比 $2 + 2$ 更复杂的任务. 例如, 我们可以如下的写出 *Fibonacci* 序列的最初子序列:

```
1 >>> # Fibonacci 序列:
2 ... # 两个元素的值定义下一个
3 ... a, b = 0, 1
4 >>> while b < 10:
5 ...     print(b)
6 ...     a, b = b, a+b
7 ...
8 1
9 1
10 2
11 3
12 5
13 8
```

这个例子介绍了几个新特性.

- 第一行包括一次多重赋值: 变量 `a` 和 `b` 同时地得到新值 0 和 1. 在最后一行又使用了一次, 演示了右边的表达式在任何赋值之前就已经被计算了. 右边表达式从左至右地计算.
- 当条件 (在这里: `b < 10`) 保持为真时, `while` 循环会一直执行. 在 Python 中, 就像 C 里一样, 任何非零整数都为真; 零为假. 条件也可以是字符串或列表, 实际上可以是任意序列; 长度不为零时就为真, 序列为假. 本例中使用的测试是一个简单的比较. 标准比较符与 C 中写得一样: `<` (小于), `>` (大于), `==` (等于), `<=` (小于或等于), `>=` (大于或等于) 和 `!=` (不等于).
- 循环体是缩进的: 缩进是 Python 分组语句的方法. Python 不 (到目前!) 提供智能输入行编辑功能, 因此, 你需要为每个缩进键入制表符或空格. 在练习中, 你会使用一个文本编辑器来为 Python 准备更复杂的输入; 大多文本编辑器带有自动缩进功能. 当一个复合语句交互地输入时, 必须跟上一个空行以表明语句结束 (因为语法分析器猜不到何时你键入了最后一行). 注意, 在同一基本块里的每一行必须以同一个数量缩进.
- `print()` 函数写出给它的表达式的值. 与单单输出你希望的表达式 (正如之前我们在计算器示例当中做的那样) 不同, `print()` 可以处理多个表达式、浮点数值、字符串. 字符串会被以没有引号的方式输出, 并且在项与项之间会插入空格, 故此你能漂亮地格式化这些东西, 就像这样:

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

关键词 `end` 可以用来避免输出后的回车, 或者以一个不同的字符串结束输出:

```
1 >>> a, b = 0, 1
2 >>> while b < 1000:
3 ...     print(b, end=',')
4 ...     a, b = b, a+b
```

```
5 ...  
6 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
```

第四章 深入流程控制⁶

除了刚介绍的 `while` 语句外, Python 也支持有其它语言中常见的控制语句, 当然有点小改动。

4.1 if 语句

也许最为人所知的语句类型就是 `if` 语句了。例如:

```
1 >>> x = int(input("Please enter an integer: "))  
2 Please enter an integer: 42  
3 >>> if x < 0:  
4 ...     x = 0  
5 ...     print('Negative changed to zero')  
6 ... elif x == 0:  
7 ...     print('Zero')  
8 ... elif x == 1:  
9 ...     print('Single')  
10 ... else:  
11 ...     print('More')  
12 ...  
13 More
```

这里可以有零个或多个 `elif` 分支, 而 `else` 是可选的。关键字 `elif` 是 `else if` 的缩写, 它可以有效避免过度缩进。 `if ... elif ... elif ...` 序列是其它语言中 `switch` 或 `case` 语句的替代。

4.2 for 语句

Python 中的 `for` 语句与你在 C 或是 Pascal 中使用的略有不同。不同于在 Pascal 中总是依据一个等差的数值序列迭代, 也不同于在 C 中允许用

⁶初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

户同时定义迭代步骤和终止条件, Python 中的 `for` 语句在任意序列 (列表或者字符串) 中迭代时, 总是按照元素在序列中的出现顺序依次迭代. FOR (这里不是循环;-) example:

```
1 >>> # 测试一些字符串:
2 ... a = ['cat', 'window', 'defenestrate']
3 >>> for x in a:
4 ...     print(x, len(x))
5 ...
6 cat 3
7 window 6
8 defenestrate 12
```

如果你需要在循环体内修改你正迭代的序列 (例如说, 复制序列中选定的项), 你最好是制作一个副本. 在序列上的迭代并不会自动隐式地创建一个副本. 切片标记让这种操作十分方便:

```
1 >>> for x in a[:]: # 制造整个列表的切片副本
2 ...     if len(x) > 6: a.insert(0, x)
3 ...
4 >>> a
5 ['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 range() 函式

如果你需要一个数值序列, 使用内建函式 `range()` 会很方便. 它产生等差级数序列:

```
1 >>> for i in range(5):
2 ...     print(i)
3 ...
4 0
5 1
6 2
7 3
8 4
```

给出的终止点不会在生成的序列里; `range(10)` 生成 10 个值, 组成一个长度为 10 的合法序列. 可以让 `range` 的起始初值定为另一个数, 也可以指

定一个不同的增量 (甚至可以为负; 有时这被称为' 步长'):

```
1 range(5, 10)
2     5 through 9
3
4 range(0, 10, 3)
5     0, 3, 6, 9
6
7 range(-10, -100, -30)
8     -10, -40, -70
```

要对一个序列的索引进行迭代的话, 组合使用 `range()` 和 `len()`:

```
1 >>> a = ['Mary', 'had', 'a', 'little', 'lamb']
2 >>> for i in range(len(a)):
3 ...     print(i, a[i])
4 ...
5 0 Mary
6 1 had
7 2 a
8 3 little
9 4 lamb
```

多数情况中, 用 `enumerate()` 函式更加方便, 参见遍历技巧.

直接打印 `range` 的时候会发生奇怪的事情:

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

在许多情况下, `range()` 的返回值和列表的行为类似, 但是事实上并非如此. `range()` 是这样一个对象: 当你对它的返回值进行迭代时, 它确实返回了一些连续的项, 但是它并没有创建这样一个列表, 并因此节省了空间.

我们将这种对象称为是可迭代的 (*iterable*), 也就是说, 它们能够作为某些期望获得一个完整的连续项序列的函式和构造器的目标. 我们已经知道 `for` 语句就是这样一个迭代器. 函式 `list()` 是另外一个这样的示例, 它能够自可迭代对象生成列表:

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

后面我们将看到更多返回可迭代对象和将可迭代对象作为参数的函式.

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

`break` 语句工作得如同 C 语言一样, 跳出最小的 `for` 或 `while` 循环.

循环语句可以有一个 `else` 子句; 该子句会在以下情况被执行: 循环因迭代到列表末尾而终止 (`for` 语句), 或者, 当循环条件为假 (`while` 语句), 同时它不会在循环因 `break` 语句终止的情况下被执行. 下面搜索素数的代码说明了这一特性:

```
1 >>> for n in range(2, 10):
2 ...     for x in range(2, n):
3 ...         if n % x == 0:
4 ...             print(n, 'equals', x, '*', n//x)
5 ...             break
6 ...         else:
7 ...             # loop fell through without finding a factor
8 ...             print(n, 'is a prime number')
9 ...
10 2 is a prime number
11 3 is a prime number
12 4 equals 2 * 2
13 5 is a prime number
14 6 equals 2 * 3
15 7 is a prime number
16 8 equals 2 * 4
17 9 equals 3 * 3
```

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

与循环搭配使用时, `else` 子句的行为和它与 `try` 语句的搭配时相对于它与 `if` 语句的搭配时有更多共性: `try` 语句的 `else` 子句在没有异常发生时被执行, 循环的 `else` 子句在没有 `break` 语句是被执行. 查阅[处理异常](#)一节获取更多关于 `try` 语句和异常的信息.

`continue` 语句同样是从 C 语言借用的, 它终止当前迭代而进行循环的下一迭代.

```
1 >>> for n in range(2, 10):
2 ...     for x in range(2, n):
3 ...         if n % x == 0:
4 ...             print(n, 'equals', x, '*', n//x)
5 ...             break
```

```
6 ...     else:
7 ...         # 循环因为没有找到一个因数而停止
8 ...         print(n, 'is a prime number')
9 ...
10 2 is a prime number
11 3 is a prime number
12 4 equals 2 * 2
13 5 is a prime number
14 6 equals 2 * 3
15 7 is a prime number
16 8 equals 2 * 4
17 9 equals 3 * 3
```



4.5 pass 语句

`pass` 语句什么都不做. 当语法上需要一个语句, 但程序不要动作时, 就可以使用它. 例如:

```
1 >>> while True:
2 ...     pass # 忙等待键盘中断 (Ctrl+C)
3 ...
```

一般也可以用于创建最小类:

```
1 >>> class MyEmptyClass:
2 ...     pass
3 ...
```

另一个使用 `pass` 的地方是, 作为函式或条件体的占位符, 当你在新代码工作时, 它让你能保持在更抽象的级别思考. `pass` 会被默默地被忽略:

```
1 >>> def initlog(*args):
2 ...     pass # 记得实现这里!
3 ...
```

4.6 定义函式

我们可以创建函式来输出任意指定范围内的菲波那契 (Fibonacci) 数列:

```
1 >>> def fib(n):      # 打印 Fibonacci 序列到 n
2 ...     """打印到 n 的 Fibonacci 序列."""
3 ...     a, b = 0, 1
4 ...     while a < n:
5 ...         print(a, end=' ')
6 ...         a, b = b, a+b
7 ...     print()
8 ...
9 >>> # 现在调用我们刚定义的函数:
10 ... fib(2000)
11 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

关键字 `def` 引入了一个函数定义。后面必须跟上函数名和在圆括号里的参数序列。函数体从一行开始，并且一定要缩进。

函数体的第一个语句可以是字符串；这个字符串就是函数的文档字符串，或称为 *docstring*。（可以在[文档字符串](#)一节找到更多信息）有很多能将文档字符串自动转换为在线或可打印文档的工具，或让用户在代码中交互地浏览它的工具；在代码里加上文档字符串是一个好的实践，因此，请养成这个习惯。

执行函数，会引入新的符号表 (symbol table) 用于该函数的局部变量。更精确地说，所有在函数中被赋值的变量和值都将存储在局部符号表中；鉴于变量引用会首先在局部符号表里寻找，然后才是闭包函数的局部符号表，再然后是全局变量，最后是内建名字表。因此，在函数中的尽管全局变量可以引用，但是不可直接赋值（除非用 `global` 语句进行声明）。

函数的实参在它被调用时被引入到这个函数的局部变量表；因此，参数是按值传递的（值总是对象的一个引用，而不是对象本身的值）。⁷ 当一个函数调用另一个时，对应这次调用，一个新的局部符号表就会被创建。

函数定义会在当前的符号表里引入该函数的名字。函数名对应的值被解释器认定为自定义函数类型函数名的值可以被赋予另一个名字，使其也能作为函数使用。这是常规的重命名机制：

```
1 >>> fib
2 <function fib at 10042ed0>
```

⁷实际上，通过对象引用调用会是个更好的描述，因为如果传入了一个可变参数，调用者将看到被调用者对它作出的任何改变（项被插入到列表）。


```
3 >>> f = fib
4 >>> f(100)
5 0 1 1 2 3 5 8 13 21 34 55 89
```

根据其它语言的经验, 你可能会指出 `fib` 不是一个函数, 而是一个程序, 因为它不返回值. 事实上, 即使没有 `return` 语句的函数也会返回一个值. 尽管这个值相当无聊. 这个值名为 `None` (它是个内建名字). 如果要唯一的一个值是 `None`, 那么解释器会正当的抑制这次返回. 如你实在想看看这个值, 可以使用 `print()`:

```
1 >>> fib(0)
2 >>> print(fib(0))
3 None
```

写个返回 Fibonacci 序列而不是打印输出的函数, 很简单:

```
1 >>> def fib2(n): # 放回直到 n 的 Fibonacci 序列
2 ...     """返回一个列表, 包含直到 n 的 Fibonacci 序列."""
3 ...     result = []
4 ...     a, b = 0, 1
5 ...     while a < n:
6 ...         result.append(a)    # 见下文
7 ...         a, b = b, a+b
8 ...     return result
9 ...
10 >>> f100 = fib2(100)    # 调用
11 >>> f100                # 输出结果
12 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

像往常一样, 这里介绍了一些 Python 特性:

- `return` 语句从函数中返回一个值. 没有表达式参数的 `return` 语句返回 `None`. 直到函数结束也没有 `return` 语句也返回 `None`.
- 语句 `result.append(a)` 调用了列表对象 `result` 的方法. 所谓方法是“属于”一个对象并且被命名为 `obj.methodname` 的函数, 此处 `obj` 是对象的名字 (这也可能是一个表达式), `methodname` 是这个对象的类型定义的一个方法的名字. 不同的类型定义不同的方法. 不同类型的方法可以有相同的名字而不会引起歧义. (你可以通过类定义你自

己的对象类型和方法, 参阅类) 示例中的方法 `append()` 是为列表对象定义的; 它会在列表的末尾添加新的元素. 在这个例子中, 他等价于 `result = result + [a]`, 但相对而言更加高效.

4.7 深入函式定义



函式定义时候可以带若干参数, 有三种可以组合使用的不同形式.

4.7.1 默认参数

最有用的形式是为一个或更多参数指定默认值. 这样创建的函式调用时可以用不用给足参数. 例如:

```
1 def ask_ok(prompt, retries=4, complaint='Yes_or_no,_please!')
2     :
3     while True:
4         ok = input(prompt)
5         if ok in ('y', 'ye', 'yes'):
6             return True
7         if ok in ('n', 'no', 'nop', 'nope'):
8             return False
9         retries = retries - 1
10        if retries < 0:
11            raise IOError('refusenik_user')
12        print(complaint)
```

这个函式有以下几种合法调用形式:

- 仅给出强制的参数: `ask_ok('Do_you_really_want_to_quit?')`
- 多出一个可选参数: `ask_ok('OK_to_overwrite_the_file?', 2)`
- 或给出所有参数: `ask_ok('OK_to_overwrite_the_file?', 2, 'Come_on,_only_yes_or_no!')`

这个例子也引入了一个关键字, `in` 用以测试序列中是否包含某一值.

默认参数的值等于函式定义域中的值, 因此:

```
1 i = 5
2
3 def f(arg=i):
```

```
4     print(arg)
5
6     i = 6
7     f()
```

将打印 5.

重要警告: 默认参数的值只会被求一次值. 这使得当默认参数的值是变对象时会会有所不同, 如列表, 字典, 或大多类的对象时. 例如, 下面的 `f` 在随后的调用中会累积参数值:

```
1 def f(a, L=[]):
2     L.append(a)
3     return L
4
5 print(f(1))
6 print(f(2))
7 print(f(3))
```

将会打印:

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

如果你不想让参数值被后来的调用共享, 你可以改写成这样:

```
1 def f(a, L=None):
2     if L is None:
3         L = []
4     L.append(a)
5     return L
```

4.7.2 关键字参数

函数也可以通过 `keyword= value` 形式的关键字参数来调用. 例如, 下面的函数:

```
1 def parrot(voltage, state='a stiff', action='vroom', type='
2     Norwegian Blue'):
3     print("-- This parrot wouldn't", action, end=' ')
```

```

3 print("if you put", voltage, "volts through it.")
4 print("-- Lovely plumage, the", type)
5 print("-- It's", state, "!")

```

通过以下任一方法调用:

```

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

```

但如下的调用是非法的:

```

1 parrot() # 缺少必要的参数
2 parrot(voltage=5.0, 'dead') # 在关键字后面跟着非关键字参数
3 parrot(110, voltage=220) # 同一参数给了多个值
4 parrot(actor='John Cleese') # 未知关键字

```

在函式调用时, 关键字参数必须跟在位置参数之后. 所有的关键字参数都必须与函式接受的形式参数匹配 (例如, `actor` 在函式 `'parrot'` 看来就是非法参数), 但他们的顺序是无关紧要的. 这条规则也适用于非可选参数 (例如, `parrot(voltage=1000)` 也是非法的). 任何形式参数都不能多次接受传值. 下面的例子产生错误的原因正是违反了这一限制:

```

1 >>> def function(a):
2 ...     pass
3 ...
4 >>> function(0, a=0)
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in ?
7 TypeError: function() got multiple values for keyword
   argument 'a'

```

当最后一个形式参数的形式为 `**name` 时, 则除去其他的形参的值, 它将以字典 (参阅映射类型——字典) 的形式包含所有剩余关键字参数. 这种调用可以与具有 `*name` 形式的形式参数 (在下一小节中介绍) 联合使用, 这种形式参数接受所有超出函式接受范围的位置参数. (`*name` 必须在 `**name` 之前使用) 例如, 如果我们像这样定义一个函式:

```

1 def cheeseshop(kind, *arguments, **keywords):
2     print("--Do you have any", kind, "?")
3     print("--I'm sorry, we're all out of", kind)
4     for arg in arguments:
5         print(arg)
6     print("-" * 40)
7     keys = sorted(keywords.keys())
8     for kw in keys:
9         print(kw, ":", keywords[kw])

```

它可以如下地调用:

```

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

```

当然它将打印:

```

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

```

注意, 关键字参数名的列表是通过之前对字典 `keys()` 进行排序操作而创建的; 如果不这样做, 参数打印的顺序是不确定的。

4.7.3 任意参数表

最后, 最不常用的选择, 是指定函数能够在调用时接受任意数量的参数. 这些参数会被包装进一个元组 (参看元组和序列). 在变长参数之前, 可以使用任意多个正常参数:

```

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

```

一般地, 这种 variadic 参数必须在形参列表的末尾, 因为它们将接收传递给函数式的所有剩余输入参数. 任何出现在 `*arg` 之后的形式参数只能是关键字参数, 这意味着它们只能使用关键字参数的方式接收传值, 而不能使用位置参数.

```

1 >>> def concat(*args, sep="/"):
2     ...     return sep.join(args)
3     ...
4 >>> concat("earth", "mars", "venus")
5 'earth/mars/venus'
6 >>> concat("earth", "mars", "venus", sep=".")
7 'earth.mars.venus'

```

4.7.4 释放参数列表

也存在相反的情形: 当参数存在于一个既存的列表或者元组之中, 但却需要解包以若干位置参数的形式被函数调用. 例如说, 内建的 `range()` 函数期望接收分别的开始和结束的位置参数. 如果它们并非分别可用 (而是同时存在于一个列表或者元组中), 下面是一个利用 `*`-操作符解从列表或者元组中释放参数以供函数调用的例子:

```

1 >>> list(range(3, 6))           # 使用分离的参数正常调用
2 [3, 4, 5]
3 >>> args = [3, 6]
4 >>> list(range(*args))         # 通过解包列表参数调用
5 [3, 4, 5]

```

同样的, 字典可以通过 `**`-操作符来释放参数:

```

1 >>> def parrot(voltage, state='a stiff', action='vroom'):
2     ...     print("-- This parrot wouldn't", action, end=' ')
3     ...     print("if you put", voltage, "volts through it.", end
4     ...           = ' ')
5     ...     print("E's", state, "!")

```

```
6 >>> d = {"voltage": "four_million", "state": "bleedin'_  
demised", "action": "VOOM"}  
7 >>> parrot(**d)  
8 -- This parrot wouldn't_VOOM_if_you_put_four_million_volts_  
through_it._E's bleedin'_demised!
```

4.7.5 Lambda 形式

根据大众的需要, 一些通常出现在诸如 Lisp 等函数式编程语言中的特性也已被加入到了 Python. 使用关键字 `lambda`, 就可以创建短小的匿名函数. 这就是能返回它两个参数和的函数: `lambda a, b: a+b`. Lambda 形式可以在任意需要函数对象的地方使用. 语法上限制它们为单一的表达式. 像内嵌函数一样, lambda 形式可以引用当前域里的变量:

```
1 >>> def make_incrementor(n):  
2 ...     return lambda x: x + n  
3 ...  
4 >>> f = make_incrementor(42)  
5 >>> f(0)  
6 42  
7 >>> f(1)  
8 43
```

4.7.6 文档字符串

这里介绍一些文档字符串有关内容和格式的约定.

第一行总应当是对该对象的目的进行简述. 为了简短, 它不用显式地陈述对象的名字或类型, 因为都是可以用其它手段获得 (除非这名字恰巧是描述函数操作的动词). 这行应当以一个大写字母开始, 并以句号结束.⁸

如果这个文档字符串不只一行, 那么第二行应当为空, 以能从视觉上分隔概述和其它部分. 接下来的行应当为一个或更多段来描述该对象的调用条件, 它的边界效应等等.

⁸译注: 出于良好的编程素养考虑, 尽可能的用 E 文注释吧.

Python 的语法分析器并不会去除多行字符串里的缩进, 所以必要的时候, 就不得不使用处理文档的工具来去除缩进. 使用下面这条约定. 在文档字符串第一行后的第一个非空行决定整个文档字符串缩进的数量. (我们不使用第一行的原因是它通常与字符串的外引号相连而使得它的缩进不明显.) 留白”相当于”是文档字符串的起始缩进将会被清除. 每行不应该有不足的缩进, 如果有前导空白, 将会全部清除. 由制表符扩展成的空白应该测试是否可用 (一般被兑换成 8 个空格).

这有一个多行文档的例子:

```
1 >>> def my_function():
2 ...     """Do nothing, but document it.
3 ...
4 ...     No, really, it doesn't do anything.
5 ...     """
6 ...     pass
7 ...
8 >>> print(my_function.__doc__)
9 Do nothing, but document it.
10
11     No, really, it doesn't do anything.
```

4.8 插曲: 代码风格

从现在开始, 你将写更长更复杂的 Python 代码, 是时候谈论代码风格了. 大多语言可以用不同风格写 (简洁地说: **格式化**) 代码; 总是有一些会比其它的更具可读性. 使其它人能够轻松读懂你的代码通常是个好主意, 而接受一个漂亮的代码风格会对那有很大的帮助.

对于 Python, :pep:‘8‘ 已经呈现了大多数项目遵循的风格; 它宣传了一种十分可读而悦目的代码风格. 每个 Python 开发者都应当在某个时刻阅读它; 这里为你萃取了最重要的几点:

- 使用 4-空格缩进, 且没有制表符.
4 空格是在小缩进 (允许更多嵌套) 和大缩进 (更易读) 之间的好的妥协. 制表符会带来混乱, 最好不要使用.

- 设定自动换行 (Wrap), 使它们不超过 79 个字符.
这会帮助小屏幕的用户, 而且使得可以在大屏幕上同时显示几个代码文件成为可能.
- 使用空行分隔函式和类, 以及函式中的大的代码块.
- 尽可能令注释独占一行.
- 使用文档字符串.
- 在操作符两边, 逗号后面使用空格, 但是括号内部与括号之间直接相连的部分不要空格: `a = f(1, 2) + g(3, 4)`.
- 保持类名和函式名的一致性; 约定是, 类名使用 `CamelCase` 格式, 方法名和函式名使用 `lower_case_with_underscores` 形式. 永远使用 `self` 作为方法的第一个参数名 (参阅:ref:‘类的初印象 <tut-firstclasses>’ 获得更多有关类和方法的信息).
- 若代码打算用在国际化的环境中, 那么不要使用奇特的编码. Python 默认的 UTF-8, 或者甚至是简单的 ASCII 在任何情况下工作得最好.
- 同样地, 如果代码的读者或维护者只有很小的概率使用不同的语言, 那么不要在标识符里使用非 ASCII 字符.



第五章 数据结构⁹

本章深入讲述一些你已经学过的东西, 当然也同样增加了一些新的内容.

5.1 深入列表

列表数据类型还有一些方法. 这里把列表对象的所有的的方法都列了出来:

- `list.append(x)`

在列表的尾部添加一个项; 等价于 `a[len(a):]=[x]`.

⁹初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

- **list.extend(L)**
用给入的列表将当前列表接长; 等价于 `a[len(a):]=L`.
- **list.insert(i, x)**
在给定的位置上插入项. 第一个参数就是准备在它之前插入的元素的索引, 因此 `a.insert(0,x)` 会在列表的头部插入, 而 `a.insert(len(a), x)` 则等价于 `a.append(x)`.
- **list.remove(x)**
移除列表中第一个值为 `x` 的项. 没有符合要求的项时, 会产生一个错误.
- **list.pop([i])**
删除列表给定位置的项, 并返回它. 如果没有指定索引, `a.pop` 移除并返回列表的最后一项. (函式原型的 `i` 在中方括号中意味着它是一个可选参数, 而不是你应当在那里键入一个方括号. 你将会在 Python 库参考中经常见到这种表示法.)
- **list.index(x)**
返回列表中第一个值为 `x` 的项索引值. 如果没有匹配的项, 则产生一个错误.
- **list.count(x)**
返回列表中 `x` 出现的次数.
- **list.sort()**
就地完成列表排序.
- **list.reverse()**
就地完成列表项的翻转.

下面这个示例演示了列表的大部分方法:

```
1 >>> a = [66.25, 333, 333, 1, 1234.5]
2 >>> print(a.count(333), a.count(66.25), a.count('x'))
3 2 1 0
4 >>> a.insert(2, -1)
5 >>> a.append(333)
6 >>> a
7 [66.25, 333, -1, 333, 1, 1234.5, 333]
```

```
8 >>> a.index(333)
9 1
10 >>> a.remove(333)
11 >>> a
12 [66.25, -1, 333, 1, 1234.5, 333]
13 >>> a.reverse()
14 >>> a
15 [333, 1234.5, 1, 333, -1, 66.25]
16 >>> a.sort()
17 >>> a
18 [-1, 1, 66.25, 333, 333, 1234.5]
```

5.1.1 把列表当成堆栈用

列表的方法使得其能十分简便的当成堆栈来使用, 堆栈的特性是最后添加的元素就是第一个取出的元素 (即“后入先出”). 要在栈顶添加一个项, 就使用 **append()**. 要从栈顶取回一个项, 就使用不带显式索引的 **pop()**. 例如:

```
1 >>> stack = [3, 4, 5]
2 >>> stack.append(6)
3 >>> stack.append(7)
4 >>> stack
5 [3, 4, 5, 6, 7]
6 >>> stack.pop()
7 7
8 >>> stack
9 [3, 4, 5, 6]
10 >>> stack.pop()
11 6
12 >>> stack.pop()
13 5
14 >>> stack
15 [3, 4]
```

5.1.2 把列表当队列使用

也可以把列表当成队列使用, 队列的特性是第一个添加的元素就是第一个取回的元素 (即”先入先出”); 然而, 这时列表是低效的. 从列表的尾部添加和弹出是很快的, 而在列表的开头插入或弹出是慢的 (因为所有元素都得移动一个位置).

要实现一个队列, 使用 `collection.deque`, 它被设计成在两端添加和弹出都很快. 例如:

```
1 >>> from collections import deque
2 >>> queue = deque(["Eric", "John", "Michael"])
3 >>> queue.append("Terry")           # Terry 进入
4 >>> queue.append("Graham")         # Graham 进入
5 >>> queue.popleft()                 # 第一个进入的现在离开
6 'Eric'
7 >>> queue.popleft()                 # 第二个进入的现在离开
8 'John'
9 >>> queue                           # 剩余的队列,
   它按照进入的顺序排列
10 deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 列表推导式

列表推导式提供了从序列中创建列表的简便途径. 通常程序会对序列的每一个元素做些操作, 并以其结果作为新列表的元素, 或者根据指定的条件来创建子序列.

而列表推导式的结构是, 在一个方括号里, 首先是一个表达式, 随后是一个 `for` 子句, 然后是零个或更多的 `for` 或 `if` 子句. 结果将是通过计算 `for` 和 `if` 子句来获得的一个列表. 如果要使表达式推导式出元组, 就必须用圆括号.

这里我们将一个数字列表每个元素翻三倍从而生成一个新列表:

```
1 >>> vec = [2, 4, 6]
2 >>> [3*x for x in vec]
3 [6, 12, 18]
```

现在加点儿小花样:

```
1 >>> [[x, x**2] for x in vec]
2 [[2, 4], [4, 16], [6, 36]]
```

这里我们对序列里每一项逐个调用某方法:

```
1 >>> freshfruit = ['_banana', '_loganberry_', 'passion_fruit_']
2 >>> [weapon.strip() for weapon in freshfruit]
3 ['banana', 'loganberry', 'passion_fruit']
```

我们可以用 if 子句来进行过滤:

```
1 >>> [3*x for x in vec if x > 3]
2 [12, 18]
3 >>> [3*x for x in vec if x < 2]
4 []
```

元组经常能不用圆括号而创建, 但这里不行:

```
1 >>> [x, x**2 for x in vec] # error - parens required for
   tuples
2 File "<stdin>", line 1, in ?
3   [x, x**2 for x in vec]
4       ^
5 SyntaxError: invalid syntax
6 >>> [(x, x**2) for x in vec]
7 [(2, 4), (4, 16), (6, 36)]
```

这里是一些循环的嵌套和其它技巧的演示:

```
1 >>> vec1 = [2, 4, 6]
2 >>> vec2 = [4, 3, -9]
3 >>> [x*y for x in vec1 for y in vec2]
4 [8, 6, -18, 16, 12, -36, 24, 18, -54]
5 >>> [x+y for x in vec1 for y in vec2]
6 [6, 5, -7, 8, 7, -5, 10, 9, -3]
7 >>> [vec1[i]*vec2[i] for i in range(len(vec1))]
8 [8, 12, -54]
```

列表推导式可使用复杂的表达式和嵌套的函数:

```
1 >>> [str(round(355/113, i)) for i in range(1, 6)]
2 ['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4 嵌套列表推导式

如果你受得了的话, 其实列表推导式是可以嵌套的. 它的确是个强大的工具, 但——就像所有强大的工具一样——需要被小心地使用.

考虑下面的例子, 有一个 3×3 的矩阵, 存储为一个包含三个列表的列表, 每一行一个列表:

```
1 >>> mat = [
2 ...     [1, 2, 3],
3 ...     [4, 5, 6],
4 ...     [7, 8, 9],
5 ...     ]
```

现在, 如果你想交换行和列, 可以使用列表推导式:

```
1 >>> print([row[i] for row in mat] for i in [0, 1, 2])
2 [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

使用嵌套列表推导式时特别需要注意: 从右至左地阅读嵌套列表推导式更容易理解.

该代码的冗长版本, 就明白地表述了流程:

```
1 for i in [0, 1, 2]:
2     for row in mat:
3         print(row[i], end=" ")
4     print()
```

现实中, 你应当选择内建函数来处理复杂流程. 这里, 函数 `zip()` 就非常好用.

```
1 >>> list(zip(*mat))
2 [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

参见解包参数列表了解本行中星号的详细内容.

5.2 del 语句

这有一种通过给定索引而不是值, 来删除列表中项的方法: 用 `del` 语句. 它与返回一个值的 `pop()` 方法不同. `del` 语句也可以移除列表中的切片, 或者清除整个列表 (之前我们通过给切片赋值为空列表来完成这点). 例如:

```
1 >>> a = [-1, 1, 66.25, 333, 333, 1234.5]
2 >>> del a[0]
3 >>> a
4 [1, 66.25, 333, 333, 1234.5]
5 >>> del a[2:4]
6 >>> a
7 [1, 66.25, 1234.5]
8 >>> del a[:]
9 >>> a
10 []
```

`del` 也可以用于删除变量实体:

```
1 >>> del a
```

在这之后引用 `a` 的话会产生一个错误 (至少到给它赋另一个值之前). 我们将在后面找到 `del` 的其它用法.

5.3 元组和序列

我们看到列表和字符串有很多通用的属性, 例如索引和切片操作. 它们是序列数据类型的两个例子 (参考 [Sequence Types—str, bytes, bytearray, list, tuple, range](#)). Python 作为一门进化中的语言, 可能还有其它序列类型会被加入. 这里就有另一种标准序列数据类型: **元组**.

元组由若干逗号分隔的值组成, 例如:

```
1 >>> t = 12345, 54321, 'hello!'
2 >>> t[0]
3 12345
4 >>> t
5 (12345, 54321, 'hello!')
```

```
6 >>> # Tuples may be nested:
7 ... u = t, (1, 2, 3, 4, 5)
8 >>> u
9 ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

如你所见, 元组输出时用圆括号包围, 以便正确表达元组的嵌套; 在输入时圆括号可加可不加, 不过圆括号经常是必要的 (特别是当元组是更大表达式的一部分时).

元组有许多用途. 例如: (x, y) 坐标对, 数据库里的员工记录等. 元组同字符串都是不可变的: 无法对元组指定项进行赋值 (尽管可通过切片和连接来模拟这个操作). 元组中可以包含可变的对象, 如列表.

构造包含 0 或 1 个项的元组是个特殊问题: 语法上为了适应这一情况, 有些额外的规则. 空元组由一对空的圆括号构造; 一个项的元组由一个值后面跟着一个逗号构造 (把一个值放入一对圆括号里并不足以构造一个元组). 丑陋, 但有效. 例如:

```
1 >>> empty = ()
2 >>> singleton = 'hello',      # <-- 注意后面的逗号
3 >>> len(empty)
4 0
5 >>> len(singleton)
6 1
7 >>> singleton
8 ('hello',)
```

语句 `t = 12345, 54321, 'hello!'` 是元组打包的一个例子: 值 12345, 54321 和 'hello!' 被打包进一个元组. 反过来, 这个操作也是可行的:

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

这种对右侧任一序列的处理很合适称为**序列解包**. 序列解包时要求等号左边的值个数与右边序列元素个数相等. 注意, 多重赋值其实是联合使用了元组打包和序列解包. (虽然元组和列表都算序列, 但是必须有所不同)

5.4 集合 (Set)

Python 还包含了集合 (*set*) 数据类型. 集合是种无序不重复的元素集. 基本用途包括成员关系测试和重复条目消除. 集合对象也支持并 (union), 交 (intersection), 差 (difference), 和对称差 (symmetric difference) 等数学操作.

花括号或函数 `set()` 可用于创建集合. 注意: 创建一个空集合只能使用 `set()`, 而不能使用 `{}`; 后者是创建一个空字典, 字典我们会在下一节里讨论.

以下是简明示范:

```
1 >>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
2 >>> print(basket)                # 重复的被移除了
3 {'orange', 'banana', 'pear', 'apple'}
4 >>> 'orange' in basket           # 快速成员关系测试
5 True
6 >>> 'crabgrass' in basket
7 False

1 >>> # 在两个单词的不重复的字母里演示集合操作
2 ...
3 >>> a = set('abracadabra')
4 >>> b = set('alacazam')
5 >>> a                               # a 中的不重复字母
6 {'a', 'r', 'b', 'c', 'd'}
7 >>> a - b                           # a 中有而 b
   中没有的字母
8 {'r', 'd', 'b'}
9 >>> a | b                           # 在 a 中或在 b
   中的字母
10 {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
11 >>> a & b                           # a 和 b 中都有的字母
12 {'a', 'c'}
13 >>> a ^ b                           # a 或 b
   中只有一个有的字母
14 {'r', 'd', 'b', 'm', 'z', 'l'}
```

和列表一样, 集合也支持推导式:

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

5.5 字典



Python 中另一很有用的内建数据类型为字典 (参考 [Mapping Types—dict](#))。在其它语言中字典一般被叫做“关联存储”或“关联数组”。与使用某个范围作为索引的序列不一样, 字典通过键来索引, 而键可以是任意不可变类型; 通常用字符串和数字作为键。如果元组只包含字符串和数字, 元组也可以作为键; 但是, 当元组直接或间接地包含可变对象时, 就不能用作一个键。不能使用列表作为键, 因为列表可以通过索引, 切片, 或如 `append()` 和 `extend()` 方法原地赋值而被改变。

最好把字典看成是一个没有顺序的键: 值对集合, 键必须是唯一的 (在一个字典里)。一对花括号创建一个空字典: `{}`。在括号中间放置的以逗号分隔的键: 值对列表就是字典的初始键: 值对。这也是字典输出时的格式。

字典最主要的操作是通过某键存储一个值, 以及从给定的键里提取它的值。使用 `del` 可以删除一个键: 值对。如果你使用一个已被使用的键进行存储操作, 该键的旧值就没有了。使用一个不存在的键提取值会产生一个错误。

在一个字典上执行 `list(d.keys())` 返回该字典中所使用键的列表, 该列表的顺序不确定 (如果需要有序, 只要使用 `sorted(d.keys())`)。¹⁰ 要检查某一个键是否在字典里, 使用 `in` 关键字。

这是一个使用字典的小例子:

```
1 >>> tel = {'jack': 4098, 'sape': 4139}
2 >>> tel['guido'] = 4127
3 >>> tel
4 {'sape': 4139, 'guido': 4127, 'jack': 4098}
5 >>> tel['jack']
```

¹⁰调用 `d.keys()` 将返回一个 `dictionary view` 对象。它支持类似成员关系测试以及迭代操作, 但是它的内容不是独立于原始字典的——它只是一个视图。

```

6 | 4098
7 | >>> del tel['sape']
8 | >>> tel['irv'] = 4127
9 | >>> tel
10 | {'guido': 4127, 'irv': 4127, 'jack': 4098}
11 | >>> list(tel.keys())
12 | ['irv', 'guido', 'jack']
13 | >>> sorted(tel.keys())
14 | ['guido', 'irv', 'jack']
15 | >>> 'guido' in tel
16 | True
17 | >>> 'jack' not in tel
18 | False

```

构造器 `dict()` 从键-值对序列里直接生成字典, 如果有固定的模式, 可在列表推导式指定特定的键值对:

```

1 | >>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
2 | {'sape': 4139, 'jack': 4098, 'guido': 4127}
3 | >>> dict([(x, x**2) for x in (2, 4, 6)]) # 使用列表推导式
4 | {2: 4, 4: 16, 6: 36}

```

在本教程后面的章节, 会学习到生成器表达式, 这更适于为 `dict()` 构造器生成键值对序列. 若键为字符串, 有时用关键字参数指定键值对更为简单:

```

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

```

5.6 遍历技巧

当对字典遍历时, 可用 `items()` 方法同时取回键和对应的值.

```

1 | >>> knights = {'gallahad': 'the_pure', 'robin': 'the_brave'}
2 | >>> for k, v in knights.items():
3 | ...     print(k, v)
4 | ...
5 | gallahad the pure
6 | robin the brave

```

对序列遍历时, 可以使用 `enumerate()` 函式来同时取回位置索引和相应的值.

```
1 >>> for i, v in enumerate(['tic', 'tac', 'toe']):
2     ...     print(i, v)
3     ...
4 0 tic
5 1 tac
6 2 toe
```

同时对两个或更多的序列进行遍历时, 可用 `zip()` 进行组合

```
1 >>> questions = ['name', 'quest', 'favorite_color']
2 >>> answers = ['lancelot', 'the_holy_grail', 'blue']
3 >>> for q, a in zip(questions, answers):
4     ...     print('What_is_your_{0}?_It_is_{1}'.format(q, a))
5     ...
6 What is your name? It is lancelot.
7 What is your quest? It is the holy grail.
8 What is your favorite color? It is blue.
```

反向遍历序列时, 先指定这个序列, 然后调用 `reversed()` 函式

```
1 >>> for i in reversed(range(1, 10, 2)):
2     ...     print(i)
3     ...
4 9
5 7
6 5
7 3
8 1
```

想有序地遍历一个序列, 用 `sorted()` 函式返回排序后的序列, 原序列将不被触及

```
1 >>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
2 >>> for f in sorted(set(basket)):
3     ...     print(f)
4     ...
5 apple
6 banana
7 orange
```

8 | pear

5.7 深入条件控制

在 `while` 和 `if` 语句中使用的条件可以包含任何操作符, 而不仅仅是比较.



- 比较操作符 `in` 和 `not in` 检查一个值是否在序列中.
- 操作符 `is` 和 `is not` 比较两个对象是否为同一对象; 这只对诸如列表的可变对象有用.

所有比较操作符具有相同的优先级, 低于所有的数值操作.

比较操作符可以连起来使用. 例如, `a < b == c` 测试 `a` 小于 `b` 且 `b` 与 `c` 相等.

比较操作 (或其它任何布尔表达式) 都能用逻辑操作符 `and` 和 `or` 连接, 结果值可以用 `not` 取反.

- 逻辑操作符的优先级又低于比较操作符;
- 这其中, `not` 优先级最高, 而 `or` 的优先级最低, 因此 `A and not B or C` 等价于 `(A and (not B)) or C`. 同样, 可以使用圆括号来表达想要的结果.
- 逻辑操作符 `and` 和 `or` 被称为短路操作符: 它从左至右计算参数, 并且当结果确定时计算就立即停止.
 - 例如, 如果 `A` 和 `C` 为真, 而 `B` 为假时, `A and B and C` 不会计算表达式 `C`.
 - 当把短路操作符的返回值作为一个常规值而不是布尔值时, 它的值就是最后计算的参数值.

可以把比较式或其它逻辑表达式的值赋给一个变量. 例如,

```
1 >>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
2 >>> non_null = string1 or string2 or string3
3 >>> non_null
4 'Trondheim'
```

注意, 在 Python 中, 不像 C, 赋值不可以发生在表达式内部. C 程序员可能对此有抱怨, 但是这样就避免了 C 程序中常见的一类错误, 比如说:

- 在使用 `==` 的表达式里键入了 `=`.

5.8 序列和其它类型的比较



序列对象可以与同一类型的其它对象比较. 使用字典编纂顺序比较

- 首先比较头两项, 如果它们不同, 它们的比较就决定整个比较的结果;
- 如果它们相同, 就比较下两项, 就这样直到其中有序列被比较完了.
- 如果要被比较的两项本身就是相同类型的序列, 那么就递归进行比较.
- 如果两个序列所有的项都相等, 那么, 它们就相等.
- 如果一个序列是另一个序列的初始子序列 (initial sub-sequence), 那么短的就是较小的.
- 字符串的字典编纂顺序由单个字符的 Unicode 字码来决定.

以下是比较相同类型序列的例子:

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

注意, 使用 `<` 或 `>` 比较两个不同类型的对象有时候是合法的, 条件是它们要有合适的比较方法. 例如, 不同的数字类型可以按照它们的数字大小来比较, 因此 `0` 等于 `0.0`, 等等. 否则, 解释器不会提供一个任意的顺序, 而会抛出一个 `TypeError` 异常.

第六章 模块¹¹

如果你从 Python 解释器退出后再重新进入, 那么你之前定义的所有(函式和变量) 都将丢失. 因此, 如果你想写一个更长的程序, 你最好离线地使用文本编辑器保存成文件, 替代解释器的输入来运行. 这称作创建一个脚本. 当你的程序变得更长, 你可能想把它分割成几个文件以能够更简单维护. 你也许还想在几个不同的程序里使用写过的程序, 而不用把一坨代码拷来拷去.

为此 Python 提供了方法, 能使用户把定义存放在文件里, 同时又能在脚本或交互式环境下方便的使用它们. 这样的文件称为模块; 一个模块中的定义可以导入 (`import`) 到另一个模块或主模块 (主模块是执行脚本的最上层或计算模式下的一组可访问变量的集合).

模块就是包含 Python 定义和语句的文件. 文件的名称就是这个模块名再加上 `.py`. 在一个模块中, 模块的名字 (一个字符串) 可以通过全局变量 `__name__` 得到. 例如, 使用你最喜欢的文档编辑器在当前目录下创建一个名为 `fibo.py` 的文件, 并输入以下内容:

```
1 # Fibonacci 数列模块
2
3 def fib(n):      # 打印小于 n 的 Fibonacci 数列
4     a, b = 0, 1
5     while b < n:
6         print(b, end=' ')
7         a, b = b, a+b
8     print()
9
10 def fib2(n): # 返回小于 n 的 Fibonacci 数列
11     result = []
12     a, b = 0, 1
13     while b < n:
14         result.append(b)
15         a, b = b, a+b
16     return result
```

¹¹初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

现在打开 Python 解释器并通过以下命令导入 (import) 这个模块:

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

这样并不会把 fibo 中定义的函数名导入 (import) 到当前的符号表里; 它只导入了模块名 fibo. 你可以使用模块名来访问函数:

```
1 >>> fibo.fib(1000)
2 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
3 >>> fibo.fib2(100)
4 [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
5 >>> fibo.__name__
6 'fibo'
```

如果你要经常使用一个函数式的话, 可以把它赋给一个局部变量:

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

6.1 深入模块

模块不仅包含函数定义, 还可以包含可执行的语句. 这些语句一般用以初始化模块. 他们仅在模块 第一次被导入时才被执行.¹²

每个模块有其私有的符号表, 由模块内部定义的函数当成全局符号表来使用. 因此, 模块的作者可以在模块中放胆使用全局变量而无需担心与用户的全局变量发生冲突. 另一方面, 当你确实明白你在做什么的时候, 你可以通过 *modname.itemname* 形式来访问模块的全局变量.

模块中可以导入其它的模块. 习惯上把 `import` 语句放在一个模块 (或者脚本,) 的最开始, 当然这只是惯例不是强制的. 被导入模块的名称被放入当前模块的全局符号表里.

`import` 语句这有一种不同用法, 它可以直接把一个模块内 (函数, 变量) 名称导入当前模块符号表里. 例如:

¹² 实际上, 函数定义也是“被执行”的“语句”; 模块级函数的执行让函数名进入这个模块的全局变量表.


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

这样不会导入相应的模块名 (在这个例子里, `fibo` 并没有被定义).

还有一种方法可一次性导入模块中所有的名字定义:

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

这样可以导入除以下划线开头 (`_`) 的所有名字. 多数情况下, Python 程序员不使用这个窍门, 因为它导入了一些未知的名字到解释器里, 因此可能会意外重载一些你已经定义的东西.

注意: 在一般的实践中, 导入 `*` 是不好的, 因为它常常产生难以阅读的代码. 然而, 在一个交互式会话里使用它可以节省键入.

Note:

因为效率的原因, 每个模块在每个解释器会话中只被导入一次. 一旦你修订了你的模块, 就需要重启解释器 — 或者, 若你只是想交互式地测试一个模块, 使用 `imp.reload()`, 例如 `import imp; imp.reload(modulename)`.

6.1.1 把模块当脚本执行

当你以如下方式运行一个 Python 模块时

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

模块中的代码就会被执行, 就像被导入时一样, 但 `__name__` 被设为 `"__name__"`. 这就意味着通过在模块最后加入以下代码:

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

就能够把这个文件既当成脚本使用, 也可以当成可导入的模块使用, 因为解析命令行的代码只当模块被当成“主 (main)”时才被直接运行:

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

如果模块被导入, 这段代码并不执行:

```
1 >>> import fibo
2 >>>
```

模块常通过这种形式提供一些方便的用户接口, 或用于测试 (把模块当脚本执行一个测试套件).

6.1.2 模块搜索路径

当导入名为 `spam` 的模块时, 解释器会先在当前目录下查找名为 `spam.py` 的文件, 若未找到, 则解释器会在由变量 `sys.path` 给定的目录列表中寻找名为 `spam.py` 的文件. `sys.path` 从这些位置初始化:

- 包含输入脚本的目录 (或当前目录).
- `PYTHONPATH` (一个目录名列表, 其语法与 shell 变量 `PATH` 相同).
- 安装时的默认目录.

初始化后, Python 程序可以修改 `sys.path`. 被运行的脚本所在的目录, 排在标准库路径之前被首先搜索. 这意味着若是脚本所在目录中有与标准库相同名称的模块, 则 Python 会加载这个模块而不是标准库中的模块. 除非有意如此, 不然可能会导致错误. 参见[标准模块](#)以获取更多信息.

6.1.3 “已编译”的 Python 文件

为了减少使用了大量标准模块的小程序的启动时间, 如果 `spam.py` 所在目录下中有名为 `spam.pyc` 的文件, 解释器就会优先导入 `spam` 模块的这一“已编译字节”版本文件. 用来创建 `spam.pyc` 的 `spam.py` 的版本修改时间被记录在 `spam.pyc` 中, 如果不匹配的话, `.pyc` 文件就会被忽略.

一般,你无需特意做什么来创建 `spam.pyc` 文件. 每次 `spam.py` 被成功编译后,都会尝试把结果写入到 `spam.pyc`. 这时有任何问题,并不会抛出错误;如果因某些原因导致这文件没有被完全的被写入,那么产生的 `spam.pyc` 文件会被辨别出是无效的,从而在导入时被忽略. `spam.pyc` 文件的内容是平台无关的,因此,一个 Python 模块目录可以在不同的体系架构中共享.



给专家的小技巧:

- 当使用 `-O` 参数来调用 Python 解释器时,python 会对代码进行优化,并存入在 `.pyo` 文件里. 当前优化器仅仅只是移除了 `assert` 语句. 当使用 `-O` 时,所有 `bytecode` 都被优化了;所有 `.pyc` 文件被忽略,而 `.py` 文件被编译为优化的字节码.
- 传递两个 `-O` 参数到 Python 解释器 (`-OO`) 会使编译器对字节码进一步优化,而该步骤在极少的情况下会产生发生故障的程序. 一般地,只是将 `__doc__` 字符串被从字节码中移除,以产生更为紧凑的 `.pyo` 文件. 因为有些程序可能依赖于这些,因此,建议只有当你真正明确这意味着什么时,才使用这个选项.
- 程序从 `.pyc` 或 `.pyo` 文件里读取时,并不会比它从 `.py` 文件中读取会有更快的执行速度;唯一提高的是载入速度.
- 在在命令行中直接调用脚本运行时,编译后的字节码不会被写入 `.pyc` 或 `.pyo` 文件. 因此,通过移动该脚本的大量代码到一个模块,并由一个小的引导脚本来导入这个模块,可能减少这个脚本的启动时间. 也可以直接在命令行里直接命名一个 `.pyc` 或 `.pyo` 文件.
- 对于同一个模块,可以只包含 `spam.pyc` (或者 `spam.pyo` 当使用 `-O` 时) 文件而无需 `spam.py` 文件. 使用这种形式可用以发布 Python 代码库,并使得反编译工程有一定的难度.
- 模块 `compileall` 可以为一个目录下的所有模块创建 `.pyc` 文件 (或 `.pyo` 文件,当使用 `-O` 时).

6.2 标准模块

Python 本身带有一个标准库, 有专门文档: Python 库参考 (以后简称“库参考”) 进行介绍.

有些模块内建到了解释器中; 有些操作尽管并不是语言核心的一部分, 但是通过模块内建提供后, 执行效率不错, 包含操作系统的一些基本访问, 例如系统调用.

这种模块能根据不同的操作系统进行专门配置, 例如, `winreg` 模块只在 Windows 系统中提供. 有一个特别的模块需要特别注意: `sys`, 它内建于每个 Python 解释器. 其中变量 `sys.ps1` 和 `sys.ps2` 定义了用于主和次提示符的字符串:

```
1 >>> import sys
2 >>> sys.ps1
3 '>>>_ '
4 >>> sys.ps2
5 '..._ '
6 >>> sys.ps1 = 'C>_ '
7 C> print('Yuck!')
8 Yuck!
9 C>
```

只有解释器在交互模式下运行时, 这两个变量才有定义.

变量 `sys.path` 是一个字符串列表, 它为解释器指定了模块的搜索路径. 它通过环境变量 `PATHONPATH` 初始化为一个默认路径, 当没有设置 `PYTHONPATH` 时, 就使用内建默认值来初始化. 你可以通过标准列表操作来修订之:

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

6.3 `dir()` 函式

内建函式 `dir()` 用于找出一个模块里定义了那些名字. 它返回一个有序字符串列表:

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

不给参数时, `dir()` 就罗列出当前已定义的所有名字.

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

注意, 它列举出了所有类型的名字: 变量, 模块, 函式, 等等.

`dir()` 并不列出内建函式和变量的名字. 如果你真心想看一下, 可以直接查询标准模块 `builtins`

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

6.4 包

包是一种 Python 模块命名空间的组织方法, 通过使用“带点号的模块名”。例如, 模块名 A.B 指定了一个名为 A 的包里的一个名为 B 的子模块。就像模块的使用使不同模块的作者避免担心其它全局变量的名字, 而带点号的模块使得多模块包, 例如 NumPy 或 Python 图像库, 的作者避免担心其它模块名。

假设你想设计一个模块集 (一个“包”), 用于统一声音文件和声音数据的处理。有许多不同的声音格式 (通常通过它们的后缀来辨认, 例如: .wave,

.aiff, .au), 因此你可能需要创建和维护一个不断增长的模块集, 用以各种各样的文件格式间的转换. 还有许多你想对声音数据执行的不同操作 (例如混频, 增加回音, 应用一个均衡器功能, 创建人造的立体声效果), 因此, 你将额外的写一个永无止尽的模块流来执行这些操作. 这是你的包的一个可能的结构:

sound/	顶级包
__init__.py	初始化这个声音包
formats/	文件格式转换子包
__init__.py	
wavread.py	
wavwrite.py	
aiffread.py	
aiffwrite.py	
auread.py	
auwrite.py	
...	
effects/	音效子包
__init__.py	
echo.py	
surround.py	
reverse.py	
...	
filters/	过滤器子包
__init__.py	
equalizer.py	
vocoder.py	
karaoke.py	
...	



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

录.

需要 `__init__.py` 文件来使得 Python 知道这个目录包含了包; 这用来预防名字为一个通用名字, 如 `string`, 的目录以外地隐藏了在模块搜索路径靠后的正当的模块. 在最简单的例子里, `__init__.py` 可以就是个空文件, 但它也可以为这个包执行初始化代码, 或者设置 `__all__` 变量, 在后面描述.

包的用户可以包里的单独的模块, 例如:

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

这载入里 `sound.effects.echo` 子模块. 一定要使用全名来引用它.

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

导入子模块的一个替代方法是:

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

这样也载入 `echo` 子模块, 并且可以不加包前缀地使用, 因此可以如下地使用:

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

另一个变种是直接导入想要的函式或变量:

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

再一次, 载入了 `echo` 子模块, 但是使它的函式 `echofilter()` 可以直接使用.

注意, 当使用 `from package import item` 时, 这个项即可以是这个包的一个子模块 (或子包), 也可以是其它的定义在这个包里的名字, 如函式, 类或变量. `import` 语句首先测试这个项是否在包里定义; 如果没有, 就假设它是一个模块并试图载入它. 如果寻找它失败, 就会抛出一个 `ImportError`.

相反地, 当使用 `import item.subitem.subsubitem` 时, 除最后的每一项都必须是包; 最后一项可以是模块或包, 但不能是在之前项中定义的类, 函式或变量.

6.4.1 从包中导入 *

当开发者写下 `from sound.effects import *` 会发生什么呢?

理想地, 我们期望程序会以某种方法进入文件系统, 寻找在指定的包文件中, 找到所有子模块, 并把它们全部导入. 这可能花费很长的时间, 而且对子模块进行显式导入时, 还可能引发非期待的副作用!

对于包作者, 唯一解决方案是提供包的显式索引. `import` 语句有以一定:

- 如果一个包的 `__init__.py` 代码定义了一个名为 `__all__` 的列表,
- 当遇到 `from sound.effects import *` 时, 它被用来作为导入的模块名字的列表.

是否在发布包的新版本时保持这个列表的更新取决于包的作者. 包作者也可能决定不支持它, 如果他们没有发现从他们的包里导入 `*` 的用途. 例如, 文件 `sound/effects/__init__.py` 可能包含如下代码:

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

这意味这 `from sound.effects import *` 将导入 `sound` 中这几个名字的子模块.

如果 `__all__` 没有被定义, `from sound.effects import *` 语句不把包 `sound.effects` 中所有的子模块都导入到当前命名空间里; 它只能确保包 `sound.effects` 被导入了 (可能同时运行在 `__init__.py` 里的一些初始化代码), 并随后导入包中定义的任何名字. 这包含任何在 `__init__.py` 定义的任何名字 (和显式载入的子模块). 它还包含通过前面的 `import` 语句显式载入的包的子模块. 考虑这段代码:

```
1 import sound.effects.echo
2 import sound.effects.surround
3 from sound.effects import *
```

在这个例子中, 模块 `echo` 和 `surround` 被导入到当前命名空间, 因为当执行 `from ... import` 语句时它们就被定义在包 `sound.effects` 里. (当定义 `__all__` 定义时, 这也会工作.)

尽管一些模块被设计成, 当你使用 `import *` 的时候, 只输出模块名 (that follow certain patterns, 不知道怎么翻), 这在最终输出的代码中仍然是不好的.

记住, 使用 `from Package import specific_submodule` 是没有问题的! 事实上, 这是推荐的用法, 只在以下情况例外: 正在导入的模块需要使用的子模块与其他包中的子模块具有相同的名字.



6.4.2 内部包参考

当包被构造到子包时 (如例子中的 `sound` 包), 你可以独立地导入来获取兄弟包的子模块的引用. 例如, 如果模块 `sound.filters.vocoder` 需要使用 `sound.effects` 包下的 `echo` 模块, 就可以使用 `from sound.effects import echo`.

你还可以使用相对导入, 通过 `import` 语句的 `from module import name` 格式. 这些导入使用句点来表明涉及这次相对导入的当前包和父包. 从例子中的:mod:“surround”, 您可以使用:

```
1 from . import echo
2 from .. import formats
3 from ..filters import equalizer
```

注意, 相对导入基于当前模块的名字. 因为主模块的名字总是 `"__main__"`, 有意用作一个 Python 程序的主模块的模块必须总使用相对导入.

6.4.3 多目录的包

包支持额外一个特殊的属性, `__path__`. 它在文件中的代码执行之前, 被初始化为一个列表, 它包含保存在这个包的 `__init__.py` 文件中目录名. 这个变量可以被更改; 这样做会影响以后对包中模块和子包的搜索.

虽然这个特性不经常需要, 但它可以用于扩展在一个包里发现的模块的集合.

第七章 输入和输出¹³

有多种方式可以展现一个程序的输出; 数据可以以一种可读的形式输出, 或者是保存于一个文件便于以后使用. 本章就将讨论几种可能.

7.1 美化输出格式



至今为止, 我们知道了两种输出值的方式: 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; 标准输出文件可以用 `sys.stdout` 引用. 参考库手册了解更多的信息.)

一般来说你会希望更多的控制其输出格式, 而不是简单的以空格分割. 有两种方式格式化你的输出. 第一种方式是由你自己控制, 使用字符串切片和连接操作, 来实现你所想象的外观. 标准模块 `string` 包含了一些有用的操作, 用以填充字符串至某一给定的宽度; 很快就会讨论这些. 第二种方式是使用 `str.format()` 方法.

`string` 模块包含了一个类模板, 提供了另一种替换字符串的方式.

还有一个问题, 当然了: 如何把值转成字符串? 幸运的是, Python 有多种方式将任何值转为字符串: 将它传给 `repr()` 或 `str()` 函数.

`str()` 函数意味着返回一个用户易读的表达式, 而 `repr()` 则意味着产生一个解释器易读的表达式 (或者如果没有这样的语法会给出 `SyntaxError`). 对于那些没有特殊表达的对象, `str()` 将会与 `repr()` 返回相同的值. 很多的值, 如数字或一些如列表和字典那样的结构, 使用这两个函数的结果完全一致. 字符串与浮点型则有两种不同的表达.

例如:

```
1 >>> s = 'Hello, \uworld.'
2 >>> str(s)
3 'Hello, \uworld.'
4 >>> repr(s)
5 "'Hello, \uworld.'"
6 >>> str(1.0/7.0)
```

¹³初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

```

7  '0.142857142857'
8  >>> repr(1.0/7.0)
9  '0.14285714285714285'
10 >>> x = 10 * 3.25
11 >>> y = 200 * 200
12 >>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr
    (y) + '...'
13 >>> print(s)
14 The value of x is 32.5, and y is 40000...
15 >>> # The repr() of a string adds string quotes and
    backslashes:
16 ... hello = 'hello, world\n'
17 >>> hellos = repr(hello)
18 >>> print(hellos)
19 'hello, world\n'
20 >>> # The argument to repr() may be any Python object:
21 ... repr((x, y, ('spam', 'eggs'))))
22 "(32.5, 40000, ('spam', 'eggs'))"

```

这里有两种方式输出一个平方与立方的表:

```

1  >>> for x in range(1, 11):
2  ...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
3  ...     # Note use of 'end' on previous line 注意前一行 'end'
    的使用
4  ...     print(repr(x*x*x).rjust(4))
5  ...
6  1    1    1
7  2    4    8
8  3    9   27
9  4   16   64
10 5   25  125
11 6   36  216
12 7   49  343
13 8   64  512
14 9   81  729
15 10  100 1000
16
17 >>> for x in range(1, 11):
18 ...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
19 ...
20 1    1    1
21 2    4    8

```

```
22 | 3    9    27
23 | 4   16   64
24 | 5   25  125
25 | 6   36  216
26 | 7   49  343
27 | 8   64  512
28 | 9   81  729
29 | 10 100 1000
```

(注意在第一个例子中, 每列间的空格是由 `print()` 添加的: 它总会在每个参数后面加个空格.)

这个例子展示了字符串对象的 `rjust()` 方法, 它可以将字符串靠右, 并在左边填充空格. 还有类似的方法, 如 `ljust()` 和 `center()`. 这些方法并不会写任何东西, 它们仅仅返回新的字符串. 如果输入很长, 它们并不会对字符串进行截断, 仅仅返回没有任何变化的字符串; 这虽然会影响你的布局, 但是这一般比截断的要好. (如果你的确需要截断, 那么就增加一个切片的操作, 如 `x.ljust(n)[:n]`.)

有另一个方法, `zfill()`, 它会在数字的左边填充 0. 它知道正负号:

```
1 >>> '12'.zfill(5)
2 '00012'
3 >>> '-3.14'.zfill(7)
4 '-003.14'
5 >>> '3.14159265359'.zfill(5)
6 '3.14159265359'
```

`str.format()` 的基本使用如下:

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

括号及其里面的字符 (称作 format field) 将会被 `format()` 中的参数替换. 在括号中的数字用于指向传入对象在 `format()` 中的位置.

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

如果在 `format()` 中使用了关键字参数, 那么它们的值会指向使用该名字的参数.

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

位置及关键字参数可以任意的结合:

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

'!a' (使用 `ascii()`), '!s' (使用 `str()`) 和 '!r' (使用 `repr()`) 可以用于在格式化某个值之前对其进行转化:

```
1 >>> import math  
2 >>> print('The value of PI is approximately {:.f}'.format(math.  
3 ...     pi))  
The value of PI is approximately 3.14159265359.  
4 >>> print('The value of PI is approximately {:.r}'.format(  
5 ...     math.pi))  
The value of PI is approximately 3.141592653589793.
```

可选项 ':' 和格式标识符可以跟着 field name. 这就允许对值进行更好的格式化. 下面的例子将 Pi 保留到小数点后三位.

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

在 ':' 后传入一个整数, 可以保证该域至少有这么多的宽度. 用于美化表格时很有用.

```
1 >>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}  
2 >>> for name, phone in table.items():  
3 ...     print('{0:10} ==> {1:10d}'.format(name, phone))  
4 ...  
5 Jack          ==>      4098  
6 Dcab          ==>      7678  
7 Sjoerd        ==>      4127
```

如果你有一个的确很长的格式化字符串, 而你不想将它们分开, 那么在格式化时通过变量名而非位置会是很好的事情. 最简单的就是传入一个字典, 然后使用方括号 '[]' 来访问键值:

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

这也可以通过在 table 变量前使用 '**' 来实现相同的功能.

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

在结合新的内置函数 `vars()` (这会以字典的形式返回所有的局部变量) 和这个时会特别有用.

要了解更多关于 `str.format()` 的知识, 参考 *Format String Syntax*.

7.1.1 旧式字符串格式化

% 操作符也可以实现字符串格式化. 它将左边的参数作为类似 `sprintf()` 式的格式化字符串, 而将右边的代入, 然后返回格式化后的字符串. 例如:

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

因为 `str.format()` 很新, 大多数的 Python 代码仍然使用 % 操作符. 但是因为这种旧式的格式化最终会从该语言中移除, 应该更多的使用 `str.format()`.

更多的信息可以在 *old-string-formatting* 中找到.

7.2 读和写文件

`open()` 将会返回一个文件对象, 并且一般使用两个参数进行使用:
`open(filename, mode)`.

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

第一个参数是包含文件名的字符串. 第二个参数是另一个字符串, 它描述文件如何使用的字符. `mode` 可以是 'r' 如果文件只读, 'w' 只用于写 (如果存在同名文件则将被删除), 和 'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾. 'r+' 同时用于读写. `mode` 参数是可选的; 'r' 将是默认值.

一般而言, 文件以 *text mode* 打开, 这就意味着, 从文件中读写的字符串, 是以一种特定的编码进行编码 (默认的是 UTF-8). 追加到 `mode` 后的 'b', 将意味着以 *binary mode* 打开文件: 现在的数据是以字节对象的形式进行读写. 这个模式应该用于那些不包含文本的文件.

在文本模式下 (text mode), 默认是将特定平台的行末标识符 (Unix 下为 `\n`, Windows 下为 `\r\n`) 在读时转为 `\n` 而写时将 `\n` 转为特定平台的标识符. 这种隐藏的行为对于文本文件是没有问题的, 但是对于二进制数据像 JPEG 或 EXE 是会出问题的. 在使用这些文件时请小心使用二进制模式.

7.2.1 文件对象的方法

本节中剩下的例子假设已经创建了一个称为 `f` 的文件对象.

为了读取一个文件的内容, 调用 `f.read(size)`, 这将读取一定数目的数据, 然后作为字符串或字节对象返回. `size` 是一个可选的数字类型的参数. 当 `size` 被忽略了或者为负, 那么该文件的所有内容都将被读取并且返回; 如果文件比你的内存大两倍, 那么就会成为你的问题了. 否则, 最多 `size` 字节将被读取并返回. 如果到达了文件的末尾, `f.read()` 将会返回一个空字符串 ('').

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



```
3 >>> f.read()
4 ''
```

`f.readline()` 会从文件中读取单独的一行; 在每个字符串的末尾都会留下换行符 (`\n`), 除非是该文件的最后一行并且没有以换行符结束, 这个字符才会被忽略. 这就使结果很明确; `f.readline()` 如果返回一个空字符串, 那么文件已到底了, 而如果是 `'\n'` 表示, 那么就是只包行一个新行.

```
1 >>> f.readline()
2 'This is the first line of the file.\n'
3 >>> f.readline()
4 'Second line of the file\n'
5 >>> f.readline()
6 ''
```

`f.readlines()` 将返回该文件中包含的所有行. 如果给定一个可选参数 `sizehint`, 它就读取这么多字节, 并且将这些字节按行分割. 这经常用于允许按行读取一个大文件, 但是不需要载入全部的文件时非常有用. 只会返回完整的行.

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

另一种方式是迭代一个文件对象然后读取每行. 这是内存有效, 快速, 并用最少的代码:

```
1 >>> for line in f:
2 ...     print(line, end='')
3 ...
4 This is the first line of the file.
5 Second line of the file
```

这个方法很简单, 但是并没有提供一个很好的控制. 因为两者的处理机制不同, 最好不要混用.

`f.write(string)` 将 `string` 写入到文件中, 然后返回写入的字符数.

```
1 >>> f.write('This is a test\n')
2 15
```

如果要写入一些不是字符串的东西, 那么将需要先进行转换:

```
1 >>> value = ('the_answer', 42)
2 >>> s = str(value)
3 >>> f.write(s)
4 18
```

`f.tell()` 返回文件对象当前所处的位置, 它是从文件开头开始算起字节数. 要改变文件当前的位置, 使用 `f.seek(offset, from_what)`. 这个位置是通过将当前位置加上 `offset` 所得. `from_what` 的值, 如果是 0 表示开头, 如果是 1 表示当前位置, 2 表示文件的结尾. `from_what` 的默认为 0, 即从开头开始.

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

在文本文件中 (那些打开文件的模式下没有 `b` 的), 只会相对于文件起始位置进行定位, (如果要定文件的最后面, 要用 `seek(0, 2)`).

当你处理完一个文件后, 调用 `f.close()` 会关闭它, 并释放系统的资源. 在调用完 `f.close()` 之后, 尝试使用那个文件对象是会失败的.

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

当处理一个文件对象时, 使用 `with` 关键字是非常好的方式. 在结束后, 它会帮你正确的关闭文件, 即使发生了异常. 而且写起来也比 `try - finally` 语句块要简短:

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

文件对象有些额外的方法, 如 `isatty()` 和 `truncate()`, 但它们都较少的使用; 更多的信息需要参考标准库手册。

7.2.2 pickle 模块

在文件中, 字符串可以很方便的读取写入. 数字可能稍微麻烦一些, 因为 `read()` 方法只返回字符串, 我们还需要将其传给 `int()` 这样的函数, 使其将如 `'123'` 的字符串转为数字 `123`. 但是, 如果要保存更复杂的数据类型, 如列表, 字典, 或者类的实例, 那么就会更复杂了.

为了让用户在时常的编程和测试时保存复杂的数据类型, Python 提供了标准模块, 称为 `pickle`. 这个模块可以将几乎任何的 Python 对象 (甚至是 Python 的代码), 转换为字符串表示; 这个过程称为 *pickling*. 而要从里面重新构造回原来的对象, 则称为 *unpickling*. 在 *pickling* 和 *unpickling* 之间, 表示这些对象的字符串表示, 可以存于一个文件, 也可以通过网络在远程机器间传输.

如果你有一个对象 `x`, 和一个已经打开并用于写的文件对象 `f`, `pickle` 这个对象最简单的方式就是使用:

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

有了 `pickle` 这个对象, 就能对 `f` 以读取的形式打开:

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

(还有其他不同的形式, 比如 *pickling* 很多对象, 或者不想保存至文件; 更多的信息参考 `pickle` 模块.)

`pickle` 是 Python 中保存及重用对象的标准方式; 标准的属于称为 *persistent* 对象 (即持久化对象). 因为 `pickle` 被广泛使用, 很多写

Python 扩展的作者都会确保, 如矩阵这样的数据类型能被合理的 pickle 和 unpickle.

第八章 错误和异常¹⁴

到现在为止, 没有更多的提及错误信息, 但是当你在尝试这些例子或多或少会碰到一些. 这里 (至少) 有两种可以分辨的错误: *syntax error* 和 *exception*, 按中文来说, 就是语法错误和异常.

8.1 语法错误

语法错误, 也可以认为是解析时错误, 这是在你学习 Python 过程中最有可能碰到的:

```
1 >>> while True print('Hello_world')
2     File "<stdin>", line 1, in ?
3         while True print('Hello_world')
4             ^
5 SyntaxError: invalid syntax
```

解析器会重复出错的那行, 然后显示一个小箭头, 指出探测到错误时最早的那个点. 错误一般是由箭头所指的地方导致 (或者至少是此处被探测到): 在这个例子中, 错误是在 `print()` 函数这里被发现的, 因为它之前少了一个冒号 (':'). 文件的名称与行号会被打印出来, 以便于你能找到一个脚本中导致错误的地方.

8.2 异常

尽管语句或表达式语法上是没有问题的, 它同样也会在尝试运行时导致一个错误. 在执行时探测到的错误被成为 *exception*, 也就是异常, 但它并不是致命的问题: 你将会很快学到如何在 Python 程序中处理它们. 大多数异常并不会被程序处理, 不过, 导致错误的信息会被显示出来:

¹⁴初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

```
1 >>> 10 * (1/0)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in ?
4 ZeroDivisionError: int division or modulo by zero
5 >>> 4 + spam*3
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in ?
8 NameError: name 'spam' is not defined
9 >>> '2' + 2
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in ?
12 TypeError: Can't convert 'int' object to str implicitly
```

每个错误信息的最后一行或说明发生了什么。异常会有很多的类型，而这个类型会作为消息的一部分打印出来：在此处的例子中的类型有 `ZeroDivisionError`、`NameError` 和 `TypeError`。作为异常类型被输出的字符串其实是发生内建异常的名称。对于所有内建异常都是那样的，但是对于用户自定义的异常，则可能不是这样（尽管有某些约定）。标准异常的名字是内建的标识符（但并不是关键字）。

改行剩下的部分则提供更详细的信息，是什么样的异常，是怎么导致的。

错误消息的前面部分指出了异常发生的上下文，以 `stack traceback`（栈追踪）的方式显示。一般来说列出了源代码的行数；但是并不会显示从标准输入得到的行数。

Built-in Exceptions 列出了内建的异常和它们的意义。

8.3 处理异常

写程序来处理异常是可能的。看看下面的例子，它请求用户输入一个合法的整数，但是也允许用户来中断程序（使用 `Control-C` 或任何操作系统支持的）；注意，用户生成的中断是通过产生异常 `KeyboardInterrupt`：

```
1 >>> while True:
2     ...     try:
3     ...         x = int(input("Please enter a number: "))
```

```
4 ...         break
5 ...     except ValueError:
6 ...         print("Oops! That was no valid number. Try again...")
7 ...
```

try 语句像下面这样使用.

- 首先, *try clause* (在 `try` 和 `except` 之间的语句) 将被执行.
- 如果没有异常发生, *except clause* 将被跳过, `try` 语句就算执行完了.
- 如果在 `try` 语句执行时, 出现了一个异常, 该语句的剩下部分将被跳过. 然后如果它的类型匹配到了 `except` 后面的异常名, 那么该异常的语句将被执行, 而执行完后会运行 `try` 后面的问题.
- 如果一个异常发生时并没有匹配到 `except` 语句中的异常名, 那么它就被传到 `try` 语句外面; 如果没有处理, 那么它就是 *unhandled exception* 并且将会像前面那样给出一个消息然后执行.

一个 `try` 语句可以有大于一条的 `except` 语句, 用以指定不同的异常. 但至多只有一个会被执行. Handler 仅仅处理在相应 `try` 语句中的异常, 而不是在同一 `try` 语句中的其他 Handler. 一个异常的语句可以同时包括多个异常名, 但需要用括号括起来, 比如:

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

最后的异常段可以忽略异常的名字, 用以处理其他的情况. 使用这个时需要特别注意, 因为它很容易屏蔽了程序中的错误! 它也用于输出错误消息, 然后重新产生异常 (让调用者处理该异常):

```
1 import sys
2
3 try:
4     f = open('myfile.txt')
5     s = f.readline()
6     i = int(s.strip())
7 except IOError as err:
8     print("I/O error: {0}".format(err))
9 except ValueError:
10    print("Could not convert data to an integer.")
```

```
11 except:
12     print("Unexpected_error:", sys.exc_info()[0])
13     raise
```

`try ... except` 语句可以有一个可选的 `else` 语句, 在这里, 必须要放在所有 `except` 语句后面. 它常用于没有产生异常时必须执行的语句. 例如:

```
1 for arg in sys.argv[1:]:
2     try:
3         f = open(arg, 'r')
4     except IOError:
5         print('cannot_open', arg)
6     else:
7         print(arg, 'has', len(f.readlines()), 'lines')
8         f.close()
```

使用 `else` 比额外的添加代码到 `try` 中要好, 因为这样可以避免偶然的捕获一个异常, 但却不是由于我们保护的代码所抛出的.

当一个异常发生了, 它可能有相关的值, 这也就是所谓的异常的参数. 该参数是否出现及其类型依赖于异常的类型.

在 `except` 语句中可以在异常名后指定一个变量. 变量会绑定值这个异常的实例上, 并且把参数存于 `instance.args`. 为了方便, 异常的实例会定义 `__str__()` 来直接将参数打印出来, 而不用引用 `.args`. 当然也可以在产生异常前, 首先实例化一个异常, 然后把需要的属性绑定给它.

```
1 >>> try:
2 ...     raise Exception('spam', 'eggs')
3 ... except Exception as inst:
4 ...     print(type(inst))    # the exception instance
5 ...     print(inst.args)     # arguments stored in .args
6 ...     print(inst)         # __str__ allows args to be
    printed directly,
7 ...                         # but may be overridden in
    exception subclasses
8 ...     x, y = inst.args     # unpack args
9 ...     print('x=', x)
10 ...    print('y=', y)
11 ...
12 <class 'Exception'>
```

```
13 ('spam', 'eggs')
14 ('spam', 'eggs')
15 x = spam
16 y = eggs
```

如果一个异常有参数, 它们将作为异常消息的最后一部分打印出来.

异常的 handler 处理的异常, 不仅仅是 try 语句中那些直接的异常, 也可以是在此处调用的函数所产生的异常. 例如:

```
1 >>> def this_fails():
2 ...     x = 1/0
3 ...
4 >>> try:
5 ...     this_fails()
6 ... except ZeroDivisionError as err:
7 ...     print('Handling run-time error:', err)
8 ...
9 Handling run-time error: int division or modulo by zero
```

8.4 抛出异常

`raise` 语句允许程序员强制一个特定的异常的发生. 举个例子:

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

给 `raise` 的唯一参数表示产生的异常. 这必须是一个异常实例或类 (派生自 `Exception` 的类).

如果你需要决定产生一个异常, 但是不准备处理它, 那么一个简单的方式就是, 重新抛出异常:

```
1 >>> try:
2 ...     raise NameError('HiThere')
3 ... except NameError:
4 ...     print('An exception flew by!')
5 ...     raise
6 ...
```



```
7 An exception flew by!
8 Traceback (most recent call last):
9   File "<stdin>", line 2, in ?
10 NameError: HiThere
```

8.5 自定义异常

程序中可以通过定义一个新的异常类 (更多的类请参考 *tut-classes*) 来命名它们自己的异常. 异常需要从 `Exception` 类派生, 既可以是直接也可以是间接. 例如:

```
1 >>> class MyError(Exception):
2 ...     def __init__(self, value):
3 ...         self.value = value
4 ...     def __str__(self):
5 ...         return repr(self.value)
6 ...
7 >>> try:
8 ...     raise MyError(2*2)
9 ... except MyError as e:
10 ...     print('My exception occurred, value:', e.value)
11 ...
12 My exception occurred, value: 4
13 >>> raise MyError('oops!')
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in ?
16 __main__.MyError: 'oops!'
```

在这个例子中, `Exception` 的默认方法 `__init__()` 被覆写了. 现在新的异常类可以像其他的类一样做任何的事, 但是常常会保持简单性, 仅仅提供一些可以被 handler 处理的异常信息. 当创建一个模块时, 可能会有多种不同的异常, 一种常用的做法就是, 创建一个基类, 然后派生出各种不同的异常:

```
1 class Error(Exception):
2     """Base class for exceptions in this module."""
3     pass
4
5 class InputError(Error):
```

```
6      """Exception raised for errors in the input.
7
8      Attributes:
9          expression -- input expression in which the error
              occurred
10         message -- explanation of the error
11     """
12
13     def __init__(self, expression, message):
14         self.expression = expression
15         self.message = message
16
17 class TransitionError(Error):
18     """Raised when an operation attempts a state transition
19         that's not
20         allowed.
21
22     Attributes:
23         previous -- state at beginning of transition
24         next -- attempted new state
25         message -- explanation of why the specific transition
26                     is not allowed
27     """
28
29     def __init__(self, previous, next, message):
30         self.previous = previous
31         self.next = next
32         self.message = message
```

大多数异常定义时都会以“Error”结尾, 就像标准异常的命名。

大多数标准模块都定义了它们自己的异常, 用于报告在它们定义的函数中发生的错误. 关于更多类的信息请参考 *tut-classes*.

8.6 定义清理动作

`try` 语句有另一种可选的从句, 用于定义一些扫尾的工作, 此处定义的语句在任何情况下都会被执行. 例如:

```
1 >>> try:
2 ...     raise KeyboardInterrupt
```

```
3 ... finally:
4 ...     print('Goodbye, world!')
5 ...
6 Goodbye, world!
7 KeyboardInterrupt
```

一个 `*finally` 语句 * 总是在离开 `try` 语句前被执行, 而无论此处有无异常发生. 当一个异常在 `try` 中产生, 但是并没有被 `except` 处理 (或者发生在 `except` 或 `else` 语句中), 那么在 `finally` 语句执行后会被重新抛出. `finally` 语句在其他语句要退出 `try` 时也会被执行, 像是使用 `break`, `continue` 或者 `return`. 一个更复杂的例子:

```
1 >>> def divide(x, y):
2 ...     try:
3 ...         result = x / y
4 ...     except ZeroDivisionError:
5 ...         print("division by zero!")
6 ...     else:
7 ...         print("result is", result)
8 ...     finally:
9 ...         print("executing finally clause")
10 ...
11 >>> divide(2, 1)
12 result is 2.0
13 executing finally clause
14 >>> divide(2, 0)
15 division by zero!
16 executing finally clause
17 >>> divide("2", "1")
18 executing finally clause
19 Traceback (most recent call last):
20   File "<stdin>", line 1, in ?
21   File "<stdin>", line 3, in divide
22 TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

正如你所看到的, `finally` 语句在任何情况下都被执行了. 由于将两个字符串相除而产生的 `TypeError` 并没有被 `except` 语句处理, 因此在执行 `finally` 后被重新抛出.

在真正的应用中, `finally` 是非常有用的, 特别是释放额外的资源 (想文件或网络连接), 无论此资源是否成功使用.

8.7 预定义的清理动作

有些对象定义了标准的清理工作, 特别是对对象不再需要时, 无论对其使用的操作是否成功. 看看下面的例子, 它尝试打开一个文件并输出内容到屏幕.

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

前面这段代码的问题在于, 在此代码成功执行后, 文件依然被打开着. 在简单的脚本中这可能不是什么问题, 但是对于更大的应用来说却是个问题. `with` 语句就允许像文件这样的对象在使用后会被正常的清理掉.

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

在执行该语句后, 文件 `f` 就会被关闭, 就算是在读取时碰到了问题. 像文件这样的对象, 总会提供预定义的清理工作, 更多的可以参考它们的文档.

第九章 类¹⁵

同别的编程语言相比, Python 的类机制中增加了少量新的语法和语义. 它是 C++ 的类机制和 Modula-3 的类机制的混合体. Python 类提供了面向对象编程的所有基本特征: 允许多继承的类继承机制, 派生类可以重写它父类的任何方法, 一个方法可以调用父类中重名的方法. 对象可以包含任意数量和类型的数据成员. 作为模块, 类也拥有 Python 的动态特征: 他们可以被动态创建, 并且可以在创建之后被修改.

¹⁵初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

从 C++ 术语上讲, Python 类的成员 (包括数据成员) 通常都是 *public* 的 (例外见下私有变量 (page 95)), 并且所有的成员函数都是 *virtual* 的. 和 Modula-3 中一样, Python 中没有关联对象成员和方法的隐式表达: 所有方法函数在声明时显式地将第一个参数表示为对象, 这个参数的值在方法被调用时隐式赋值. 同 Smalltalk 类似, Python 类本身就是对象. 这就提供了导入和重命名的语义. 与 C++ 和 Modula-3 不同的是, Python 的内置类可以被当做基类来让使用者扩展. 另外, 像 C++ 一样, 大多数有特殊语法的内置操作符 (算数运算符, 下标操作符等等) 在类的实例中都可以重定义.

(由于在谈论类的时候缺乏公认的术语, 我会偶尔使用 Smalltalk 和 C++ 的术语. 我更愿意用 Modula-3 的术语, 因为它面向对象的语义比 C++ 更贴近 Python, 但是我估计没有读者听过这个说法.)

9.1 关于名称和对象的讨论

对象都具有个别性, 多个名称 (在多个作用域中) 可以被绑定到同一个对象上. 这就是其他语言中所谓的别名. 通常第一次接触 Python 可能不会意识到这一点, 而且在处理不变的基本类型 (数值, 字符串, 元组) 时这一点可能会被安全的忽略. 但是, 在涉及到可变对象如 lists, dictionaries, 以及大多数其他类型时, 别名可能会在 Python 代码的语义上起到惊人的效果. 别名通常对编程有益处, 因为别名在某些方面表现得像指针. 比如, 由于在实现的时候传递的是指针, 所以传递一个对象的开销很小; 又比如将对象作为参数传递给一个函数来对它进行修改, 调用者将会看到对象的变化——这就消除了像 Pascal 语言中的两个不同参数之间的传递机制的必要.

9.2 Python 的作用域和命名空间

在介绍类之前, 我必须先告诉你一些关于 Python 作用域规则的事. 类定义用命名空间玩了一些巧妙的把戏, 而你为了完全理解发生了什么就必须知道命名空间和作用域是怎么工作的. 顺便说一下, 这一主题的知识对任何高级 Python 程序员都是有用的.

让我们从定义开始。

命名空间是从名称到对象的映射。大多数命名空间现在的实现就如同 Python 的字典，但通常这一点并不明显（除了在性能上），而且它有可能在将来发生改变。

顺便说一下，我用了属性这个词来称呼任何点后面跟的名称——比如在表达式 `z.real` 中，`real` 就是对象 `z` 的属性。更直接的说，对模块中名称的引用就是属性引用：在表达式 `modname.funcname` 中，`modname` 是模块对象而 `funcname` 是它的一个属性。在这种情况下模块的属性和它里面所定义的全局名称之间就刚好有一个直接的映射关系：他们共享同一个命名空间！¹⁶

属性可以是只读的或可写的。在后一种情况下，给属性赋值才是可能的。模块属性是可写的：你可以写 `modname.the_answer = 42`。可以利用 `keyword:del` 语句来删除可写属性。例如，`del modname.the_answer` 将名为 `modname` 的模块中移除属性 `the_answer`。

命名空间们是在不同时刻创建的，并且有着不同的生命期。包含内置名称的命名空间是在 Python 解释器启动时创建的，而且它永远不被删除。一个模块的全局命名空间在模块的定义被读取的时候创建；通常情况下，模块的命名空间一直持续到解释器退出时。被最高级别的解释器调用的语句，不论是从脚本还是从交互读取的，都被认为是一个名叫 `__main__` 的模块的一部分，所以它们有自己的全局命名空间。（内置名称实际上也存在于一个模块中；这个模块叫 `builtins`。）

函数的局部命名空间在函数调用时被创建，在函数返回时或者发生异常而终止时被删除。（事实上，忘记可能是更好的方式来描述真正发生了什么。）当然，递归调用会有它们自己的局部命名空间。

在 Python 中，一个作用域只是一个结构上的区域，在这里命名空间可以直接访问。“直接访问”就意味着无须特殊的指明引用。

尽管作用域是静态的决定的，它们使用时却是动态的。在执行时的任何

¹⁶ 除了一种情况。Module 对象有一个私有的只读属性，名为 `__dict__`，它返回实现这个模块命名空间的字典。显然，使用这会违反命名空间实现的抽象，而只应当限于在如 `post-mortem debuggers` 的事情中使用。

时刻, 至少有三个嵌套的作用域其命名空间可以直接访问:

- 最内层的作用域, 首先被搜索, 包含局部变量名
- 任意函数的作用域, 它从最接近的作用域开始搜索, 包括非局部的, 但也是非全局的名字
- 紧邻最后的作用域包含了当前模块的全局变量
- 最外层的作用域 (最后搜索) 是包含内置名字的命名空间

如果一个名字在全局声明, 那么所有的引用和赋值都直接到这个模块的全局名中. 为了在最内部作用域中重新绑定变量, `nonlocal` 语句就可以使用了; 如果没有声明 `nonlocal`, 那些变量只是只读 (尝试给这样的变量赋值, 只是会简单的创建一个新的局部变量, 而外部的并没有什么改变) 重新绑定.

一般来说, 局部作用域引用当前函数的局部变量名. 在函数外部, 局部变量引用和全局作用域相同的命名空间: 模块的命名空间. 类定义又放置了另一个命名空间.

意识到作用域是在结构上被决定的这很重要. 一个定义在模块中的函数的全局作用域, 就是模块的命名空间, 无论它从哪里被访问. 另一个方面, 搜寻名字的过程是动态完成的, 在运行时——但是, 语言的定义一般是静态的, 在“编译”时完成, 所以不要依赖动态命名! (事实上, 局部变量都是静态的被决定的.)

Python 的一个怪事就是 – 如果 `global` 语句没有起效果——赋值总是会使用最里层作用域的值. 赋值并没有拷贝数据——它们仅仅是绑定名字到对象上. 删除也是如此: `del x` 移除了 `x` 从局部作用域的绑定. 事实上, 所有操作引入新的名字都使用局部作用域: 特别的, `import` 语句, 和函数定义都将模块或函数绑定到了当前作用域.

`global` 语句可以用于指示, 在全局作用域中的变量可以在这里重新绑定; `nonlocal` 则表示在一个闭合的作用域中的变量可以在此处绑定.

9.2.1 域和命名空间的例子

这是一个例子用于说明如何引用不同的作用域和命名空间, `global` 和 `nonlocal` 如何影响变量绑定:

```
1 def scope_test():
2     def do_local():
3         spam = "local_spam"
4     def do_nonlocal():
5         nonlocal spam
6         spam = "nonlocal_spam"
7     def do_global():
8         global spam
9         spam = "global_spam"
10
11     spam = "test_spam"
12     do_local()
13     print("After local assignment:", spam)
14     do_nonlocal()
15     print("After nonlocal assignment:", spam)
16     do_global()
17     print("After global assignment:", spam)
18
19 scope_test()
20 print("In global scope:", spam)
```

输出的结果是:

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

注意局部的赋值 (默认) 并没有改变 `scope_test` 绑定的 `spam`. 而 `nonlocal` 则改变了 `scope_test` 中的 `spam`, 而 `global` 则改变了模块级别的绑定.


你可以看到在 `global` 赋值之前并没有绑定 `spam` 的值.

9.3 类的初印象

类引入了一些新的语法, 三种新的对象类型, 和一些新的语义.

9.3.1 类定义的语法

最简单的类的定义形式看起来像这样:



```
1 class ClassName:
2     <statement-1>
3     .
4     .
5     .
6     <statement-N>
```

类的定义, 和函数定义 (`def` 语句) 一样必须在使用它们前执行. (你可以将一个类定义放置于 `if` 语句的分支中, 或一个函数中.)

事实上, 类定义内部的语句一般是函数的定义, 但其他的语句也是允许的, 而且还很有用——我们在后面将会继续讨论该问题. 类内的函数定义一般有一个特殊形式的参数列表, 习惯上称之为方法——同样, 也将在后面解释.

当进入一个类定义, 新的命名空间就被创建了, 这一般作为局部的作用域——因此, 所有的局部变量都在这个新的作用域中. 特别是, 函数定义会绑定.

当离开一个类定义后, 一个 *class object* 就被创建. 通过类的定义, 就将这个命名空间包装了起来; 我们将在后面学到更多关于类对象的知识. 原来的局部作用域 (在进入一个类定义前的作用域) 将会复位, 而类对象就会在这里绑定, 并且命名为类定义时的名字 (在此例中是 `ClassName`).

9.3.2 类对象

类对象支持两种操作: 属性引用和实例化.

属性引用使用的语法和 Python 中所有的属性引用一样. 合法的属性名是那些在类的命名空间中定义的名字. 所以一个类定义如果是这样:

```
1 class MyClass:
2     """A simple example class"""
3     i = 12345
4     def f(self):
5         return 'hello_world'
```

那么, `MyClass.i` 和 `MyClass.f` 就是合法的属性引用, 分别返回一个数和一个函数对象. 类属性也可以被指定, 所以你可以给 `MyClass.i` 赋值以改变其数值. `__doc__` 也是一个合法的属性, 返回属于这个类的 docstring: "A simple example class".

类的实例化使用函数的形式. 只要当作一个无参的函数然后返回一个类的实例就可以了. 比如 (假设有前面的类了):

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

创建了一个新的实例, 并且将其指定给局部变量 `x`.

实例化的操作 (“调用” 一个类对象) 创建了空的对象. 在创建实例时, 很多类可能都需要有特定的初始状态. 所以一个类可以定义一个特殊的方法, 称为 `__init__()`, 像这样:

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

当一个类定义了 `__init__()` 方法, 类在实例化时会自动调用 `__init__()` 方法, 用于创建新的类实例. 所以在这个例子中, 一个新的初始化过的实例被创建:

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

当然, 为了更大的灵活性, 方法 `__init__()` 可以有更多的参数. 在这种情况下, 给类的参数会传给 `__init__()`. 例如,

```
1 >>> class Complex:
2 ...     def __init__(self, realpart, imagpart):
3 ...         self.r = realpart
4 ...         self.i = imagpart
5 ...
```

```
6 >>> x = Complex(3.0, -4.5)
7 >>> x.r, x.i
8 (3.0, -4.5)
```

9.3.3 实例对象

那么我们现在可以对实例对象做什么？实例对象唯一能理解的操作是属性引用。有两种合法的属性，数据属性和方法。

data attribute 在 Smalltalk 中相应于 “instance variable”，在 C++ 中相应于 “data member”。数据属性不需要声明；像局部变量，当它们第一次指定时就会被引入。比如，如果 *x* 是前面创建的 *MyClass* 的实例，那么下面的例子就会打印出 16，而不会有问题：

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

实例属性引用的另一种是方法。一个方法就是“属于”一个对象的函数。（在 Python 中，方法的概念并不是类实例所特有：其他对象类型也可以有方法。例如，列表对象有 *append*, *insert*, *remove*, *sort*, 及等等的方法。但是，在下面的讨论中，我们指的就是类实例对象的方法，除非特别指出。）

合法的方法名依赖于实例的类。在定义中，类的属性如果是那些定义的函数对象，而这也就是实例的方法。所以在我们的例子中，*x.f* 是一个合法的方法引用，因为 *MyClass.f* 是一个函数，但是 *x.i* 就不是，因为 *MyClass.i* 就不是。但是 *x.f* 和 *MyClass.f* 并不一样——它是一个 *method object*，而不是 *function object*。

9.3.4 方法对象

通常，一个方法在其绑定后就可以调用了：

```
1 x.f()
```

在 `MyClass` 这个例子中, 这将会返回字符串 `'hello world'`. 但是, 像这样的调用并不是必须的: `x.f` 是一个方法对象, 它可以被保存起来以供下次调用. 例如:

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

将会持续的打印 `'hello world'`.

那么在方法调用是发生了什么? 你可能注意到 `x.f()` 调用时并没有参数, 尽管 `f()` 定义时是有一个参数的. 那么这个参数怎么了? 当然, Python 在一个参数缺少时调用一个函数是会发生异常的——就算这个参数没有真正用到...

事实上, 你会猜想到: 关于方法, 特殊的东西就是, 对象作为参数传递给了函数的第一个参数. 在我们的例子中, `x.f()` 是严格等价于 `MyClass.f(x)`. 在多数情况下, 调用一个方法 (有个 n 个参数), 和调用相应的函数 (也有那 n 个参数, 但是再额外加入一个使用该方法的对象), 是等价的.

如果你仍然不知道方法如何工作, 那么看看实现或许会解决这些问题. 当一个实例属性被引用时, 但是不是数据属性, 那么它的类将被搜索. 如果该名字代表一个合法的类属性并且是一个函数对象, 一个方法对象就会被创建, 通过包装 (指向) 实例对象, 而函数对象仍然只是在抽象的对象中: 这就是方法对象. 当方法对象用一个参数列表调用, 新的参数列表会从实例对象中重新构建, 然后函数对象则调用新的参数列表.

9.4 随机备注

数据属性覆写了同名的方法属性; 为了避免这个偶然的名称冲突, 在大型的程序中这会导致很难寻找的 bug, 使用某些命名约定是非常明智的, 这样可以最小的避免冲突. 可能的约定包括大写方法名称, 在数据类型前增加特殊的前缀 (或者就是一个下划线), 或对于方法使用动词, 而数据成员则使用名词.

数据属性可以被该类的方法或者普通的用户 (“客户”) 引用. 换句话说, 类是不能实现完全的抽象数据类型. 事实上, 在 Python 中没有任何东西是强制隐藏的——这完全是基于约定. (在另一方面, Python 是用 C 实现的, 这样就可以实现细节的隐藏和控制访问; 这可以通过编写 Python 的扩展实现.)

客户需要小心地使用数据属性——客户会弄乱被方法控制的不变. 通过使用它们自己的方法属性. 注意用户可以增加它们自己的数据到实例对象上, 而没有检查有没有影响方法的有效性, 只要避免名字冲突 – 在说一次, 命名约定可以避免很多这样令人头疼的问题.

在引用数据属性 (或其他方法!) 并没有快速的方法. 我发现这的确增加了方法的可读性: 这样就不会被局部变量和实例中的变量所困惑, 特别是在随便看看一个方法时.

通常, 方法的第一个参数称为 `self`. 这更多的只是约定: `self` 对于 Python 来说没有任何意义. 但注意, 如果不遵循这个约定, 对于其他的程序员来说就比较难以理解了, 一个 `class browser` 程序可能会依赖此约定.

作为类属性的任何函数对象, 定义了一个方法用于那个类的实例. 函数是否在一个类体中其实并不重要: 指定一个函数对象给类中的局部变量也是可以的. 例如:

```
1 # Function defined outside the class
2 def f1(self, x, y):
3     return min(x, x+y)
4
5 class C:
6     f = f1
7     def g(self):
8         return 'hello_world'
9     h = g
```

现在 `f`, `g` 和 `h` 都是类 `C` 的属性, 并且指向函数对象, 而且都是类 `C` 实例的方法——`h` 和 `g` 是等价的. 注意这个只会是读者感到困惑.

方法可以通过使用 `self` 参数调用其他的方法:

```
1 class Bag:
```

```

2     def __init__(self):
3         self.data = []
4     def add(self, x):
5         self.data.append(x)
6     def addtwice(self, x):
7         self.add(x)
8         self.add(x)

```

方法可以引用全局变量, 就像普通函数中那样. 与这个方法相关的局部作用域, 是包含那个类定义的模块. (类本身永远不会作为全局作用域使用.) 如果的确需要在方法中使用全局数据, 那么需要合法的使用: 首先一件事, 被导入全局作用域的函数和模块可以被方法使用, 就如定义在里面的函数和类一样. 通常来说, 定义在全局作用域中, 包含方法的类是它自己本身, 并且在后面我们会知道为何方法应该引用自己的类.

每个值都是一个对象, 所以对于 *class* (或称为它的 *type*) 也是这样. 它存于 `object.__class__`.

9.5 继承

当然, 一个有 “class” 的语言如果没有继承就没有多大的价值了. 派生类的定义如下:

```

1 class DerivedClassName(BaseClassName):
2     <statement-1>
3     .
4     .
5     .
6     <statement-N>

```

`BaseClassName` 的定义对于派生类而言必须是可见的. 在基类的地方, 任意的表达式都是允许的. 这就会非常有用, 比如基类定义在另一个模块:

```

1 class DerivedClassName(modname.BaseClassName):

```

派生类就可以像基类一样使用. 当一个类被构建, 那么它就会记下基类. 这是用于解决属性引用的问题: 当一个属性在这个类中没有被找到, 那

么就会去基类中寻找. 然后搜索就会递归, 因为如果基类本身也是从其他的派生.

实例化一个派生类没有什么特别: `DerivedClassName()` 会创建这个类的新实例. 方法的引用如下: 相应的类的属性会被搜寻, 如果需要回去搜寻基类, 如果返回一个函数对象, 那么这个引用就是合法的.

派生类会覆写基类的方法. 因为当调用同样的对象的其他方法时, 并没有什么特别的, 基类的方法会因为先调用派生类的方法而被覆写. (对于 C++ 程序员: 所有的方法在 Python 中都是 `virtual` 的.)

一个在派生类中覆写的方法可能需要基类的方法. 最简单的方式就是直接调用基类的方法: 调用 `BaseClassName.methodname(self, arguments)`. 这对于可续来说也是很方便的. (这仅在 `BaseClassName` 可访问时才有效.)

Python 有两个内置函数用于继承:

- 使用 `isinstance()` 检查实例的类型: `isinstance(obj, int)` 只有在 `obj.__class__` 是 `int` 或其派生类时才为 `True`.
- 使用 `issubclass()` 用于检查类的继承关系: `issubclass(bool, int)` 会返回 `True`, 因为 `bool` 是 `int` 的派生类. 但是, `issubclass(float, int)` 会是 `False` 因为 `float` 并不是 `int` 的派生类.

9.5.1 多重继承

Python 支持多重继承. 一个多重继承的类定义看起来像这样:

```
1 class DerivedClassName(Base1, Base2, Base3):
2     <statement-1>
3     .
4     .
5     .
6     <statement-N>
```

对于大多数目的, 在最简单的情况下, 你可以将属性搜寻的方式是, 从下至上, 从左到右, 在继承体系中, 同样的类只会被搜寻一次. 如果一个属性在 `DerivedClassName` 中没有被找到, 它就会搜寻 `Base1`, 然后 (递归地)

搜寻 Base1 的基类, 然后如果还是没有找到, 那么就会搜索 Base2, 等等.

事实上, 这更加的复杂; 方法的搜寻顺序会根据调用 `super()` 而变化. 这个方法在某些其他多重继承的语言中以 `call-next-method` 被熟知, 而且比单继承的语言中要有用.

动态的顺序是很有必要的, 因为在那些处于菱形继承体系中 (这里至少有个父类被多次派生). 比如, 所有的类都从 `object` 派生, 所以到达 `object` 的路径不止一条. 为了防止基类被多次访问, 动态的算法线性化了搜寻的路径, 先从左至右搜索指定的类, 然后这样就可以让每个父类只搜寻一次, 并且单一 (这就意味一个类可以被派生, 但是不会影响其父类的搜寻路径. 使用了这些, 就使得以多重继承设计的类更可靠和可扩展. 具体参考 <http://www.python.org/download/releases/2.3/mro/>.

9.6 私有变量

在 Python 之中, 并不存在那种无法访问的“私有”变量. 但是, 在数多的 Python 代码中有一个约定: 以一个下划线带头的名字 (如 `_spam`) 应该作为非公共的 API (不管是函数, 方法或者数据成员). 这应该作为具体的实现, 而且变化它也无须提醒.

因为有一个合法的情况用于使用私有的成员 (名义上是说在派生类中避免名字的冲突), 因此就有这样的一种机制称为 *name mangling*. 任何如 `__spam` 形式的标识符, (在开头至少有两个下划线) 将被替换为 `_classname__spam`, 此处的 `classname` 就是当前的类. 这样的处理无须关注标识符的句法上的位置, 尽管它是在一个类的定义中.

注意, 这样的规则只是用于防止冲突; 它仍然可以访问或修改, 尽管认为这是一个私有变量. 在某些特殊情况下, 如测试等, 是有用的.

注意, 传递给 `exec()` 或 `eval()` 的代码并不会考虑被调用类的类名是当前的类; 这个和 `global` 语句的效果一样, 字节编译的代码也有同样的限制. 而对于 `getattr()`, `setattr()` 和 `delattr()` 也有这种限制, 直接访问 `__dict__` 也是有这样的问题.

9.7 杂物

有些时候, 有类似于 Pascal 的 “record” 或 C 的 “struct” 这样的数据类型非常有用, 绑定一些命名的数据. 一个空的类定义就将很好:

```
1 class Employee:
2     pass
3
4 john = Employee() # Create an empty employee record
5
6 # Fill the fields of the record
7 john.name = 'John_Doe'
8 john.dept = 'computer_lab'
9 john.salary = 1000
```

一段 Python 代码中如果希望一个抽象的数据类型, 那么可以通过传递一个类给那个方法, 就好像有了那个数据类型一样.

例如, 如果你有一个函数用于格式化某些从文件对象中读取的数据, 你可以定义一个类, 然后有方法 `read()` 和 `readline()` 用于读取数据, 然后将这个类作为一个参数传递给那个函数.

实例方法对象也有属性: `m.__self__` 就是一个方法 `m()` 的实例对象, 而 `m.__func__` 是相应于该方法的函数对象.

9.8 异常也是类

用户定义的异常其实也是类. 使用这个机制, 就可以创建可扩展的异常继承体系.

有两种合法的形式用于 `raise` 语句:

```
1 raise Class
2
3 raise Instance
```

在第一种形式下, `Class` 必须是 `type` 的实例或者其派生. 第一种形式可以简化为这样这样:

```
1 raise Class()
```

一个在 `except` 中的类, 可以与一个异常相容, 如果该异常是同样的类, 或是它的基类 (但是并不是另一种 – 一个 `except` 语句列出的派生类与其基类并不相容). 如下面的代码, 以那种顺序打印出 B, C, D:

```
1 class B(Exception):
2     pass
3 class C(B):
4     pass
5 class D(C):
6     pass
7
8 for c in [B, C, D]:
9     try:
10        raise c()
11    except D:
12        print("D")
13    except C:
14        print("C")
15    except B:
16        print("B")
```

但是注意, 如果 `except` 语句是反着的 (先用 `except B`), 那么打印的结果将是 B, B, B – 第一个总是匹配.

当因为一个未处理的异常发生时, 错误信息将被打印, 异常类名将被打印, 然后是一个冒号和空格, 最后是使用 `str()` 转换后的实例.

9.9 迭代器

到目前为止, 你可能注意到, 大多数的容器对象都可以使用 `for` 来迭代:

```
1 for element in [1, 2, 3]:
2     print(element)
3 for element in (1, 2, 3):
4     print(element)
5 for key in {'one':1, 'two':2}:
6     print(key)
7 for char in "123":
8     print(char)
```

```

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

```

这种形式简洁,明了并且方便. 迭代器的使用遍布于 Python 之中. 在这个外表之下, `for` 语句对容器对象调用了 `iter()`. 这个函数返回一个迭代器对象, 它定义了 `__next__()` 方法, 用以在每次访问时得到一个元素. 当没有任何元素时, `__next__()` 将产生 `StopIteration` 异常, 它告诉停止迭代. 你可以使用内置函数 `next()` 来调用 `__next__()` 方法; 这个例子展示了它如何工作:

```

1  >>> s = 'abc'
2  >>> it = iter(s)
3  >>> it
4  <iterator object at 0x00A1DB50>
5  >>> next(it)
6  'a'
7  >>> next(it)
8  'b'
9  >>> next(it)
10 'c'
11 >>> next(it)
12
13 Traceback (most recent call last):
14   File "<stdin>", line 1, in ?
15     next(it)
16 StopIteration

```

在看到迭代器的机制之后, 就可以很简单的将迭代行为增加到你的类中. 定义一个 `__iter__()` 方法用以返回一个具有 `__next__()` 的对象. 如果这个类定义了 `__next__()`, 那么 `__iter__()` 仅需要返回 `self`:

```

1  class Reverse:
2      "Iterator for looping over a sequence backwards"
3      def __init__(self, data):
4          self.data = data
5          self.index = len(data)
6      def __iter__(self):
7          return self
8      def __next__(self):
9          if self.index == 0:

```

```
10         raise StopIteration
11         self.index = self.index - 1
12         return self.data[self.index]
13
14 >>> rev = Reverse('spam')
15 >>> iter(rev)
16 <__main__.Reverse object at 0x00A1DB50>
17 >>> for char in rev:
18     ...     print(char)
19     ...
20 m
21 a
22 p
23 s
```



9.10 发生器

Generator (生成器) 是一个用于创建迭代器简单而且强大的工具。它们和普通的函数很像, 但是当它们需要返回值时, 则使用 `yield` 语句。每次 `next()` 被调用时, 生成器会从它上次离开的地方继续执行 (它会记住所有的数据值和最后一次执行的语句)。一个例子用以展示如何创建生成器:

```
1 def reverse(data):
2     for index in range(len(data)-1, -1, -1):
3         yield data[index]
4
5 >>> for char in reverse('golf'):
6     ...     print(char)
7     ...
8 f
9 l
10 o
11 g
```

任何可用生成器实现的东西都能用基于迭代器的类实现, 这个在前面有所描述。让生成器看起来很紧密的原因是它自动创建了 `__iter__()` 和 `__next__()`。

另一个关键的特性在于, 局部变量和执行状态都被自动保存下来. 这就使函数更容易编写并且更加清晰, 相对于使用实例的变量, 如 `self.index` 和 `self.data`.

除了自动创建方法和保存程序状态, 当生成器终止时, 它们会自动产生 `StopIteration` 异常. 在这些结合起来后, 这就使得能够很简单的创建迭代器, 除了仅需要编写一个函数.



9.11 生成器表达式

有些简单的生成器可以简洁的写出来, 而且和列表推导很类似, 仅仅是将方括号换成了圆括号. 这些表达式设计用于在一个函数中正好可以用生成器的情况. 生成器表达式更加紧密, 但是功能相对来说也少点, 并且与同样的列表推导式来说更节约内存.

例子:

```
1 >>> sum(i*i for i in range(10))           # sum of
    squares
2 285
3
4 >>> xvec = [10, 20, 30]
5 >>> yvec = [7, 5, 3]
6 >>> sum(x*y for x,y in zip(xvec, yvec))    # dot product
7 260
8
9 >>> from math import pi, sin
10 >>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}
11
12 >>> unique_words = set(word for line in page for word in
    line.split())
13
14 >>> valedictorian = max((student.gpa, student.name) for
    student in graduates)
15
16 >>> data = 'golf'
17 >>> list(data[i] for i in range(len(data)-1, -1, -1))
18 ['f', 'l', 'o', 'g']
```

第十章 标准库的简明介绍¹⁷

10.1 与操作系统的接口

`os` 模块提供了许多与操作系统交互的接口:

```
1 >>> import os
2 >>> os.getcwd()      # 返回当前工作目录 (current working
    directory)
3 'C:\\Python32'
4 >>> os.chdir('/server/accesslogs')  # 改变当前工作目录
5 >>> os.system('mkdir_today')  # 在系统的 shell 中运行 mkdir 命令
6 0
```

记住要使用 `import os` 这种风格而不是 `from os import *`. 这样会避免使得 `fun:os.open` 覆盖了功能截然不同的 `open()`.

内置函数 `dir()` 和 `help()` 对于处理像 `os::` 这样的大型模块来说是一种十分有效的工具:

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

对于日常文件和目录的管理, `shutil` 模块提供了更便捷、更高层次的接口:

```
1 >>> import shutil
```

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

¹⁷初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

10.2 文件的通配符

`glob` 模块提供了这样一个函数, 这个函数使我们能以通配符的方式搜索某个目录下的特定文件, 并列出它们:

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

10.3 命令行参数

一些实用的脚本通常需要处理命令行参数. 这些参数被 `sys` 模块的 `argv` 属性以列表的方式存储起来. 下例中, 命令行中运行 `python demo.py one two three`, 其结果便能说明这一点:

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

`getopt` 模块使用 Unix 通常用的 `getopt()` 函数去处理 `sys.argv`. `argparse` 提供了更强大且更灵活的处理方法.

10.4 错误的重定向输出和程序的终止

`sys` 模块还包括了 `stdin`, `stdout`, `stderr` 属性. 而最后一个属性 `stderr` 可以有效地使警告和出错信息以可见的方式传输出来, 即使是 `stdout` 被重定向了:

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

最直接地结束整个程序的方法是调用 `sys.exit()`.

10.5 字符串模式的匹配

`re` 模块提供了一些正则表达式工具, 以便对字符串做进一步的处理。对于复杂的匹配和操作, 正则表达式提供了简洁的、优化了的解决方法:

```
1 >>> import re
2 >>> re.findall(r'\b[a-z]*', 'which_foot_or_hand_fell_fastest')
3 ['foot', 'fell', 'fastest']
4 >>> re.sub(r'(\b[a-z]+)_\1', r'\1', 'cat_in_the_the_hat')
5 'cat_in_the_hat'
```

而当只需要一些简单功能的时候, 我们更倾向于字符串方法, 因为它们更容易阅读和调试:

```
1 >>> 'tea_for_too'.replace('too', 'two')
2 'tea_for_two'
```

10.6 数学处理

`math` 模块使我们可以访问底层的 C 语言库里关于浮点数的一些函数:

```
1 >>> import math
2 >>> math.cos(math.pi / 4)
3 0.70710678118654757
4 >>> math.log(1024, 2)
5 10.0
```

`random` 模块提供了产生随机数的工具:

```
1 >>> import random
2 >>> random.choice(['apple', 'pear', 'banana'])
3 'apple'
4 >>> random.sample(range(100), 10)    #
    生成无需更换的随机抽样样本
5 [30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
6 >>> random.random()    # 生成随机浮点数
7 0.17970987693706186
8 >>> random.randrange(6)    # 以 range(6)
    里的数为基准生成随机整数
9 4
```


Scipy 项目 <<http://scipy.org>> 里有许多关于数值计算的模块.

10.7 访问互联网

python 里包含了许多访问互联网和处理互联网协议的模块. 其中最简单的两个分别是, 从网址中检索数据的 `urllib.request` 模块, 和发送邮件的 `smtplib` 模块:

```
1 >>> from urllib.request import urlopen
2 >>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/
   timer.pl'):
3 ...     line = line.decode('utf-8') #
   将二进制文件解码成普通字符
4 ...     if 'EST' in line or 'EDT' in line: #
   查找西方国家的时间
5 ...         print(line)
6
7 <BR>Nov. 25, 09:43:32 PM EST
8
9 >>> import smtplib
10 >>> server = smtplib.SMTP('localhost')
11 >>> server.sendmail('soothsayer@example.org', '
   jcaesar@example.org',
12 ... """To: jcaesar@example.org
13 ... From: soothsayer@example.org
14 ...
15 ... Beware the Ides of March.
16 ... """)
17 >>> server.quit()
```

(注意: 第二个例子需要本地有一个邮件服务器.)

10.8 日期和时间

`datetime` 模块提供了操作日期和时间的类, 包括了简单和复杂两种方式. 当我们知道了时间和日期的算法后, 工作的重心便放在了如何有效地格式化输出和操作之上了. 该模块也提供了区分时区的对象.

```
1 >>> # dates are easily constructed and formatted
```

```
2 >>> from datetime import date
3 >>> now = date.today()
4 >>> now
5 datetime.date(2003, 12, 2)
6 >>> now.strftime("%m-%d-%y.%d_%b_%Y_is_a_%A_on_the_%d_day_of
   %B.")
7 '12-02-03.02_Dec_2003_is_a_Tuesday_on_the_02_day_of_December
   .'
8
9 >>> # dates support calendar arithmetic
10 >>> birthday = date(1964, 7, 31)
11 >>> age = now - birthday
12 >>> age.days
13 14368
```

10.9 数据的压缩

有些模块可以支持常规的数据压缩和解压, 这些模块包括: `zlib`, `gzip`, `zipfile` 和 `tarfile`.

```
1 >>> import zlib
2 >>> s = b'witch_which_has_which_witches_wrist_watch'
3 >>> len(s)
4 41
5 >>> t = zlib.compress(s)
6 >>> len(t)
7 37
8 >>> zlib.decompress(t)
9 b'witch_which_has_which_witches_wrist_watch'
10 >>> zlib.crc32(s)
11 226805979
```

10.10 性能测试

一些 Python 的使用者对相同问题的不同解决方法的相对性能优劣有着极大的兴趣. 而 Python 也提供了一些测试工具, 使得这些问题的答案一目了然.

例如, 我们会使用元组的打包和解包的特性而不是传统的方法去交换参数. `timeit` 模块可以很快地显示出性能上的优势, 即使这些优势很微小:

```
1 >>> from timeit import Timer
2 >>> Timer('t=a;_a=b;_b=t', 'a=1;_b=2').timeit()
3 0.57535828626024577
4 >>> Timer('a,b=_b,a', 'a=1;_b=2').timeit()
5 0.54962537085770791
```

和 `timeit` 良好的精确性不同的是, `profile` 模块和 `pstats` 模块提供了一些以大块代码的方式辨别临界时间的工具.

10.11 质量控制

一种开发高质量软件的方法是, 针对每一个功能, 在假使它们已经开发完成的状态下, 编写一些测试程序, 而且在开发的过程中, 不断地去运行这些测试程序.

`doctest` 模块提供了工具去浏览一个模块并通过嵌入在文档中的测试程序进行有效性测试. 测试的构成简单到只需将这个模块的调用过程和结果进行剪切和粘贴操作, 保存到文档当中. 通过在文档中给用户呈现一个例子, 从而提高了文档的可读性. 同时, 它还确保了代码是忠实于文档的:

```
1 def average(values):
2     """Computes the arithmetic mean of a list of numbers.
3
4     >>> print(average([20, 30, 70]))
5     40.0
6     """
7     return sum(values) / len(values)
8
9 import doctest
10 doctest.testmod() # 自动地通过嵌入的测试程序进行有效性检测
```

`unittest` 模块并没有 `doctest` 这么轻松简单, 但它在一个独立维护的文件中, 提供了更综合的测试集:

```
1 import unittest
2
```

```
3 class TestStatisticalFunctions(unittest.TestCase):
4
5     def test_average(self):
6         self.assertEqual(average([20, 30, 70]), 40.0)
7         self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
8         self.assertRaises(ZeroDivisionError, average, [])
9         self.assertRaises(TypeError, average, 20, 30, 70)
10
11 unittest.main() # 通过命令行调用所有的测试程序
```

10.12 充电区

Python 有一个关于原理的“充电区”(batteries included). 这是你了解 python 原理和它的各种包的强大功能的最佳方式. 例如:

- `xmlrpc.client` 模块和 `xmlrpc.server` 模块使得远距离程序的调用变得简单便捷. 你不用去管任何模块的名字, 也不必掌握 XML 的知识.
- `email` 包是一个处理 email 消息的库, 包括 MIME 和其它以 RFC 2822 为基准的消息文档. 它不像 `poplib` 模块和 `smtpplib` 模块只发送和接收消息, `email` 包有一个完整的工具集去创建或者解码复杂的消息结构(包括附件)和执和互联网编码和包头协议.
- `xml.dom` 包和 `xml.sax` 包为解析这种流行的数据交换格式提供了强大的支持. 同样地, `csv` 模块对读写常规的数据库文件提供了支持. 这些包和模块结合在一起, 大大简化了 Python 应用程序和其它工具的数据交换方法.
- 一些模块如: `mode: 'gettext'`, `locale` 和包 `codecs`, 为 Python 的国际化, 提供了支持.

第十一章 标准库的简明介绍（第二部分）¹⁸

第二部分涵盖了许多关于专业编程需求的高级模块。这些模块很少出现在小的程序当中。

11.1 格式化输出



`reprlib` 模块提供了一个面向内容很多或者深度很广的嵌套容器的自定义版本 `repr()`，它只显示缩略图

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

`pprint` 模块通过一种能够让解释器读懂的方法，来对内置的和用户自定义的一些对象的输出进行更复杂的操作。当返回的结果大于一行时，“pretty printer” 功能模块会加上断行符和适当的缩进，以使数据的结构更加清晰明朗：

```
1 >>> import pprint
2 >>> t = [[['black', 'cyan'], 'white', ['green', 'red']], [['
    magenta',
3     'yellow'], 'blue']]
4 ...
5 >>> pprint.pprint(t, width=30)
6 [[['black', 'cyan',
7     'white',
8     ['green', 'red']],
9  [['magenta', 'yellow'],
10  'blue']]]
```

`textwrap` 模块会根据屏幕的宽度而适当地去调整文本段落：

```
1 >>> import textwrap
2 >>> doc = """The wrap() method is just like fill() except
    that it returns
3 ... a list of strings instead of one big string with newlines
    to separate
```

¹⁸初译：刘鑫 精译：DocsPy3zh 校对：Zoom.Quiet 整理：Liam Huang

```

4 ... the wrapped lines."""
5 ...
6 >>> print(textwrap.fill(doc, width=40))
7 The wrap() method is just like fill()
8 except that it returns a list of strings
9 instead of one big string with newlines
10 to separate the wrapped lines.

```

`locale` 模块访问一个包含因特定语言环境而异的数据格式的数据。 `locale` 模块的格式化函数的分组属性, 可以用组别分离器, 直接地去格式化数字:

```

1 >>> import locale
2 >>> locale.setlocale(locale.LC_ALL, 'English_United_States.
   1252')
3 'English_United_States.1252'
4 >>> conv = locale.localeconv()          #
   得到一个常规框架的映射
5 >>> x = 1234567.8
6 >>> locale.format("%d", x, grouping=True)
7 '1,234,567'
8 >>> locale.format_string("%s%.*f", (conv['currency_symbol'],
   conv['frac_digits'], x), grouping=
9     True)
10 '$1,234,567.80'

```

11.2 模板化

`string` 模块包括一个多元化的 `Template` 类, 为用户提供简化了的语法格式, 使其可以方便的编辑. 这样可以使用户自定义自己的程序而不用去修改程序本身.

这种格式使用由 `$` 和合法的 Python 标识 (包括文字、数字和下划线) 组成的占位符名称. 将这些占位符包含在一对花括号里时允许周围存在更多的字母或者数字而不用理会是否有空格. 必要时使用 `$$` 去表示单独的 `$`

```

1 >>> from string import Template
2 >>> t = Template('${village}folk send $$10 to $cause.')

```

```

3 >>> t.substitute(village='Nottingham', cause='the_ditch_fund'
4 )
'Nottinghamfolk_send_$10_to_the_ditch_fund.'

```

当一个字典或者关键字参数没有给占位符提供相应的值时, `substitute()` 方法会抛出一个 `KeyError` 异常. 对于像 `mail-merge` 风格的应用程序, 用户可能会提供不完整的数据, 此时, `safe_substitute()` 方法可能会更合适——当数据缺失的时候, 它不会改变占位符:

```

1 >>> t = Template('Return_the_$item_to_$owner.')
2 >>> d = dict(item='unladen_swallow')
3 >>> t.substitute(d)
4 Traceback (most recent call last):
5   ...
6 KeyError: 'owner'
7 >>> t.safe_substitute(d)
8 'Return_the_unladen_swallow_to_$owner.'

```

`Template` 类的子类可以指定一个自己的分隔符. 例如, 现在有一大批文件的重命名工作, 针对的是一个照片浏览器, 它可能会选择使用百分号将当前时间、图片的序列号或者文件格式分隔出来作为占位符:

```

1 >>> import time, os.path
2 >>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.
3 jpg']
4 >>> class BatchRename(Template):
5 ...     delimiter = '%'
6 >>> fmt = input('Enter_rename_style_(%d-date%n-seqnum%f-
7 format):_')
8 Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%
9 f
10 >>> t = BatchRename(fmt)
11 >>> date = time.strftime('%d%b%y')
12 >>> for i, filename in enumerate(photofiles):
13 ...     base, ext = os.path.splitext(filename)
14 ...     newname = t.substitute(d=date, n=i, f=ext)
15 ...     print('{0}-->{1}'.format(filename, newname))
16
img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg

```

```
17 | img_1077.jpg --> Ashley_2.jpg
```

另一个用于模板化的应用程序是将项目的逻辑按多种输出格式的细节分离开来。这使得从传统的模板形式转化为 XML 文件、纯文本形式和 html 网页成为了可能。

11.3 Working with Binary Data Record Layouts



`struct` 模块一些函数, 如:`fun:'pack'` 和:`fun:'unpack'` 函数去处理长度可变的二进制记录格式。下面这个例子演示了如何在不使用 `zipfile` 模块的情况下去循环得到一个 ZIP 文件的标题信息。包代码 `H` 和 `I` 分别表示两个和四个字节的无符号数字。而 `<` 则表示它们是标准大小并以字节大小的顺序排列在后面:

```
1 | import struct
2 |
3 | data = open('myfile.zip', 'rb').read()
4 | start = 0
5 | for i in range(3):                # 显示最开始的3个标题
6 |     start += 14
7 |     fields = struct.unpack('<IIIHH', data[start:start+16])
8 |     crc32, comp_size, uncomp_size, filenamesize, extra_size =
9 |         fields
10 |
11 |     start += 16
12 |     filename = data[start:start+filenamesize]
13 |     start += filenamesize
14 |     extra = data[start:start+extra_size]
15 |     print(filename, hex(crc32), comp_size, uncomp_size)
16 |     start += extra_size + comp_size    # 跳到另一个标题
```

11.4 多线程

线程是一种使没有顺序关系的任务并发执行的技术。线程可以用来改进应用程序的响应方式使这些程序可在接受用户输入的同时在后台执行另

一些操作。一个与此相关的例子是运行输入输出程序的同时在另一个线程中执行计算操作。

下面的代码显示了高级模块 `threading` 如何实现主程序执行的同时在后台执行另一个相应的程序：

```
1 import threading, zipfile
2
3 class AsyncZip(threading.Thread):
4     def __init__(self, infile, outfile):
5         threading.Thread.__init__(self)
6         self.infile = infile
7         self.outfile = outfile
8     def run(self):
9         f = zipfile.ZipFile(self.outfile, 'w', zipfile.
10                               ZIP_DEFLATED)
11         f.write(self.infile)
12         f.close()
13         print('Finished background zip of:', self.infile)
14
15 background = AsyncZip('mydata.txt', 'myarchive.zip')
16 background.start()
17 print('The main program continues to run in foreground.')
18 background.join()    # 等待后台任务结束
19 print('Main program waited until background was done.')
```

多线程应用程序的最大挑战就是协调线程之间数据或者其它资源的共享。为此，线程模块提供了许多同步原始函数，包括锁定、条件变量和信号。

虽然有这些强大的工具，但设计上的一个小错误仍然可以导致难以恢复的问题。因此，对于协调各线程我们更倾向于把有的访问集中在一个单独的线程上，这个线程使用 `queue` 模块把其它线程的请求全部集中起来。应用程序使用 `Queue` 的对象来进行跨线程的交流和协调可以使得设计变得更简单，而且更易阅读，更可靠。

11.5 日志

`logging` 模块提供一整套富有特色且灵活的日志系统. 用最简单的方式说, 就是把日志消息传送给一个文件或者“`sys.stderr`”:

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

上述会产生以下的输出:

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

默认的情况下, 信息和调试消息会被捕捉, 并发送给标准错误流. 其它的一些输出选项包括经由邮件、数据报套接字或者发送给一个 HTTP 服务器的路由消息. 新的过滤器可以选择不同的基于消息优先级的路由, 而消息的优先级有: `DEBUG`, `INFO`, `WARNING`, `ERROR`, 和 `CRITICAL`.

日志系统可以被 Python 语言配置, 或者被一个用户的可编辑的配置文件加载, 以此去自定义日志而不用去修程序本身.

11.6 弱引用

Python 语言自动管理内存（大多数对象都有一个对它的引用而 *garbage collection* 对它们进行回收）. 当最后一个引用结束之后内存即刻被回收.

这种机制对大多数应用程序来说都是有效的, 但也有偶然情况, 只有当它们被其它东西引用的时候才需要去跟踪对象. 不幸的是, 仅仅是跟踪它们也需要创建一个引用. `weakref` 模块提供一些工具可以达到同样的效果而不用去创建一个引用. 当不再需要这个对象的时候, 它会自动地从一个

weakref 表中移除然后触发对 weakref 对象的回调。通常应用程序都会对那些创建时花费较多时间的对象提供一个缓存：

```
1 >>> import weakref, gc
2 >>> class A:
3 ...     def __init__(self, value):
4 ...         self.value = value
5 ...     def __repr__(self):
6 ...         return str(self.value)
7 ...
8 >>> a = A(10)                                # 创建一个引用
9 >>> d = weakref.WeakValueDictionary()
10 >>> d['primary'] = a                          # 没有创建一个引用
11 >>> d['primary']                             # 如果存在的话获取这个对句
12 10
13 >>> del a                                    # 移除这个引和
14 >>> gc.collect()                          # 直接调用回收机制
15 0
16 >>> d['primary']                             # 调用的入口已经自动被移除了
17 Traceback (most recent call last):
18   File "<stdin>", line 1, in <module>
19     d['primary']                             # 调用的入口已经自动被移除了
20   File "C:/python31/lib/weakref.py", line 46, in __getitem__
21     o = self.data[key]()
22   KeyError: 'primary'
```

11.7 处理列表的工具

许多数据的结构都需要用到内置的列表类型。但有时候需要在可选择地不同呈现方式中进行权衡。

`array` 模块提供了一个 `array()` 对象，这个对象像一个列表一样存储同一类型的数据，而且更简洁。下面这个例子演示了将一组数字以两个字节的无符号整数（类型码 `H`）形式存储为一个数组而不是通常的 Python 的 `list` 对象的 16 字节的形式：

```
1 >>> from array import array
2 >>> a = array('H', [4000, 10, 700, 22222])
3 >>> sum(a)
4 26932
```

```
5 >>> a[1:3]
6 array('H', [10, 700])
```

`collections` 模块提供了一个 `deque()` 对象, 它可以像一个列表一样在左边进行快速的 `append` 和 `pop` 操作, 但在内部查寻时相对较慢. 这些对象可以方便地成为一个队列和地行广度优先树搜索:

```
1 >>> from collections import deque
2 >>> d = deque(["task1", "task2", "task3"])
3 >>> d.append("task4")
4 >>> print("Handling", d.popleft())
5 Handling task1
6
7 unsearched = deque([starting_node])
8 def breadth_first_search(unsearched):
9     node = unsearched.popleft()
10    for m in gen_moves(node):
11        if is_goal(m):
12            return m
13    unsearched.append(m)
```

此外, 标准库里也提供了一些其它的工具, 如 `bisect` 模块, 它有一些对列表进行排序的函数:

```
1 >>> import bisect
2 >>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
3 >>> bisect.insort(scores, (300, 'ruby'))
4 >>> scores
5 [(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

`heapq` 模块提供了一些函数通过常规列表去实现堆. 最低层的入口通常都在零处. 这对于重复访问一些很小的元素但又不想对整个列表进行排序的应用程序来说十分有效:

```
1 >>> from heapq import heapify, heappop, heappush
2 >>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
3 >>> heapify(data) # 将列表重新整理成堆
4 >>> heappush(data, -5) # 增加一个新入口
5 >>> [heappop(data) for i in range(3)] # 取出这三个最小的入口
```

6 | [-5, 0, 1]

11.8 十进制浮点数的运算

`decimal` 模块提供了一个 `Decimal` 针对十进制浮点小数运算的数据类型. 与内置的数据类型 `float`（针对二进制浮点小数）相比而言, 它对以下几种情况更为有效

- 金融方面的应用程序和其它需要准确显示小数的地方
- 需要精确控制,
- 需要四舍五入以满足法制或者监管要求,
- 需要跟踪有意义的小数部分, 即精度, 或者,
- 一些用户希望结果符合自己的计算要求的应用程序.

例如, 计算七毛钱话费的 5% 的锐收, 用十进制浮点小数和二进制浮点小数, 得到的结果会不同. 如果结果以分的精确度来舍入的话, 这种差异就会变得很重要:

```
1 >>> from decimal import *
2 >>> round(Decimal('0.70') * Decimal('1.05'), 2)
3 Decimal('0.74')
4 >>> round(.70 * 1.05, 2)
5 0.73
```

`Decimal` 的结果会在末尾追加 0, 自动从有两位有效数字的乘数相乘中判断应有四位有效数字. `Decimal` 复制了手工运算的精度, 避免了二进制浮点小数不能准确表示十进数精度而产生的问题.

Exact representation enables the `Decimal` class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

```
1 >>> Decimal('1.00') % Decimal('.10')
2 Decimal('0.00')
3 >>> 1.00 % 0.10
4 0.09999999999999995
5
6 >>> sum([Decimal('0.1')]*10) == Decimal('1.0')
7 True
```

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

精确的显示使得 `Decimal` 可以进行模运算和判断值的等同性, 而这些是二进制浮点数不适合的:

```
1 >>> Decimal('1.00') % Decimal('.10')
2 Decimal('0.00')
3 >>> 1.00 % 0.10
4 0.09999999999999995
5
6 >>> sum([Decimal('0.1')]*10) == Decimal('1.0')
7 True
8 >>> sum([0.1]*10) == 1.0
9 False
```

`decimal` 模块可以实现各种需求的精度运算:

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

第十二章 现在干什么?¹⁹

读完这篇教程, 可能会提高你对使用 Python 的兴趣——你应该能够使用 Python 来解决你现实中的问题. 那么, 你应该去哪里学习更多的东西呢?

This tutorial is part of Python's documentation set. Some other documents in the set are:

这篇文档是 Python 文档的一部分. 其他的部分还有:

- *The Python Standard Library*:

你应该浏览这个手册, 这里给出了完整 (虽然简洁) 的参考资料, 包括类型, 函数, 标准库模块. 在标准 Python 的发布版本中, 包含了大量的额外代码. 比如读取 Unix 的邮箱, 通过 HTTP 获取文档, 产生随机

¹⁹初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

数字, 解析命令行, 编写 CGI 程序, 压缩数据, 以及等等. 浏览标准库参考手册会让你知道什么是可以用的.

- *Installing Python Modules* 解释了如何安装其他 Python 用户编写的额外的模块.
- *The Python Language Reference*: 更详细的介绍了 Python 的语法和语义. 这里可能有大量的需要阅读, 但是作为语言本身来说是非常详细的指导.

更多的 Python 资源:

- <http://www.python.org>: Python 主要的网站. 这里包含了代码, 文档, 和各种指向 Python 相关的网页. 这个网站在世界各地都有镜像, 比如欧洲, 日本, 和澳大利亚; 依赖于你的地理位置, 镜像会比主网站更快.
- <http://docs.python.org>: 快速访问的 Python 参考手册.
- <http://pypi.python.org>: Python 包索引, 先前也被叫做 Cheese Shop, 是用户创建的 Python 模块的索引, 此处就可以提供下载. 如果你开始发布代码, 你可以在这里注册一个帐号, 以便其他用户可以找到它.
- <http://aspn.activestate.com/ASPN/Python/Cookbook/>: Python Cookbook 是一个收集了大量代码的地方, 此处有很多模块, 有用的脚本. 很多收集于此的优秀的代码被收录到一本书中, 名为 Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)
- <http://scipy.org>: 科学计算的 Python 项目. 包括更快的数组运算和操作, 还有像线性代数, 傅立叶变换, 非线性解, 随机数, 统计及等等.

关于更多 Python 相关的问题, 你可以在 `comp.lang.python` 新闻组发贴, 或者发到邮件列表 python-list@python.org 中. 新闻组和邮件列表会自动将这些问题转发, 也就是所有人都可以看到. 一般一天有 120 条信息 (更多时会有几百), 提问 (和解答) 问题, 给出建议, 发布新的模块. 在发布前, 请先确保在 [Frequently Asked Questions](#) (也称为 FAQ), 或在 Python 源代码的 `Misc/` 目录下查看. 邮件列表归档可以在 <http://mail.python.org/pipermail/> 中找到. FAQ 中回答了很多重复会问的

问题, 所以一般也包含了你所需的问题.

第十三章 交互式输入编辑及历史替代²⁰

某些版本的 Python 解释器支持编辑当前输入行和历史替代, 就像 Korn shell 和 GNU Bash shell 一样. 这是通过 GNU Readline 库实现的, 支持 Emacs 模式和 vi 模式. 这个库有它自己的文档, 所以在这里就不再说; 但是基础的会简单的解释. 交互式的编辑及历史记录可能仅适用于 Unix, 及 Cygwin 版本上的解释器.

本章并 * 不 * 会讲 Mark Hammond 的 PythonWin 包或者是伴随 Python 一起发布, 基于 Tk 的 IDLE. 而在 NT 平台下的命令行窗口或者是 DOS 这些 Windows 流派的东西, 也不会描述.

13.1 行编辑

如果可以, 行编辑是一直激活的, 无论是处于第一或第二提示符. 当前行可以使用通用的 Emacs 控制符进行编辑. 最重要的几个: **C-A** (Control-A) 移动光标至行首, **C-E** 移动至行尾, **C-B** 向左移动光标, **C-F** 向右. 退格键删除光标左边的一个字符, **C-D** 则删除右边的字符. **C-K** 删除光标右边剩余的所有字符, **C-Y** 则召回最后一次被删的字符串. **C-underscore** 取消最后一次的改变; 这可以重复的执行.

13.2 历史替代

历史替换运行如下. 所有的非空输入行都会被保存于一个历史记录缓存, 当新的提示符给出时, 你处于这个缓存的最底部. **C-P** 可以向前翻一条记录, **C-N** 则向后翻一条. 任何的历史记录都是可以被编辑的; 如果被修改了, 那么会在提示符前增加一个星号. 按下 **Return** (也就是回车) 将当前行传递给解释器. **C-R** 进行反向搜索; **C-S** 开始前向搜索.

²⁰初译: 刘鑫 精译: DocsPy3zh 校对: Zoom.Quiet 整理: Liam Huang

13.3 按键绑定

按键的绑定和 Readline 库其他的一些参数可以在 `~/.inputrc` 中指明。按键绑定的形式:

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

或:

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

选项可以这样设置:

```
1 set option-name value
```

举个例子:

```
1 # I prefer vi-style editing:
2 set editing-mode vi
3
4 # Edit using a single line:
5 set horizontal-scroll-mode On
6
7 # Rebind some keys:
8 Meta-h: backward-kill-word
9 "\C-u": universal-argument
10 "\C-x\C-r": re-read-init-file
```

注意, 默认情况下 `Tab` 在 Python 中是用于插入一个 `Tab` 字符, 而非 Readline 默认的文件名补全函数。如果你要用, 可以写入下面这行用于改写这个行为:

```
1 Tab: complete
```

在你的 `~/.inputrc` 中。(当然, 这样在缩进时就会有些麻烦.)

自动补全变量与模块名也是可选的。为了开启这个模式, 将下面这段增加到你的启动文件中: ²¹

²¹ 在你打开一个交互式解释器的时候, Python 会执行一个文件的内容, 这个文件由环境变量 `PYTHONSTARTUP` 指定。

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

这就将 Tab 键绑定了补全的功能, 所以按两次 Tab 键就会给你提示; 它会搜寻 Python 中语句的名字, 当前的局部变量, 和可用的模块名. 对于点号的表达式如 `string.a`, 则会执行到最后的点号位置的语句, 然后给出这个对象的属性. 注意, 这会执行程序中定义的代码, 如果一个有 `__getattr__()` 方法的对象是表达式的一部分.

一个更聪明的启动文件可以是这样的. 注意这个会删除它创建的但是不需要再用的名字; 从启动文件在同一命名空间执行后, 它就开始运行, 然后移除这些名字以避免对后面的交互环境产生副作用. 你会发现它对于用于保留某些导入的模块非常有用, 比如 `os`, 这在多数会话中都会被大量使用.

```
1 # Add auto-completion and a stored history file of commands
  # to your Python
2 # interactive interpreter. Requires Python 2.0+, readline.
  # Autocomplete is
3 # bound to the Esc key by default (you can change it - see
  # readline docs).
4 #
5 # Store the file in ~/.pystartup, and set an environment
  # variable to point
6 # to it: "export PYTHONSTARTUP=/home/user/.pystartup" in
  # bash.
7 #
8 # Note that PYTHONSTARTUP does *not* expand "~", so you have
  # to put in the
9 # full path to your home directory.
10
11 import atexit
12 import os
13 import readline
14 import rlcompleter
15
16 historyPath = os.path.expanduser("~/pyhistory")
17
18 def save_history(historyPath=historyPath):
```

```
19     import readline
20     readline.write_history_file(historyPath)
21
22 if os.path.exists(historyPath):
23     readline.read_history_file(historyPath)
24
25 atexit.register(save_history)
26 del os, atexit, readline, rlcompleter, save_history,
    historyPath
```



13.4 交互式解释器的替代品

相对于早期的解析器来说，这是一个很大的进步；但是，有些愿望仍未实现：如果在续行时有合适的缩进那么会很棒（解析器会知道下面的缩进是否需要）。自动补全的机制可能使用解释器的符号表，用于检查或建议匹配括号，引号等的工具也会非常有用。

一个可选的增强型解释器应该算 `IPython` 了，它有 `tab` 补全，对象探索，和更高级的历史管理功能。它也可以被定制，嵌入其他的应用。另外一个则是 `bpython`。

第十四章 浮点算术：问题和限制²²

浮点数在计算机中以二进制的除法表示。比如，十进制的：

0.125

其值为 $1/10 + 2/100 + 5/1000$ ，同样，以二进制表示则为：

0.001

其值为 $0/2 + 0/4 + 1/8$ 。这两种表示法的值是一样的，唯一的区别是，前者以十进制表示，而后者则以二进制表示。

²²初译：刘鑫 精译：DocsPy3zh 校对：Zoom.Quiet 整理：Liam Huang

不幸的是, 大多数的十进制小数都无法严格的以二进制来表示. 一个结果就是, 普遍来说, 你输入的十进制的小数, 通常只是以接近的二进制数表示.

在十进制中这个问题很容易理解. 考虑分数 $1/3$. 你可以用一个接近的十进制数表示:

0.3

要更一些,

0.33

更好一些,

0.333

等等. 但是不管你怎么写, 都不是严格的等于 $1/3$, 但是可以使结果更接近于 $1/3$.

同样, 无论你用了多少位, 二进制的数也无法精确表示十进制的 0.1 . 在以二为底的情况下, 将是个无限循环

0.00011001100110011001100110011001100110011001100110011...

在任意有限的位上停止, 可以得到一个近似值. 在今天的大多数机子上, 浮点数近似地使用二元的分数来表示, 其分子是一个 53 位的数字, 分母则是 2 的幂. 像 $1/10$, 其二进制表示为 $3602879701896397 / 2^{55}$. 这个很接近, 但不是准确的等于.

很多用户因为这些值显示的方法而并不知道近似. Python 仅仅打印一个合适的十进制分数, 而真正的二进制近似还是存储于机器中. 在大多数情况下, 如果让 Python 打印一个十进制小数, 那么其会以真实存储的数字显示, 例如 0.1

```
1 >>> 0.1
2 0.1000000000000000055511151231257827021181583404541015625
```



此处的位数比一般用到的要多，所以 Python 使用一个近似的值代替之：

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

只要记住，尽管打印的值看起来是准确的 $1/10$ ，但是真正存储的只是最接近的二元分数罢了。

有趣的是，有很多不同的值共享同样的分数。举个例子，数字 0.1 和 0.100000000000000001 及 0.1000000000000000055511151231257827021181583404541015625 都是 $3602879701896397 / 2^{55}$ 的近似。因为同样的分数值表示不同值的近似，这样任何的值都可以保证不变式 `eval(repr(x)) == x`。

历史原因，Python 的提示符和内置的 `repr()` 函数会选择 17 位，0.100000000000000001。在 Python 3.1 开始，Python（在绝大多数的系统上），会选择最简的表示 0.1。

注意，二进制表示的浮点数在此处是非常自然的：这个不是 Python 中的 bug，也不是你代码中的 bug。在很多支持硬件浮点型的语言中也可以看到这样的事情（尽管有些语言默认下不会显示不同，或在所有的输出模式下）。

对于更多友好的输出，你可能需要使用字符串格式化来产生一个有限制的数字：

```
1 >>> format(math.pi, '.12g') # give 12 significant digits
2 '3.14159265359'
3
4 >>> format(math.pi, '.2f') # give 2 digits after the point
5 '3.14'
6
7 >>> repr(math.pi)
8 '3.141592653589793'
```

有一点很重要，你需要意识到，在真实情况下，这是个幻觉：你仅仅是四舍五入了显示的真实值。

其中一个幻觉会产生另一个. 举个例子, 因为 0.1 并不是严格的 $1/10$, 三个 0.1 相加并不会生成准确的 0.3:

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

同样, 因为 0.1 不能够得到更接近 $1/10$ 的值, 而 0.3 不能得到更接近 $3/10$ 的值, 因此使用 `round()` 函数来进行四舍五入也是不起作用的:

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

尽管数字不能够更接近它们理想上的准确值, `round()` 函数在计算后使用, 确实可以实现两个数字之间的比较:

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

像这样的浮点数算法, 会导致很多令人奇怪的事情. “0.1” 的问题将在后面更详细的解释, 具体看 “Representation Error” 那节. 更多常见的令人吃惊的事情, 可以参考 [The Perils of Floating Point](#).

就像后面说的, “没有简单的答案.” 但是, 对于浮点数请不要过度谨慎. 在 Python 中这些浮点数的问题来源于其硬件, 在多数的机器中, 浮点的精度没有必要达到 $1/2^{53}$. 对于普通的任务已经足够了, 你需要记住的就是, 这不是算数的问题, 每个浮点数操作都会遇到这样的问题.

当不合理的情况真的存在时, 对于多数情况, 你最终还是能得到希望的结果, 如果你将显示的数值进行四舍五入, 并确保你所需要的位数. `str()` 函数经常就能满足需要了, 使用 `str.format()` 方法, 来指明其 *Format String Syntax* (<http://docs.python.org/library/string.html#formatstrings>).

在需要严格的数值表示时, 试试使用 `decimal` 模块, 这个模块实现了用于账目运算或更高精度时用到的数值算法.

另一种就是 `fractions` 模块, 它实现了基于有理数的算法 (所以 $1/3$ 就可以准确的表述).

如果你有大量浮点数的运算, 那么你可以看看 Python 的数值库或其他的计算和统计的包, SciPy 这个项目对此有很好的支持. 参考 <<http://scipy.org>>.

Python 提供了工具来帮助你获得浮点数的准确值. 你可以使用 `float.as_integer_ratio()` 方法来表示一个分数:

```
1 >>> x = 3.14159
2 >>> x.as_integer_ratio()
3 (3537115888337719, 1125899906842624)
```

因为这个比率是准确的, 它就可以用来比较原始的数字:

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

`float.hex()` 方法以十六进制表述, 这也同样给出了一个被你计算机准确存储的值:

```
1 >>> x.hex()
2 '0x1.921f9f01b866ep+1'
```

前面的十六进制表示, 可以用来重新建立一个浮点值:

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

因为这个表示是严格的, 所以对于不同版本的 Python (跨平台) 都是兼容的, 而且也可以和其他的语言进行交换 (比如 java 和 C99).

另一个有用的工具就是 `math.fsum()` 函数. 它可以在计算总和时减少精度的丢失. 它会记录在求和时丢失的精度. 这样误差就不会积累而最终影响结果了.

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

14.1 表示错误

本节会更详细的解释“0.1”的例子, 并且教你如何进行准确的分析. 此处假设你已有了基本的二元浮点数表示的基础.

Representation error 涉及到这样的事实, 有些 (更准确来书是大多数) 小数的分数表示不能够以二进制为底的分数表述. 这就是主要的原因, Python (或者 Perl, C, C++, Java, Fortran, 还有更多的) 常常不能够如所愿的表示.

为什么会那样? $1/10$ 不能够被二进制的分数准确表示. 基本上全部的机器在今 (2000 年 11 月) 来说都是使用了 IEEE-754 浮点数算法, 并且几乎全部的平台将 Python 的浮点映射为 IEEE-754 “double 精度”. 754 doubles 包含了 53 位的精度, 所以在计算机中, 0.1 被转成一个很接近的分数, 而它又可以这种 $J/2^{**N}$ 的形式表示, 此处的 J 是一个包含 53 位的整数. 重写:

$$1 / 10 \approx J / (2^{**N})$$

为:

$$J \approx 2^{**N} / 10$$

并且记着 J 有严格的 53 位 (也就是 $\geq 2^{**52}$ 但 $< 2^{**53}$), 对于 N 最好的值就是 56:

```
1 >>> 2**52 <= 2**56 // 10 < 2**53
2 True
```

也就是说, 56 是唯一能让 J 为 53 位的 N 值. 而 J 最有可能的值就是那个商:

```
1 >>> q, r = divmod(2**56, 10)
2 >>> r
3 6
```

因为剩余的如果大于 10 的一半, 那么最好的近似就是进一位:


```
1 >>> q+1
2 7205759403792794
```

所以以 754 double 精度表示的 $1/10$ 最合适的值就是：

$7205759403792794 / 2^{56}$

将分子分母约化：

$3602879701896397 / 2^{55}$

注意，因为我们进了一位，所以值会比 $1/10$ 稍微大一点；如果我们没有进位，那么商又会比 $1/10$ 稍微小点。但无论如何，都不是准确的 $1/10$ ！

所以计算机从没有看过 $1/10$ ：它看到的是前面给出的分数，以 754 双精度近似的结果：

```
1 >>> 0.1 * 2 ** 55
2 3602879701896397.0
```

如果我们将这个分数乘以 10^{55} ，我们可以看到 55 位的数字：

```
1 >>> 3602879701896397 * 10 ** 55 // 2 ** 55
2 10000000000000000055511151231257827021181583404541015625
```

这意味存于计算机中的准确值等于 $0.10000000000000000055511151231257827021181583404541015625$ 。很多语言（包括 Python 的旧版本）不是直接显示所有的位数，而是将其保留为 17 位有效数字：

```
1 >>> format(0.1, '.17f')
2 '0.10000000000000001'
```

`fractions` 和 `decimal` 模块使这些计算变得简单：

```
1 >>> from decimal import Decimal
2 >>> from fractions import Fraction
3
4 >>> Fraction.from_float(0.1)
5 Fraction(3602879701896397, 36028797018963968)
```

```
6
7 >>> (0.1).as_integer_ratio()
8 (3602879701896397, 36028797018963968)
9
10 >>> Decimal.from_float(0.1)
11 Decimal('0.
    1000000000000000055511151231257827021181583404541015625')
12
13 >>> format(Decimal.from_float(0.1), '.17')
14 '0.10000000000000001'
```

跋

大约是在 2013 年的 10 月 30 日,我在 <http://docspy3zh.readthedocs.org/en/latest/tutorial/index.html> 看到了刘鑫等人翻译的 Python 入门教程。作为一个 Python 初学者和 L^AT_EX 爱好者,在发现这份在线文档没有高质量的 PDF 输出之后,立即决定用 LaTeX 输出这份文档。

这份文档经过 Sphinx 的自动输出,我逐字句地调整版式,现在终于勉强能见人了。与此同时,我在 Python 一途也算勉强入门——托排版的福,我前前后后仔细琢磨了四遍以上的翻译文稿,并有对照英文原版阅读。

简单的故事到此为止,希望读到此处的读者会对这份 PDF 的质量感到满意,对 Python 和 L^AT_EX 充满热情,也希望我学习 Python 顺顺利利。

Liam Huang

2013 年 11 月 7 日

A 使用 Sphinx 输出包含中文支持的 L^AT_EX 源文件

关于 Sphinx 输出中文 PDF, 还是比较简单的。这归因于这几年中文 T_EX 支持的发展, 特别是刘海洋他们的工作。如果不涉及版式变更, 在用户本地的方法大概可以叙述如下 (假设需要对 index.txt 输出对应的 L^AT_EX 源文件):

1. 安装好 Sphinx (当然还有 Python 作为其基础), 以及 T_EX Live 2013.
2. 在 index.txt 所在目录运行命令行 `sphinx-quickstart`, 按照需求回答问题 (基本都可以按照默认的来, 需要注意的是扩展名以及文件名要根据需求选择, 同时项目名称等会影响最后的输出)。
3. 编辑 `conf.py` 的 L^AT_EX 输出部分 (如果需要; 这部分可以调整输出格式, 但也仅限于很小的一部分格式, 所以不建议 L^AT_EX 新手做, 同时有经验的人也不必要在这里修改, 所以是个鸡肋功能)。
4. 在 index.txt 所在目录运行命令行 `sphinx-build -b latex . _build` (Win 下的命令), 之后就能在 `_build` 下找到 `.tex`, `.sty`, `.cls` 等所需的 L^AT_EX 源文件了。(至此 Sphinx 的部分结束)
5. 打开 `project-name.tex` (project-name 取决于运行 `sphinx-quickstart` 时起的项目名称), 修改第二行为 `\def\sphinxdocclass{ctexrep}`, 第三行为 `\documentclass[UTF8,english]{sphinxmanual}` 保存。
6. 使用 pdfL^AT_EX 或者 XeL^AT_EX 编译即可得到所需的 PDF 文件 (推荐 XeL^AT_EX 编译)。