

第 1 章

Python 核心知识回顾

本章主要内容:

- Python 语言的基本特性
- 如何使用 Python 模块机制
- 如何创建新模块
- 如何创建新包

从 wrox.com 下载本章代码

可以在 wrox.com 网站找到本章涉及的代码。代码可以在 www.wrox.com/go/pythonprojects 的 Download Code 选项卡下找到。第 1 章代码位于 Chapter 1 download, 每个文件都是根据本章提到的代码文件名命名的。

如果已经遗忘了一些基础知识, 本章开始部分提供了 Python 的简短回顾以及一些基础知识。本书的内容都是基于这部分基础知识的。如果对自己的 Python 编程基础有足够的信心, 则可以跳过这些内容直接去阅读感兴趣的部分。毕竟, 之后当需要温习这些内容时, 可以随时回到本章。

在本章, 你首先会了解 Python 的生态系统、数据类型和主要的控制结构, 以及函数和类的定义的知识。然后, 你会看到 Python 模块和包系统。最后会创建一个基础的新模块包。这个包会包含数个模块。

在本章的结尾, 你应该为进一步的内容做好准备, 并且开始将标准 Python 模块应用在真实的项目任务中。

1.1 探索 Python 语言和解释器

Python 是一个动态的但是拥有严格类型的编程语言(Python 是严格类型的编程语言,

解释器会追踪每个变量的类型)。Python 代码既被解释又被编译。Python 源代码首先被编译成字节码，然后被解释器解释。但是这个过程对于用户来说是透明的。你不需要显式地调用 Python 来编译你的代码。

Python 语言有几种实现版本。但最常用的版本是用 C 语言实现的，通常被称为 CPython。其他实现版本包括 Java 语言实现的 Jython，以及专门为微软.NET 平台实现的 IronPython。本书所用的 Python 实现版本是 CPython。



注意：在本书成文时，Python 存在两种版本流：2.x 版本和 3.x 版本。本书将专注于 3 版本。本书所涉及的代码已经在 3.x 版本流的几个版本中测试过，包括最新的 3.4 版本。当涉及与 2.x 版本的较大兼容性问题时，我们通常指的是 2.7 版本。

Python 程序通常被写在带有.py 后缀的文本文件中。Python 解释器被称为 python(小写)。实际上，Python 解释器并不在意文件的后缀，添加这个后缀仅仅是为了方便用户(在有些操作系统中，这样做也允许文件和解释器建立关联)。

也可以直接在解释器中输入 Python 代码。这个方法适用于高度交互的开发风格。在这种开发风格中，想法首先在解释器中形成原型或被测试，然后转移到代码编辑器中。当开始使用一个新概念或代码模块时，Python 解释器是一个强大的学习工具。

在这种模式下工作时，可以在操作系统命令提示符中输入 python 来启动解释器。系统会反馈给你一个包含 Python 版本和一些创建细节的信息。在这个信息的下面是一个交互式提示符。可在这里输入代码。它看起来如下所示：

```
ActivePython 3.3.2.0 (ActiveState Software Inc.) based on
Python 3.3.2 (default, Sep 16 2013, 23:10:06) [MSC v.1600 32 bit (Intel)]
on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

从这个信息中，我们知道解释器适用于 3.3.2.0 版本的 Python。它是一个 ActiveState 的发行版本(而不是 python.org 发行版本)。它是基于 Windows 32 位操作系统创建的。你获得的信息可能会稍有不同，但应该包含相同类型的信息。

如果想执行一个存储在文件中的程序而不是与 Python 解释器交互，则可以在操作系统提示符中使用 python 命令，并在后面附上文件的名称：

```
$ python myscript.py
```




注意：通常，也可以在你的文件管理器工具中双击文件，操作系统会调用 python 来自动运行程序。然而，这通常会导致：程序打开一个窗口，运行结束，然后在你看到结果前关闭窗口。所以你可能会倾向于在一个命令行提示中完整地输入 `python filename` 命令。

Python 带有两个非常有帮助的函数：`dir(name)`和 `help(name)`。它们有助于你学习和探索这门语言。`dir(name)`会告诉你 `name` 所确定的对象有哪些可用的名称。而 `help(name)`将会展示被称为 `name` 的对象的信息。当第一次导入一个新模块时，通常是不知道这个模块包含哪些函数或类。通过查看 `dir()`得到的模块列表，可以知道有哪些方法和成员是可用的。然后可以在列出的任意特性上调用 `help()`。一定要试试这些函数，它们是非常重要的信息来源。

1.2 回顾 Python 数据类型

Python 支持很多强大的数据类型。表面上看，这些数据类型和其他编程语言中对应的数据类型一样。但是在 Python 中，这些数据类型通常拥有强大的能力。任何东西在 Python 中都是对象，也因此拥有方法。这意味着可以对任何变量执行一系列的操作。内置的 `dir()` 和 `help()`函数可以帮助你了解全部信息。在本节，你会看到标准数据类型以及它们最重要的操作。



提示：如果需要，Python 参考手册提供了全部细节(<http://docs.python.org/3.3/reference/>)。

需要注意 Python 的一些基础概念。首先，Python 变量仅是名称。变量名的创建是通过把类型的实例赋值给它们。变量本身并没有类型，而与它们绑定在一起的对象拥有类型。名称只是个标签，同样，它也可以被一个完全不同的对象重新赋值。赋值操作使用 `=` 操作符，所以把一个值赋给一个变量就如下所示：

```
aVariable = aValue
```

这段代码把值 `aValue` 绑定到变量名 `aVariable` 上。如果此变量名不存在，解释器会把这个名称添加到合适的命名空间中。

因此，在 Python 中区别变量和它指向的对象是非常重要的。可以用双等号(`==`)来检查两个变量是否相等，也可以用 `is` 操作符来检查两个变量的对象身份(也就是两个名称是否

指向同一个对象), 如下所示:

```
>>> aString = 'I love spam'
>>> anotherString = 'I love spam'

>>> anInt = 6
>>> intAlias = anInt
>>> aString == anotherString # test equality of value
True
>>> aString is anotherString # test object identity
False
>>> anInt == intAlias        # same value
True
>>> anInt is intAlias        # also same object identity
True
```

Python 根据你使用类型的方式对它们进行分类。比如, 所有类型都可以分类为可变的 (mutable) 或不可变的 (immutable)。如果一个类型是不可变的, 它意味着这种类型的对象一旦创建后就不能再改变。可以创建一个新的数据项并把它赋值给同一个变量, 但是不能更改原来不可变的值。

Python 也支持多个容器类型, 有时也被称作序列(严格来讲, 容器是序列的子集, 稍后会清晰阐述它们之间的差别)。尽管并不是所有的序列都支持所有的操作, 但它们有一组共同的操作。

一些 Python 数据类型是可被调用的。这意味着可以像调用函数一样使用类型名来生成这种类型的一个新实例。如果没有给定初值, 则会返回一个默认值。你将在下面每个数据类型的描述中看到相关的示例。

现在, 你已经了解了操作 Python 数据类型的基础知识, 下面会看到不同的数据类型, 包括数值、布尔、None 类型以及各种容器类型。

1.2.1 数值类型: 整数和浮点数

Python 支持多种数值类型, 包括最基本的整数类型和浮点类型。

Python 整数类型的特点在于它在理论上是无限大的。事实上, 整数的大小只被你的计算机的内存限制。整数类型支持所有常用的数值操作, 比如加法、减法、乘法等。可以使用传统的中缀表示法进行算术运算。比如, 当相加两个整数时:

```
>>> 5 + 4
9
```

或:

```
>>> result = 12 + 8
>>> print (result)
20
```


整数的字面值会默认以十进制表示。可以通过在数值前面加 0 和进制的首字母作为前缀来使用其他进制。因此，二进制被表示为 0bnnn，八进制被表示为 0onnn，十六进制被表示为 0xnnn。

一个整数的类型是 `int`。对于浮点数和数字的字符串表示形式，如 '123'，可以用 `int` 从中创建整数。如下所示：

```
>>> int(5.0)
5
>>> int('123')
123
```

`int` 也可以通过设定第二个可选的参数进行非十进制到十进制的转换(不仅是二进制、八进制和十六进制，最高到三十六进制都可以)。如果想把一个十六进制数的字符串表示转换成一个十进制整数，可以使用：

```
>>> intValue = int('AB34',16)
43828
```

Python 浮点数的类型是 `float`。与 `int` 一样，可以用 `float()` 来转换字符串表示，比如把 '12.34' 转换成浮点数。也可以用它把整数转换成一个浮点数。与整数不同的是，`float()` 不能处理不同进制的字符串。

`float` 类型也支持常用的算术操作以及几种舍入选项。Python 的浮点数是基于美国电气与电子工程师协会(IEEE)标准并且拥有与底层计算机架构相同的取值范围。但与此同时，它们也存在着同样水平的精确度问题，这意味着进行浮点数的比较是非常冒险的。Python 提供了模块来处理固定精度小数(`decimal`)和有理分数(`fractions`)。这可以帮助缓解精度带来的问题。Python 也原生地支持复数(或虚数)数字类型，称为 `complex`。这些类型都有特定的应用场景，所以本书不做详细介绍。

1.2.2 布尔类型

Python 支持一种布尔类型 `bool`。它拥有两种字面值 `True` 和 `False`。`bool` 类型的默认值是 `False`。也就是说，调用 `bool()` 会产生 `False`。

Python 对其他类型也支持 `Truth` 和 `False` 的概念。例如，如果整数值是 0，那么它会被认为是 `False`，而其他所有的非 0 整数值都被认为是 `True`。这点对浮点数一样适用。0.0 被认为是 `False`，而其他所有的浮点数被认为是 `True`。

可以用 `int()` 把布尔数值转换成整数。`False` 会被转换成 0，而 `True` 会被转换成 1。

布尔类型拥有大部分你所期待的布尔代数运算，包括 `and`、`or` 和 `not`。但奇怪的是，Python 并不支持 `xor` 运算。



注意：布尔类型被实现为整数类型的子类。也就是说，布尔类型也支持一些你可能预料不到的操作，比如幂运算。可以输入像 `True**False` 的代码然后得到结果 1。但是你应该只是假装这些特性不存在，并且把它们当作实现的细节。否则，你的代码会变得特别混乱。

除了布尔类型，Python 也支持整数的按位布尔操作。也就是说，Python 会把两个整数的每一个位对当成布尔数值对，并且对每一个位对执行相应的操作。这些操作包括按位与(&)、或(|)、非(^)和异或(~)。还包括把二进制位组左移(<<)或右移(>>)的位移操作符。在本章，你还会看到更多关于按位操作的介绍。

1.2.3 None 类型

None 类型代表一个空对象。在 Python 环境中，只有一个 None 对象。所有对 None 的引用都使用同一个实例。这意味着与 None 的对象值相等测试通常会被对象身份测试代替，如下所示：

```
aVariable is None
```

而不是：

```
aVariable == None
```

None 是 Python 函数的默认返回值。在函数中，它经常作为默认参数的位置标记或标志位。None 是不可调用的，所以不能作为一个转换函数把其他类型转换为 None。在被当作一个布尔数值时，None 的值是 False。

1.2.4 容器类型

如上所述，Python 有几种用来表示不同容器或序列的类型。这些类型有字符串、字节、元组、列表、字典和集。你会在之后看到这些类型之间的相同点和不同点。标准库模块 `collections` 提供了其他一些特殊的容器类型。在接下来的叙述中，你偶尔会接触到它们。



注意：在接下来的讨论中，你会看到接受容器类型作为参数的操作。通常，这就包括 Python 中的可迭代变量。可迭代变量是指符合 Python 迭代协议的对象。简单说，可迭代变量就是可以在循环结构中使用的对象。大多数情况下，你不必担心它们。如果很有兴趣，可以在 Python 文档中详细了解。一个不错的学习起点是：<https://wiki.python.org/moin/Iterator>。

一些特性是所有集合共有的。为了避免在每种容器类型的讨论中都赘述这些特性，在此会介绍它们。

可以通过内置函数 `len()` 得到 Python 中任何集合的长度。这个函数接受一个集合对象作为参数，返回集合中元素的个数。

可以通过索引下标访问一个集合中的单个元素。可以通过在方括号中提供一个下标值(或为字典提供有效的键值)实现这一点。集合的下标从 0 开始，也可以通过使用负的下标实现从后往前索引。这样，集合最后一个元素的下标就是 -1。

虽然使用索引下标只能访问集合中的一个特定的元素，但可以用切片(slicing)来访问集合中的多个元素。切片操作包括一个开始下标、一个结束下标和一个步长。这三个数值通过冒号分隔。切片操作对于字典或集是无效的。步长参数可以帮助你实现比如每隔一个获取元素的操作。这几个参数都是可选的，默认值就是集合的开始下标、集合的最后一个元素下标和为 1 的步长。切片操作返回下标从 `start` 到 `end-1` 的所有(选择的)元素。

下面是在 Python 交互式提示中对字符串进行切片操作的示例：

```
>>> '0123456789'[:]  
'0123456789'  
>>> '0123456789'[3:]  
'3456789'  
>>> '0123456789'[:3]  
'012'  
>>> '0123456789'[3:7]  
'3456'  
>>> '0123456789'[3:7:2]  
'35'  
>>> '0123456789'[:3]  
'0369'
```

可以使用 `sorted()` 函数对大多数集合进行排序。返回的结果是包含原集合元素的已排序列表。`sorted()` 的可选参数使元素的排序和排序顺序变得更加灵活。

通常，在布尔表达式中，空集合被认为是 `False`，反之是 `True`。有两个函数——`any()` 和 `all()`，对集合的真假判断进行了完善。它们有助于对集合的布尔值进行更加精确的判断。`any()` 函数接受一个集合作为参数，如果集合中的任何成员是 `True`，就返回 `True`。`all()` 函数接受一个集合作为参数，当且仅当集合中所有成员都为 `True` 时才返回 `True`。

1.2.5 字符串

Python 字符串本质上是 Unicode 字符的集合(使用 Unicode 的影响会在第 4 章中讨论)。默认的编码方式是 UTF8。如果你的工作语言环境是英语，大部分事情会如你所料的工作。一旦你开始使用非英语字符，事情就开始变得有趣了。目前，你将会在英语语言环境下工作，并一直使用 UTF8。

在 Python 中，字符串的字面值需要被引号封闭。Python 是非常灵活的。就这一点而言，Python 可以使用单引号('Joe')、双引号("Joe")、三重单引号(““Joe””)和三重双引号(“““Joe”””)

来界定一个字符串。很明显，引号的开头和结尾应该是同一种类型，但是任何其他类型的引号都可以被包含在字符串中。这对于撇号或其他的类似情况是尤其有用的(‘He said, “Hi!”’ 或 “My brother’s hat”)。任意类型的三重引号可以跨越多行。下面是一些示例：

```
>>> 'using single quotes'
'using single quotes'
>>> "using double quotes"
'using double quotes'
>>> print('''triple single quotes spanning
... multiple lines ''')
triple single quotes spanning
multiple lines
```

当一个字符串的字面值出现在模块、类或函数的开始并且没有赋值给任何变量时，系统会把它当成文档。当对这个对象调用内置函数 `help()` 时，这个字符串会作为输出的一部分。

当有特殊字符，比如制表符(`\t`)或换行符(`\n`)，在字符串中时，需要加一个反斜杠字符前缀。反斜杠的字面值也必须加这个前缀，这样它们看起来就像双反斜杠。也可以通过在整个字符串前面加一个字母 `r`(代表 `raw`)来避免加反斜杠前缀。这样做就表示特殊字符将不会被处理。不可打印的字符可以用一个反斜杠加上这个字符的十六进制代码在字符串中表示。例如，转义字符表示为 `\x1A`(注意：开头的 `0` 并没有出现在十六进制整数的字面值中)。

字符串是不可变的。这意味着一旦一个字符串形成之后，就不可以直接对它进行修改或添加。然而，可以基于现有的字符串，创建一个新的字符串。这就是许多 Python 字符串操作的工作原理。Python 支持范围广泛的字符串操作。这些操作大部分都实现为字符串类的方法。表 1-1 列出了一些最常用的字符串操作。

表 1-1 字符串操作

操 作	描 述
+	连接字符串。但是该操作在某种程度上是一个低效的操作。通常，可改用 <code>join()</code> 来避免使用它
*	乘法。该操作复制多个字符串并把它们连接起来
<code>upper</code> , <code>lower</code> , <code>capitalize</code>	这些操作改变字符串中字符的大小写
<code>center</code> , <code>ljust</code> , <code>rjust</code>	这些操作会将字符串按照需要在给定的字符宽度内进行对齐。空白部分会根据需要填充指定字符(默认是空格)
<code>startswith</code> , <code>endswith</code>	这些操作会检查子字符串是否匹配一行的开头或结尾。可选参数控制真正被测试的子部分。但这样会让操作名显得有些令人困惑。如果输入为多个子字符串组成的一个元组，它们也可以一次测试多个子字符串
<code>find</code> , <code>index</code> , <code>rfind</code>	这些操作返回找到子字符串的最低下标。如果查找失败， <code>find</code> 会返回 <code>-1</code> ，而 <code>index</code> 会抛出 <code>ValueError</code> 异常。 <code>rfind</code> 从字符串的右边开始查找。所以它会返回满足条件的最高下标

(续表)

操 作	描 述
isalpha, isdigit, isalnum, 等	这些操作检查字符串的内容。几个常用的测试类型分别是字母、数字和字母数字式字符。最常用的操作就是列出来的这几个
join	该操作连接一个字符串列表，并使用给定字符串作为分隔符。通常，一个单空格或无空格会作为分隔符来构建一个字符串。相比字符串连接操作，该操作更快也更节省内存
split, splitlines, partition	这些操作根据一个给定的分隔符将一个字符串分割成一个子字符串列表(默认分隔符为空白，多个连续空格也算一个空白)。需要注意的是，在这个过程中，原字符串中的分隔符将会被删除。splitlines()可以高效地根据换行符分割并返回一个行列表。partition() 根据给定的分隔符分割一个字符串。但该操作只运行到第一次分割，然后会返回分割出来的第一个子字符串、分隔符和剩余的字符串
strip, lstrip, rstrip	这些操作从字符串的两端移除空白(默认)或指定的字符。lstrip ()只处理字符串的左边。rstrip()只处理字符串的右边。这些操作中，没有操作从字符串的中间移除空白。它们只是移除外围的字符。如果需要全局地移除字符，请使用 replace 操作
replace	该操作执行字符串替换。通过指定一个空字符串作为替代者，该操作可以高效地删除字符
format	该操作用来替换在 Python 版本 2 中使用的老版本的 C 语言 printf 风格的字符串格式化操作。在 Python 版本 3 中，printf 风格依然可用，但为了支持 format()，它已经被弃用了。Python 文档对字符串格式化有详细的解释。基本的概念是字符串中嵌入的大括号对形成了作为 format() 参数传入的数据的占位符。大括号内可以包含可选的风格信息，比如填充字符(可以在本书中找到一些相关示例)

其他字符串操作也是可用的，只不过表 1-1 中的操作是最常用的。
在布尔表达式中，空字符串被看作是 False。所有其他的字符串都被看作是 True。

1.2.6 字节和字节数组

Python 支持两种面向字节的数据类型。一个字节是一个八位值，相当于一个范围在 0~255 之间的整数。它代表着计算机存储的或通过网络传输的原始的位模式。它们在使用上和字符串非常相似，并且支持很多相同的方法。这两个类型的名称分别是 byte 和 bytearray。
字节字符串的字面值被表示成被引号引起来，然后在前面加一个字母 b。字节字符串是不可改变的。字节数组与它类似，但它们是可改变的。

在实际中，你很少会用到字节字符串或字节数组，除非你要处理来自文件或网络的二进制数据。有一个问题可能会令你非常惊讶。如果用下标来访问单一的元素，返回值会是一个整数。这意味着把一个单一字符的字节字符串和一个索引下标的字符串值进行比较，产生的结果是 `False`。这个结果与对字符串进行相同操作产生的结果是不同的。下面是一个示例：

```
>>> s = b'Alphabet soup'
>>> c = b'A'
>>> s[0] == c
False
>>> s[0] == c[0]
True
```

可以看到，关键是在比较的两边都使用了下标索引。

可以使用 `struct` 模块将二进制数据从字节表示形式转换成正常的 Python 类型。当然，在完成该操作之前，首先还是要知道字节模式代表的是什么类型。

在布尔表达式中，空字节字符串被看作是 `False`。所有其他的字节字符串都被看作是 `True`。

1.2.7 元组

元组是任意对象的集合。既然这些对象被收集到一起，这就暗示着在它们之间可能存在逻辑关联。但 Python 语言并没有对元组包含的对象有任何的限制。Python 中的元组通常被形容为其他语言中记录或结构体的等价物。

元组的字面值包含一系列用逗号分开的值(或变量)。通常，为避免语义歧义，元组整体都被包括在圆括号中。但元组本身并没有这样的要求。

元组是不可改变的，所以一旦元组被创建之后，就不能修改或扩展它。可以像字符串一样，基于现有元组创建一个新的元组。可以通过调用 `tuple()` 类型函数来创建一个新的空元组。由于元组是不可改变的，因此它可以作为字典的键。

Python 元组有一个非常有用的特性，被称为解包(unpacking)。这个特性能够帮助你将元组中的值提取出来并赋值给单独的变量。最常在这样的场景见到这个特性：你想要将函数返回的元组中的值存储在不同的变量中。下面是一个使用 `divmod()` 函数的示例。`divmod()` 函数将整数除法得到的商和余数组成的元组作为返回值。

```
>>> print(divmod(12,7))
(1, 5)
>>> q,r = divmod(12,7)
>>> print (q)
1
>>> print (r)
5
```

注意，`q` 和 `r` 是如何被看作新的单值变量。

在 collections 模块中有一个 namedtuple 类。它允许通过名称(而不是位置)来索引元素。这一点结合了字典的一些优势和元组的紧凑性(节省内存)和不可变性。

在布尔表达式中，空元组被看作是 False。所有其他元组都被看作是 True。

1.2.8 列表

列表在 Python 中是一种高度灵活的强大的数据结构。它们可以被用来模仿许多经典数据结构的行为，也可以使用自定义对象类的形式作为其他数据结构的基础。它们是动态的，与元组相似的是它们可以保存任何类型的对象，与元组不同的是它们是可以改变的。所以可以修改它们的内容。也可以使用元组风格的解包，把列表的元素赋值给单独的变量。

列表的字面值表示为包含在方括号内的、用逗号分隔开的对象序列。可以通过指定一对空方括号或使用 list() 类型函数的默认值来创建一个空列表。列表有很多方法来添加和移除成员。它们也支持一些算术风格的操作，比如像字符串那样的连接和复制列表。

可以直接用值列表来初始化一个列表，或以编程方式使用列表推导来构建一个列表。列表推导看起来就像是在列表方括号里面的一行 for 循环。下面的示例构建了一个 1~10 的偶数平方的列表：

```
>>> [n*n for n in range(1,11) if not n*n % 2]
[4, 16, 36, 64, 100]
```

表 1-2 列出了一些最常用的列表操作。

在布尔表达式中，空列表被看作是 False。所有其他列表都被看作是 True。

表 1-2 列表操作

操 作	描 述
+	该操作连接两个列表
*	该操作会复制多个第一个列表。注意，所有的复制对象都指向同一个对象。然而，修改一个对象通常会导致奇怪的副作用。通常，列表切片或列表推导是更好的选择
append	该操作添加一个元素到现有列表的尾端。新元素自己也可以是个列表。该操作可以高效地就地执行，返回值是 None
extend	该操作添加一个列表的内容到另一个列表的尾端。它可以高效地连接两个列表。原列表被就地修改。返回值是 None
pop	该操作从列表的尾端或者根据提供的参数在指定位置移除一个元素。返回值是被移除的元素
index	该操作返回元素在列表中的第一个下标。如果在列表中没有发现这个元素，会抛出 ValueError 异常(同名的字符串操作也有类似的行为)
count	该操作返回指定元素在列表中的个数
insert	该操作在指定下标前插入一个元素。如果这个下标过大而超出列表范围，这个元素会被添加到列表的尾端

(续表)

操 作	描 述
remove	该操作移除列表中的第一个指定元素。如果在列表中没有这个元素，会抛出 ValueError 异常
reverse	该操作原地反转列表中的元素
sort	该操作对列表中的元素进行原地排序。可选参数为如何进行排序提供了灵活性。如果想得到一个排好序的列表副本并且不修改原列表，则可以使用 sorted()函数

1.2.9 字典

字典是初学者经常忽略的超级强大的数据结构。它为大量常见的编程问题提供了解决方案。字典使用起来就像列表，但是它的元素是通过键值(而不是数值索引)的方式被访问的。因此，一个 Python 字典就是一个(无序的)键值对序列。

键可以是任意不可改变的值，包括元组。键在字典中必须是唯一的。值可以是任意的 Python 对象，包括另一个字典、一个列表或者其他任何 Python 认为是对象的东西。

字典是高度优化的，所以字典的访问时间是非常快的。实际上，Python 内部的其他部分也广泛地使用字典，包括实现命名空间和类。字典也为这种情况在任何地方提供了一种解决方案：动态命名的值需要被存储起来并支持访问。字典的键不是连续的，这一点也让字典很高效。因为 Python 使用了哈希算法将键映射到一个稀疏数组结构中(如果不理解最后一句话，不必担心，很多人都不理解，但是你真的不需要理解它。对你而言，这句话意味着 Python 字典很快，而且在内存的使用上也很高效)。

字典的字面值包含逗号分隔的键值对。键和它对应的值用冒号分隔，然后整体被包含在一对花括号或 {} 中。它看起来如下所示：

```
>>> {'aKey': 'avalue', 2: 7, 'booleans': {False: 0, True: 1}}
{'aKey': 'avalue', 2: 7, 'booleans': {False: 0, True: 1}}
```

可通过键值而不是数值索引的方式来访问字典中存储的值。如果之前的示例被存放在一个名为 D 的变量中，可按如下方式访问键为 aKey 和 True 的值：

```
>>> D['aKey']
'avalue'
>>> D['booleans'][True]
1
```

可以使用一对空的花括号或 dict() 类型函数的默认值来创建一个空字典。

字典有一些额外的操作可用来抽取键列表和值列表，以及处理默认值。表 1-3 描述了一些操作。

由于字典的实现本质，字典是无序的。确实，当新数据被插入时，顺序可能发生改变。collections 模块有一个 OrderedDict 类。如果需要的话，它可以维持插入的顺序。如果键是

可比较的，`sorted()`函数会返回一个已排序的键列表。如果键不可比较(就像前面示例那样)，`sorted()`方法会抛出一个 `TypeError` 异常。

`collections` 模块也提供了一个 `defaultdict` 类。这个类能帮助你指定一个默认值。在任何情况下使用一个不存在的键时，它都会为这个给定的键用默认值创建一个新元素。这之前描述的 `setdefault` 方法很相似。这可能是个喜忧参半的事情，因为它可能会因为糟糕拼写的键产生虚假的条目。

在布尔表达式中，空字典被看成是 `False`。所有其他字典都被看作是 `True`。

表 1-3 字典操作

操 作	描 述
keys, values, items	这些方法返回类似列表的对象(称为字典视图)，分别包含键、值和键值元组。这些视图是动态的，所以任何针对字典的改动(删除操作等)在它们创建之后还是会反映在视图中
get, pop	这些方法接受一个键和一个可选的默认值作为参数。如果键存在，那么 <code>get</code> 方法从字典中返回键对应的值。如果键不在字典中，则返回这个默认值。 <code>pop</code> 方法以同样的方式工作，但是如果键存在，则会把它从字典中删除。 <code>get</code> 有一个默认值 <code>None</code> 。但是如果没有给出默认值， <code>pop</code> 会抛出一个 <code>KeyError</code> 异常
setdefault	该操作的表现很像 <code>get</code> 。但是，如果键不在字典中，它会用给定的键和默认值在字典中创建一个新的键值对
fromkeys	该操作使用一个序列来提供键和一个给定的默认值(如果没有给定值，则默认值是 <code>None</code>)来初始化一个字典。该操作通常会直接调用 <code>dict.fromkeys()</code> ，而不是在一个现有的字典上调用它

1.2.10 集

集(Set)体现了一个在编程中常用的数学概念：一个集中的元素必须是唯一的。在 Python 中，集和有键但无对应值的字典有很多相似的地方。

集在 Python 的类型是 `set`。对于字典键的基本规则同样也适用于集。也就是说，集里的值必须是不可变的并且是唯一的(的确这也是集的精髓)。`set()`函数的默认返回值是空集。这也是唯一能表达一个空集的方式，因为 `{}` 已经被用来表示一个空字典。`set()`函数接受任意类型的集合(collection)作为参数，并且把它转换为一个集(字典的值会丢失)。

在 Python 中还有一种集类型，称为 `frozenset`。它是不可变的，并且基本上是一个只读集。它的构造函数和 `set()`一样工作，但是它只支持一部分集操作。因为 `frozenset` 是不可变的，所以可以使用 `frozenset` 作为一个普通集的元素。

集的字面值表示为用花括号将逗号分隔的元素括起来：

```
myset = {1, 2, 3, 4, 5}
```

集不支持索引或分片操作。和字典一样，集没有任何固有的顺序。`sorted()`函数返回一个有序的值序列。集有一系列数学风格的集操作。这些操作是其他集合所没有的。表 1-4 列出了 `set` 类型和 `frozenset` 类型都支持的操作。

表 1-4 集操作

操 作	描 述
<code>in</code>	该操作检查单一元素是否在集中。注意：如果被测试的元素本身是一个集， <code>S1</code> ，当且仅当 <code>S1</code> 作为一个集是目标集的元素时，结果才是真。它与 <code>subset()</code> 测试是不同的
<code>issubset</code> , <code><=</code> , <code><</code>	这些测试检查一个集是不是目标集的子集。被测试集的所有元素是否都是目标集的元素。如果两个集是完全相同的，前两个测试的结果会返回 <code>True</code> ，而 <code><</code> 操作符的测试会返回 <code>False</code>
<code>issuperset</code> , <code>>=</code> , <code>></code>	这些测试检查一个集是否是另一集的超集。目标集的所有元素是否都是源集的元素。如果两个集是完全相等的，那么前两个测试的结果会返回 <code>True</code> ，而最后一个操作会返回 <code>False</code>
<code>union</code> , <code> </code>	这些操作返回两个(或更多)集的并集。 <code>union</code> 方法接受一个逗号分隔的集列表作为参数，但是 <code> </code> 操作符的用法是将它插在集之间
<code>intersection</code> , <code>&</code>	这些操作返回两个(或更多)集的交集。这个用法与 <code>union()</code> 相似
<code>difference</code> , <code>-</code>	这些操作会返回那些在源集中但不在目标集中的元素
<code>symmetric_difference</code> , <code>^</code>	返回不在两个集的交集中的所有元素。这个方法只能应用于两个集。但 <code>^</code> 操作符可以通过中缀风格被应用在多个集上

需要注意的是，表 1-4 中的方法衍生形式可以接受任意容器类型作为参数，但是中缀操作只能作用在集上。

表 1-5 列出了只支持集的修饰符操作。尽管 `frozenset` 可以作为其中几个操作的参数，但这些操作不能被用在 `frozenset` 上。注意：这些操作会修改源集本身，它们并不返回一个集，而是返回 Python 的默认值 `None`。这里面的中缀操作只作用于两个集(不像表 1-4 中的操作)，并且只支持真正的集，不支持其他容器类型。可以在多个集上使用这些方法。在需要时，其他容器类型会被转换成集。

在布尔表达式中，空集被看作是 `False`。其他集都被看作是 `True`。

在下一节中，当探索 Python 提供的不同的控制结构时，你会在代码中用到数据类型。

表 1-5 集修饰符操作

操 作	描 述
<code>update</code> , <code> =</code>	这些操作会把目标集(或多个集)的元素添加到源集中
<code>intersection_update</code> , <code>&=</code>	除了在源集和目标集的交集的元素，这些操作会移除其他所有元素。如果涉及多于两个集，该操作的结果是所有涉及的集的交集

(续表)

操 作	描 述
<code>difference_update, -=</code>	这些操作会移除所有在集交集集中的元素。如果涉及多个集，被移除的元素就是在源集与其他任意集的交集集中的元素
<code>symmetric_difference_update, ^=</code>	这些操作返回除了在交集集中的两个集的值。注意，该操作一次只能作用于两个集
<code>add</code>	该操作把给定的元素添加到集中
<code>remove</code>	该操作把指定元素从集中移除。如果没有发现这个元素，它会抛出一个 <code>KeyError</code> 异常
<code>discard</code>	如果给定的元素存在，该操作会把它从集中移除。如果元素不存在，该操作不会抛出 <code>KeyError</code> 异常
<code>pop</code>	该操作会随机地移除一个元素，并返回这个元素。如果集为空，它会抛出一个 <code>KeyError</code> 异常
<code>clear</code>	该操作会清空集中的所有元素

1.3 使用 Python 控制结构

在这一节，你会首先看到一个 Python 程序的整体结构，然后了解到每个基础的控制结构，包括顺序、选择和迭代。最后，你会看到 Python 如何处理错误，回顾上下文管理器，以及学习如何与外部世界交换数据。

1.3.1 结构化你的程序

Python 程序并没有任何必需的预定义入口(比如一个 `main()` 函数)。它们的表现形式就是一个文本文件中的源代码。程序从文件的顶端开始被顺序地读入和执行(定义，比如函数，被执行的方式是函数被创建然后把它赋值给一个名称。但函数内部的代码在函数被调用之前并不执行)。

Python 并没有任何特别的语义来表明一个源文件是一个程序还是一个模块。而且可以看到，一个给定的文件既可以作为程序也可以作为模块使用。一个经典的可执行程序文件包括一系列重要的用于导入任何需要的代码模块的语句，一些函数和类的定义，以及一些可以直接执行的代码。

在实际中，对于一个重要的程序来说，大部分函数和类定义都会放在模块文件中，然后被包含在导入中。这在启动应用程序时添加了一个很短的驱动代码。通常这个代码会放在一个函数中，而这个函数常被命名为 `main()`。但这纯粹只是为了遵循编程惯例，并非是 Python 要求的。

最后，这个 `main` 函数需要被调用。通常在主脚本的末端会放置一个特殊的 `if` 语句来

调用这个 `main` 函数。如下所示：

```
if __name__ == "__main__":  
    main()
```

当 Python 检测到一个程序文件正在被解释器执行而不是作为一个被导入的模块，它会把特殊变量 `__name__` (注意两边都是双下划线) 设置为 `"__main__"`。这意味着任何在 `if` 代码块内的代码只有在这种情况下才会被执行：脚本被作为主程序运行而不是作为被另一个程序导入的文件。如果这个文件本来只是想被用作模块的话，`main()` 函数可能会被 `test()` 函数替换。`test()` 函数会执行一些列的单元测试。再次强调下，实际使用的函数名对 Python 来说是无关紧要的。

1.3.2 使用序列、块和注释

最基础的编程结构就是一个语句序列。通常，Python 语句会单独占据一行，所以一个序列也仅仅是一系列的行。

```
x = 2  
y = 7  
z = 9
```

在这个示例中，所有语句都是赋值操作。其他有效的语句包括函数调用、模块导入或定义。定义包括函数定义和类定义。下面讨论的各种各样的控制结构也是有效的语句。



注意：Python 允许你在一行中包含多个语句，但是这些语句之间要用分号分隔。因此，下面这行代码包含了三个语句：

```
x = 2; y = 7; z = 9
```

Python 社区不推荐使用这种风格，更倾向于一个语句单独一行。

Python 是一个块结构语言，并且程序块是通过缩进级别指示的。缩进的数量是非常灵活的。尽管大多数 Python 程序员坚持使用三个或四个空格来最优化可读性，但 Python 本身不介意这个。不同的集成开发环境(IDE)和文本编辑器对如何使用缩进有它们自己的理解。如果使用多种编程工具，你可能发现你会因为编程工具使用了不同的制表符和空格组合而得到缩进错误报告。如果可能的话，设置你的编辑器来使用空格而不是制表符。

缩进原则在注释上是个例外。一个 Python 注释以一个 `#` 符号开始一直到这行的结束。不管当前的缩进级别，Python 接受在一行中的任何位置开始注释。但是根据惯例，即使是注释，程序员也倾向于同样保持缩进的级别。

1.3.3 选择一个执行路径

Python 支持有限选项的选择。最基础的结构是 if/elif/else 构造。elif 和 else 部分是可选的。它看起来如下所示：

```
if pages < 9:
    print("It's too short")
elif pages > 99:
    print("It's too long")
else: print("Perfect")
```

注意在每个测试表达式末端的冒号。这是 Python 中的指示器，它用来指示一个新的代码块即将来临。它没有开始和结束标记(比如{}), 冒号是唯一的标示。如果它包含的代码块只有一行，那么可以把它放在与冒号相同的行中，否则，它必须作为一个缩进的代码块。即使是只包含一行代码，许多 Python 程序员也更喜欢使用缩进的代码块风格。

同样需要注意的是，可以有任意多个 elif 测试表达式，但是至多只可以有一个 else 子句，没有也可以。

另一个在 Python 中的你能发现的选择结构是条件表达式选择器。它根据给定的测试条件从多个值中产生一个。它看起来如下所示：

```
<a value> if <an expression> else <another value>
```

一个示例是一个屏幕坐标一直被加 1 直到某一极限(可能是屏幕的最大分辨率)，然后被重置为 0。这可以被写为：

```
coord = coord + increment if coord < limit else 0
```

与之相对等的更传统的写法是：

```
if coord < limit
    coord += increment
else:
    coord = 0
```

你应该非常小心地使用条件表达式选择器，因为它非常容易产生晦涩的代码。如果不确定的话，则应该使用扩展的 if/else 形式。

最终，还有一点值得注意的是 Python 的比较表达式。在许多编程语言中，如果想要测试一个值是否在两个限定值之间，则需要两个单独的测试，如下所示：

```
if aValue < upperLimit and aValue > lowerLimit:
    # do something here
```

Python 会非常乐意看到这样的代码，但是它也提供了一个有用的捷径。可以把两个比较表达式按如下方式联合起来：

```
if lowerLimit < aValue < upperLimit:
    # do something here
```

1.3.4 迭代

Python 提供了几种迭代的方法。最基本也最常见的就是 `while` 循环。`while` 循环如下所示：

```
while BooleanExpression:
    aBlockOfCode
else:
    anotherBlock
```

注意在 `while` 语句的末端有一个冒号(:)。这表示在它之后跟着代码块。也要注意代码块的缩进。原则上，只要 `BooleanExpression` 的值还是 `True`，代码块就会被执行。然而，有两种方法可以忽略 `BooleanExpression` 的值，直接退出 `while` 循环。它们是 `break` 语句和 `return` 语句。`break` 语句会立刻退出循环。如果循环是在一个函数定义中，`return` 语句也可以起到这个作用。`return` 语句会立刻退出函数，所以也会退出函数内的任意循环。

`else` 子句是可选的，而且在实际中也很少被用到。它会在 `BooleanExpression` 为 `False` 时执行，包括循环正常退出的情况。如果循环是被 `break` 或 `return` 语句退出的，则 `else` 子句就不会被执行。

`while` 循环的一个惯用语法是把 `True` 作为测试条件来创建一个无限的循环，然后在循环体内部放置一个 `break` 测试。在下面的示例中，循环会读取用户命令并处理它们。如果命令包含字母 `q`，循环就会退出。

```
while True:
    command = input('Enter a command[rwq]: ')
    if 'q' in command.lower(): break
    if command.lower() == 'r':
        # process 'r'
    elif command.lower() == 'w':
        # process 'w'
    else:
        print('Invalid command, try again')
```

`break` 有一个名为 `continue` 的同伴语句。`break` 会退出语句块和循环，然而 `continue` 只会退出当前循环迭代的语句块。控制权又会回到 `while` 语句。如果 `while` 的测试条件为 `True`，一个新的代码块迭代就会开始。

Python 中的下一个重要的循环构造是 `for` 循环。它看起来如下所示：

```
for item in <iterable>:
    code block
else:
    another code block
```

`for` 循环将各项从可迭代对象中取出，并且对每一项都执行代码块。可以像之前在 `while` 循环中描述的那样使用 `break` 或者 `return` 来终止循环。可以像之前那样使用 `continue` 来终止循环中的单次迭代。

当所有迭代都执行完毕后，`else` 代码块就会被执行。如果使用 `break` 或者 `return` 退出循环，它就不会被执行。

可迭代对象是任何遵守 Python 迭代器协议的对象。实际上，它经常是一个容器，比如一个列表、元组，或者是一个可以返回一些值的函数，比如 `range()`。`open` 函数返回一个文件迭代器。它有助于你在不需要先把文件读入内存的情况下循环遍历一个文件。也可以定义自己的自定义迭代类。

`for` 循环中有一个经常用到的函数是 `enumerate()`。这个函数返回包含迭代对象和一个序列数的元组。在默认情况下，序列数等同于列表的下标。这意味着 `for` 代码块可以更加轻松地直接更新可迭代对象。`enumerate()` 接受第二个可选参数。这个参数指定了序列数的开始数字。例如，可以使用这个参数来使文件中的行数从 1 而不是默认的 0 开始。

下面这个示例通过打印一个文件和关联的行数来说明这些知识点：

```
for number, line in enumerate(open('myfile.txt')):
    print(number, '\t', line)
```

最后，Python 有两个内嵌循环结构。你已经在本章早些时候列表的讨论中看到了它们中的一个：列表推导。

列表推导是一种更一般的循环形式的特定应用。这种循环形式被称为生成器表达式。在有些地方，如果不使用它，可能就需要使用一个字面值的序列。如果回想一下本章早些时候列表推导的示例，你使用生成器表达式把 1~10 的偶数的平方填充进了列表，如下所示：

```
>>> [n*n for n in range(1,11) if not n*n % 2]
[4, 16, 36, 64, 100]
```

在方括号内的部分就是一个生成器表达式。它的一般形式如下：

```
<result expression> for <loop variable> in <iterable> if <filter expression>
```

通过比较这个一般式和列表推导的示例，你会发现在示例中的结果表达式就是 `n*n`，循环变量是 `n`，可迭代对象是 `range(1,11)`。筛选表达式是 `if not n*n % 2`。

也可以像这样把它重写成常见的 `for` 循环：

```
result = []
for n in range(1,11):
    if n*n % 2:
        result.append(n)
```

关于生成器表达式，有一点特别重要的是它们并不是一次就生成所有数据。更确切地说，它们根据需要来生成(由此得名)数据项。这样当处理大型数据集时，它可以极大地帮助节省内存资源。在本章后面，你会在名为 `generator function` 的一种特殊的函数类型中学到更多关于生成器表达式的知识。

1.3.5 异常处理

有两种方法可以处理错误。第一种要在每个动作被执行时显式地检查它，另一种则尝试执行操作并且依赖系统在发生错误时产生一个错误条件或异常(exception)。尽管第一种方法在一些情况下非常适用，但在 Python 中，第二种方法则更为常用。Python 通过 try/except/else/finally 构造来支持这种技术。它的一般表达式看起来如下所示：

```
try:
    A block of application code
except <an error type> as <anExceptionObject>:
    A block of error handling code
else:
    Another block of application code
finally:
    A block of clean-up code
```

except、else 和 finally 都是可选的。但是如果使用 try 语句，则 except 和 finally 必须至少存在一个。结构中可以有多个 except 语句，但是只可以有一个 else 或 finally 语句。如果不需要异常详情，则可以省略 except 语句行的 as...部分。

try 语句块被执行，如果发生了错误，就会测试异常类。如果存在与错误类型匹配的异常语句，就会执行相应的语句块(如果有多个异常语句块都指定同一个异常类型，则只有第一个匹配语句会被执行)。如果不存在匹配的 except 语句，异常会被向上传播一直到达顶层的解释器。Python 会产生其常规的错误追溯报告。注意，一个空的 except 语句可以捕捉任何错误类型。然而，这通常是一个糟糕的主意，因为它可能隐藏任何发生在非预期中的错误。

如果 try 代码块在没有任何错误的情况下执行成功，else 代码块就会执行。在实际中，else 很少被用到。不管是否有错误被捕捉或向上传播，也不管 else 语句是否被执行，finally 语句总是会被执行。这在锁定状态下提供了一个释放任何计算资源的机会。即使是在使用 break 或 return 语句退出 try/except 子句的情况下，finally 语句依然会被执行。

可以使用单一的 except 语句来处理多种异常类型。如果想要这样做，就要把异常类放在一个元组内(需要使用圆括号)。一个任意的异常对象包含异常发生位置的具体信息，同时提供了一个字符串转换方法。这样就可以通过打印这个对象来提供一个有意义的错误消息。

可以在你自己的代码中抛出异常，也可以使用任意现有的异常类型或通过创建一个 Exception 类的子类来定义你自己的异常类型。还可以给你抛出的异常传递参数，并且在 except 子句中使用错误对象的 args 特性来访问这些存在于异常对象中的参数。

下面这个示例抛出了带有一个自定义参数的 ValueError 异常，然后捕获这个错误并将给定的参数打印出来。

```
>>> try:
...     raise ValueError('wrong value')
... except ValueError as error:
...     print (error.args)
```



```
...
('wrong value',)
```

注意，你没有得到一个完整的回溯，只是打印了 `except` 代码块中的输出。也可以只是简单调用无参数的 `raise` 来处理它，然后重新抛出原异常。

1.3.6 上下文管理

Python 有一个运行时上下文(context)的概念。它通常包括一个临时性的资源。这个资源就是你的程序想要交互的一些东西。一个常见的示例可能是一个打开的文件或一个并发执行的线程。为了处理这个，Python 使用了关键字 `with` 和一个上下文管理器(context manager)协议。这个协议帮助你定义你自己的上下文管理器类，但是你在大部分情况下还是会使用 Python 提供的管理器。

你通过调用 `with` 语句来使用一个上下文管理器：

```
with open(filename, mode) as contextName:
    process file here
```

上下文管理器保证文件在使用后会被关闭。这对于上下文管理器来说是相当常用的功能。它可以保证宝贵的资源在使用后会被释放，或者对首次使用的资源采取适当的共享预防措施。上下文管理器通常可以避免使用 `try/finally` 结构。`contextlib` 模块为构建你自己的上下文管理器提供了支持。

现在你已经看到了 Python 可以处理的不同数据类型，以及在处理过程中可以使用的控制结构。现在是时候去探索如何在你的 Python 程序中读入和输出数据。这就是下一节的主题。

1.4 在 Python 中读取和输出数据

基本的数据输入和输出对于任何编程语言来说都是必要的。需要考虑你的程序会如何与用户和存储在文件中的数据进行交互。

1.4.1 与用户交互

如果想要通过 `stdout` 发送数据给用户，则可以使用已经多次见到的 `print()` 函数。在这部分，你会学到如何更加精确地控制输出。如果想要从用户读取数据，则可以使用 `input()` 函数。它会提示用户输入然后从 `stdin` 中返回原始字符的字符串。

相比第一次出现，`print()` 函数由于有几个可选参数要变得更加复杂些。最简单的用法是：你只是把一个字符串传给它，然后 `print()` 函数会在 `stdout` 上显示字符串，后面跟一个行结束符(eol)。稍微复杂点的用法是：可以一次性地传多个项给 `print()`，然后它会转换并且依次展示这些项。这些项之间用空格分隔。

上段文字在 `print()` 函数的行为中确定了三个固定元素：

- 它在 stdout 上展示输出。
- 它用一个 eol 符号结束。
- 它用空格分隔多个项。

实际上，以上这些没有一个是固定的，`print()` 允许使用可选参数来修改任意或全部元素。可以通过指定一个 `file` 参数来修改输出；分隔符是通过 `sep` 参数来定义的，而结束字符是通过 `end` 参数定义的。下面这行打印了臭名昭著的拥有两个字符串的 "hello world" 信息，用连字符(-)分隔，以 "END" 作为结束标记，并把它输出到文件中：

```
with open("tmp.txt", "w") as tmp:
    print("Hello", "World", end="END", sep="-", file=tmp)
```

这样，文件的内容应该是："Hello-WorldEND"。

字符串 `format()` 方法在与 `print()` 结合在一起时才会真正地体现它的作用。这个组合可以整洁清晰地分开呈现你的数据。此外，在打印长列表的对象和字符串片段时，使用 `format()` 通常会更加高效。Python 文档中有许多如何使用 `format()` 的示例。

也可以使用 `input()` 函数与用户进行交流。这个函数会读取用户对给定屏幕上的提示的响应值。需要自己负责把返回的字符转换成任何显示的数据类型，并且要处理转换中出现的任何错误。



注意：在 Python 版本 2 中，使用的是 `raw_input()` 函数，而不是 `input()`。在版本 2 中，`input()` 函数有非常不同的表现。它会对用户输入的任何东西都进行求值。这会产生一个安全问题，因为用户可能输入恶意代码。版本 3 移除了版本 2 的 `input()` 函数，然后把 `raw_input()` 重命名为 `input()`。

下面这个示例要求用户输入一个数字。如果数字太高或者太低，它会打印一个警告(如果愿意的话，它可以形成一个猜测游戏的核心)。

```
target = 66

while True :
    value = input("Enter an integer between 1 and 100")
    try:
        value = int(value)
        break
    except ValueError:
        print("I said enter an integer!")

if value > target:
    print (value, "is too high")
elif int(value) < target:
    print("too low")
```



```
else:
    print("Pefect")
```

程序提示用户输入一个适当范围内的整数,然后用 `int()`把读取到的值转换成一个整数。如果转换失败,则会抛出一个 `ValueError` 异常,然后展示错误信息。如果转换成功,就肯定能得到一个有效的整数。这样,可以跳出 `while` 循环,继续把它和目标值进行测试。

1.4.2 使用文本文件

在编程中保存数据时,文本文件是非常重要的。Python 提供了几个函数用来处理文本文件。



注意: Python 中的文件接口实际上是一个更高层次抽象接口的具体化。抽象接口从一个名为 `io.IOBase` 的类开始。你基本上可以忽略它们。它们只是创建了标准的操作集,并把它们应用于文本文件和类文件对象。

在前面你看到了 `open()`函数,它接受文件名和一个模式作为参数。模式可以为 `r`、`w`、`rw` 和 `a` 中的任意一个。它们分别代表着读、写、读写和附加(有一些其他模式,但是不经常用到。还有一些可选参数用来控制数据如何被解释。可以在文档中获得细节)。`r` 模式需要文件是存在的; `w` 和 `rw` 模式会创建一个新的空文件(或者覆盖任何已存在的同名文件)。`a` 模式会打开一个已经存在的文件。如果指定名称的文件并不存在,它会创建一个新的空文件。返回的文件对象也是一个上下文管理器。就像你在上下文管理器部分看到的那样,它可以被用在一个 `with` 语句块中。如果没有使用一个 `with` 语句块,那么当用完这个文件时,你应该调用 `close()`方法显式地关闭文件。这样可以保证任何驻留在内存缓冲区中的数据都会被发送到磁盘上的物理文件中。`with` 结构体会自动地调用 `close()`,这也是使用上下文管理器方法的优势之一。

一旦已经打开了文件对象,就可以根据需要使用 `read()`、`readlines()`或 `readline()`这些方法。`read()`方法读取整个文件内容到一个字符串中,并以一个换行符结束。`readlines()`方法把文件一行一行地读入一个列表中,并且保留每一行的换行符。`readline()`方法读取文件中的下一行,它也保留换行符。文件对象是可迭代的,所以你不需要任何读取方法就可以直接在 `for` 循环中使用它。因此,从文件中读取行的推荐模式如下:

```
with open(filename, mode) as fileobject:
    for line in fileobject:
        # process line
```

可以使用 `write()`或 `writelines()`方法向一个可写入的文件对象中写入数据。它们与类似名称的读方法是等同的。注意,没有写单独一行的 `writeline()`方法。

如果在使用 `rw` 模式,你可能想要移动到文件的特定位置来重写现有的数据。使用 `tell()`

方法可以找到你在文件中的当前位置。使用 `seek()` 方法可以定位到一个指定位置(可能是你之前使用 `tell()` 记录下来一个位置)。`seek()` 有几种计算位置的模式, 默认模式是计算距离文件开头的偏移值。

现在你已经掌握了编写 Python 程序的所有的基本技术。然而, 为了实现本书的重点, 即能够处理更大的项目, 你会想要扩展 Python 的功能。下一节我们开始探索它。

1.5 扩展 Python

最简单的扩展 Python 的方法就是编写你自己的函数。你定义的函数可以与使用它们的代码放在相同的文件中。或者也可以创建一个新的模块, 然后从它导入函数。你会在下一节看到模块。现在, 你将会创建函数并且在同一个文件中使用它们。实际上在这一节, 你大多数时间会使用交互式提示符来试验示例。

Python 创建新功能中的下一步就是定义你自己的类并且从它们创建对象。同样, 普遍的做法是在模块中创建类。你会在下一节看到怎么实现它。这部分涉及的示例都很简单, 你仅仅使用 Python 提示符就可以了。

Python 程序员经常在他们的程序中使用文档字符串。文档字符串就是字符串面值。它们没有被赋值给变量, 而且遵循在它们被定义地方的缩进级别。可以用文档字符串来描述函数、类或者模块。`help()` 函数会读取并展示这些文档字符串。

1.5.1 定义并使用函数

Python 中有多种类型的函数。这部分会首先介绍标准函数, 然后是生成器函数, 最后是有些神秘的 `lambda` 函数。

在 Python 中, 你会使用 `def` 关键字来定义函数。它看起来如下所示:

```
def functionName(parameter1, param2,...):  
    function block
```

Python 函数永远会返回一个值。可以用 `return` 关键字来指定一个显式的返回值。否则, Python 会默认返回 `None`(如果在输出中发现意想不到的 `None` 值, 请检查相关函数体中是否含有显式的 `return` 语句)。可以通过在参数名后面加上等号和指定值来给参数赋予默认值。在下面, 你会在 `odds()` 生成器函数中看到一个示例。

通过下面的“试一试”中, 可以最容易理解一个函数定义是怎么创建和使用的。

试一试: 创建并使用一个函数

在这个“试一试”中, 你会创建一个函数。这个函数接受多个输入参数并返回一个值。此函数会根据给定的斜率、 x 坐标和常量, 使用直线的数学方程, 返回对应的 y 坐标。然后, 你会使用这个函数去生成一条直线上的一系列坐标。

(1) 启动 Python 解释器。

(2) 输入下面的代码来定义函数：

```
>>> def straight_line(gradient, x, constant):
...     ''' returns y coordinate of a straight line
...         -> gradient * x + constant'''
...     return gradient*x + constant
...
>>>
```

(3) 既然你已经定义了函数，使用一些简单的值来测试它。可以提前心算出这些值。试着使用斜率值为 2、x 值为 4、常数为-3 的参数来调用函数：

```
>>> # test with a single value first
>>> straight_line(2,4, -3)
5
```

(4) 现在让我们用一种更加复杂的方法来测试函数。使用下面的代码：

```
>>> for x in range(10):
...     print(x, straight_line(2,x,-3))
...
0 -3
1 -1
2 1
3 3
4 5
5 7
6 9
7 11
8 13
9 15
```

(5) 最后，检查 help() 函数能否正确地识别函数：

```
>>> help(straight_line)
Help on function straight_line in module __main__:

straight_line(gradient, x, constant)
    returns y coordinate of a straight line
    -> gradient * x + constant
(END)
```

示例说明

第二步的第一行创建了函数定义。函数被命名为 `straight_line`，并且有三个必要的函数：`gradient`、`x` 和 `constant`。这些参数对应着在数学方程式 $y=mx+c$ 中使用的值。在这个方程式中， m 是斜率， c 是常量。

第二行是一个文档字符串。它描述了这个函数的功能以及应该如何使用它。

第三行是函数的代码块。它可能任意的复杂并且长达多行。但是在这个示例中，它只

有一行。因为要返回结果，所以在前面加了关键字 `return`。注意，代码行首的缩进级别应与文档字符串的行首相同。否则，你会得到一个缩进错误。

然后，我们使用一些简单值来测试函数。通过心算，我们确定返回值 5 确实等于 $(2*4-3)$ 。看起来，函数至少在一些简单的情况下能够正常工作。

我们在 `for` 循环中使用函数来产生一组 `x, y` 坐标对。在这里，`gradient` 的值被固定为 2，`constant` 值被固定为 -3，而把 `x` 作为循环变量。如果身边有便利的纸的话，可以尝试在纸上画出函数产生的坐标，并确认这些坐标能否形成一条直线。

最后，我们使用 `help()` 函数来确定文档字符串被正确地检测到并展示出来。

1. 生成器函数

你将看到的下一个函数形式是生成器函数。生成器函数与标准函数看起来几乎一样，除了标准函数使用 `return` 返回数据，而生成器函数使用关键字 `yield` (理论上生成器函数除了使用 `yield` 之外也可以使用 `return`，但是只有 `yield` 表达式能产生生成器行为)。

Python 优雅的魔力使生成器函数很特殊。它们像定格相机一样工作。当一个标准函数遇到 `return` 语句时，它会返回值，然后函数就会丢弃它的所有内部数据。当下次函数被调用时，一切都从头做起。

`yield` 语句会做不同的事情。它像 `return` 一样返回一个值，但是它不会使函数丢弃数据；相反，所有数据都被保存起来了。下次函数被调用时，即使 `yield` 语句在代码块的中间或者处于循环中，程序也会从 `yield` 语句开始执行。在一个函数中，甚至可以有多个 `yield` 语句。由于 `yield` 语句可以被放在一个循环中，这就可以创建一个高效的返回一个无限系列结果的函数。下面的示例返回一个递增的奇数序列：

```
def odds(start=1):
    ''' return all odd numbers from start upwards'''
    if int(start) % 2 == 0: start = int(start) + 1
    while True:
        yield start
        start += 2
```

在这个函数中，首先检查 `start` 参数是否是一个奇数(偶数除以 2 的余数是 0)。如果不是奇数，你会强迫它加 1 而成为下一个最近的奇数。然后，你创建了一个无限的 `while` 循环。通常，这是一个糟糕的主意，因为你的程序会陷入死循环。然而，由于这是一个生成器函数，你在使用 `yield` 语句返回 `start` 的值。这样，函数在返回 `start` 的值时就会退出。当下一次函数被调用时，程序会从之前离开的地方再开始。所以 `start` 会被加 2，然后继续循环，产生下一个奇数并且退出函数。函数会在下一次调用时继续。

Python 确保生成器函数能够变成迭代器，这样你就可以在 `for` 循环中使用它们。如下所示：

```
for n in odds():
    if n > 7: break
```



```
else: print(n)
```

你把 `odds()` 当成一个集合在使用。每次循环访问它，它会调用生成器函数，然后接收下一个奇数值。

通过插入 `break` 测试，可以避免一个无限循环。这样，当大于 7 时，`odds()` 就不会被调用了。



注意：如果在同一个程序中第二次使用 `odds()`，它会创建一个全新的迭代器实例，并且序列会从头开始。

现在你已经理解生成器函数是如何工作了，你也可能已经认识到本章前面介绍的生成器表达式就是高效的匿名生成器函数。生成器表达式实际上是一种变相的没有名称的生成器函数。

这为我们将要学到的最后一个函数类型，`lambda` 函数，提供了一个完美的桥梁。

2. `lambda` 函数

术语 `lambda` 来自于阿隆佐·邱奇(Alonzo Church)发明的一种微积分。好消息是，你不需要知道任何使用 `lambda` 函数的数学知识。`lambda` 函数背后的原理是它通常是一个很小的匿名函数块。可以把它插入到代码中，然后像一个普通函数一样调用 `lambda` 函数。`lambda` 函数并不常用。但是当需要创建很多只会被使用一次的小函数时，它们是非常方便的。它们通常被用在 GUI 或网络编程的环境中。在这些情况下，编程工具包需要一个用来回调得到结果的函数。

`lambda` 函数的定义如下所示：

```
lambda <param1, param2, ..., paramN> : <expression>
```

这是一个 `lambda` 字面值加上一个可选的逗号分隔的参数名列表、一个冒号和一个任意的 Python 表达式。这个表达式通常会使用输入参数。注意，表达式前面并没有关键字 `return`。

一些语言允许 `lambda` 函数可以任意的复杂。但是 Python 只允许使用一个表达式。这个表达式也可以非常复杂。但是在实际中，这更适合创建一个标准的函数。这样代码具有更好的可读性，在出现问题时，也更便于调试。

可以把 `lambda` 函数赋值给一个变量。这种情况下，那些变量看起来就像是标准的 Python 函数名。例如，以下是一个用 `lambda` 函数重新实现的 `straight_line` 函数的示例：

```
>>> straight_line = lambda m,x,c: m*x+c
>>> straight_line(2,4, -3)
5
```

在本书之后的部分中，你会经常看到 `lambda` 函数的身影。切记：它们就是用来简明地表达那种简短的单行表达式的函数。

1.5.2 定义并使用类和对象

Python 使用一种传统的、基于类的方法支持面向对象编程。Python 类支持多继承和操作符重载(但不支持方法重载)，以及常用的封装机制和消息传递。尽管一些命名习惯可以为特性提供一层微弱的保护并建议客户端不应该在何时直接使用特性，但是 Python 类不直接实现数据隐藏。Python 支持类方法和类数据，以及属性(properties)和插槽(slots)的概念。类拥有构造函数(__new__())和初始化函数(__init__())。尽管并不保证被调用，Python 也有析构函数机制(__del__())。对于定义在类内部的方法和数据，类也充当它们的命名空间。

对象是类的实例。尽管不常用，但是实例在创建之后可以添加自己的属性。

类的定义使用 class 关键字，后面跟着类名和用括号括起来的超类的列表。类定义包含一些类数据和方法定义。一个类方法定义把指向调用实例的引用作为第一个参数，通常被称为 self。一个简单的类定义看起来如下所示：

```
class MyClass(object):
    instance_count = 0
    def __init__(self, value):
        self.__value = value
        MyClass.instance_count += 1
        print("instance No {} created".format(MyClass.instance_count))
    def aMethod(self, aValue):
        self.__value *= aValue
    def __str__(self):
        return "A MyClass instance with value: " + str(self.__value)
    def __del__(self):
        MyClass.instance_count -= 1
```

类名通常以一个大写字母开头。在 Python 3 中，除非特别声明，超类总是 object。所以前面示例使用 object 作为超类实际上是多余的。由于没有出现在任何类方法中，因此 instance_count 数据项是一个类特性。__init__() 函数是一个初始化器(在 Python 中，除非是从一个内置类继承，构造函数很少被用到)。它设置了实例变量 self.__value 的值，添加之前定义的类变量 instance_count，然后打印一条信息。value 前的双下划线表明它实际上是一个私有数据，不应该被直接使用。在对象被构建之后，__init__() 方法立刻被 Python 自动调用。实例方法 aMethod() 修改在 __init__() 方法中创建的实例特性。__str__() 方法是一个用来返回一个格式化字符串的特殊方法。例如，传递到打印函数的对象会以一种有意义的方法被打印出来。当对象被销毁时，析构函数 __del__() 减少类变量 instance_count。

可以如下所示创建一个类的实例：

```
myInstance = MyClass(42)
```

该操作在内存中创建了一个实例，然后把新实例作为 self，42 作为 value 来调用 MyClass.__init__()。

可以如下所示使用点符号调用 aMethod() 方法：

```
myInstance.aMethod(66)
```


这可以被转换成更加显式的调用：

```
MyClass.aMethod(myInstance, 66)
```

并且产生想要的行为。这样，`__value` 特性的值被调整了。

如果打印实例，则可以看到`__str__()`方法起的作用：

```
print(myInstance)
```

该操作会打印下面这条信息：

```
A MyClass instance with value: 2772
```

也可以在创建或销毁一个实例之前和之后打印 `instance_count` 的值：

```
print(MyClass.instance_count)
inst = MyClass(44)
print(MyClass.instance_count)
del(inst)
print(MyClass.instance_count)
```

这会显示计数被增加然后又被减少(在垃圾回收过程中，在析构器调用之前可能会有一点小小的延迟，这应该只是一小会儿)。

`__init__()`、`__del__()`和`__str__()`方法不是仅有的特殊方法。它们中的一些都使用双下划线来标识(它们有时也被称为双下划线(dunder)方法)。操作符重载是通过一组这些特殊的方法来支持的，包括`__add__()`、`__sub__()`、`__mul__()`和`__div__()`等。其他方法为实现 Python 协议，比如迭代或上下文管理，做了准备。可以在自己的类中重写这些方法。你永远都不要定义自己的以双下划线开始的函数，否则 Python 未来的改进和增强可能会破坏你的代码。

可以在子类中重写方法，并且新的定义可以通过使用 `super()` 函数触发被继承版本的方法。如下所示：

```
class SubClass(Parent):
    def __init__(self, aValue):
        super().__init__(aValue)
```

对 `super().__init__()` 的调用会转换成对父类的 `__init__()` 方法的调用。使用 `super()` 可以避免问题，尤其是在多继承的情况下。在多继承的情况下，一个类可能被继承多次，但是通常你不想它被多次初始化。



注意：相比于 Python 2，在 Python 3 中使用 `super()` 被大大简化了。Python 2 中的 `super()` 看起来就像 `super(SubClass, self).__init__(aValue)`，但这种方式用起来非常不直观。

插槽是一种节省内存的设备。可以使用 `__slots__` 这个特殊特性并提供一个对象特性名

列表来激活它们。通常，`__slots__` 是一个不成熟的优化。只有当有一个明确的、已知的需求时，才应该使用它。

属性是另一个数据特性中可用的特性。即使没有使用常用的方法语法，它们也会强制通过一组方法访问特性。这样就允许你让这个特性只读(或者甚至是只写)。可以通过一个示例透彻地理解它。这个示例创建一个 `Circle` 类，包括 `radius` 特性和 `area()` 方法。由于你希望 `radius` 的值永远为正，因此你不希望用户可以直接改变它的值，因为它们可能会传一个负值。即使 `area()` 被实现为一个方法，你也希望它看起来像一个只读的数据特性。可以通过把它作为 `radius` 和 `area` 属性来同时达到这两个目的。

试一试：在类中创建属性(testCircle.py)

在这个“试一试”中，首先会创建一个简单的 `Circle1` 类。它只有一个特性和两个可调用的方法：`setRadius()`和 `area()`。然后，会创建第二个类 `Circle2`。它把 `radius` 和 `area` 变成属性。最后，你会看到如何使用属性来简化使用客户端代码中的类。

(1) 启动你最喜欢的编程编辑器或 IDE，创建一个名为 `testCircle.py` 的新文件(或者加载从本书网站上下下载的文件)。

(2) 输入下面的代码：

```
class Circle1:
    def __init__(self, radius):
        self.__radius = radius
    def setRadius(self, newValue):
        if newValue >= 0:
            self.__radius = newValue
        else: raise ValueError("Value must be positive")
    def area(self):
        return 3.14159 * (self.__radius ** 2)

class Circle2:
    def __init__(self, radius):
        self.__radius = radius

    def __setRadius(self, newValue):
        if newValue >= 0:
            self.__radius = newValue
        else: raise ValueError("Value must be positive")
    radius = property(None, __setRadius)

    @property
    def area(self):
        return 3.14159 * (self.__radius ** 2)
```

(3) 保存代码。

(4) 启动 Python 解释器，输入下面代码来使用 `Circle1`：

```
>>> import testCircle as tc
```



```
>>> c1 = tc.Circle1(42)
>>> c1.area()
5541.76476
>>> print(c1.__radius)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: 'Circle1' object has no attribute '__radius'
>>> c1.setRadius(66)
>>> c1.area()
13684.766039999999
>>> c1.setRadius(-4)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "D:\PythonCode\Chapter1\testCircle.py", line 7, in setRadius
    else: raise ValueError("Value must be positive")
ValueError: Value must be positive
```

(5) 用下面的代码来试试 Circle2:

```
>>> c2 = Circle2(42)
>>> c2.area
5541.76476
>>> print(c2.radius)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: unreadable attribute
>>> c2.radius = 12
>>> c2.area
452.38896
>>> c2.radius = -4
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "D:\PythonCode\Chapter1\testCircle.py", line 18, in __setRadius
    else: raise ValueError("Value must be positive")
ValueError: Value must be positive
>>>
```

示例说明

在文件 `testCircle.py` 中创建了两个类。第一个类 `Circle1` 实现了你想要强制用户使用 `setRadius()` 方法修改半径值的目的。通过在特性 `self.__radius` 前面加上双下划线前缀实现了这一点。这是 Python 所用的私有化方法。然后你创建了 `setRadius()` 方法，它会在使用给定值之前检验它的有效性。如果值是负的，它会抛出一个错误。你也提供了一个 `area()` 方法，这样用户可以使用通常的方法调用技术来计算面积。

第二个类 `Circle2` 使用相当不同的方式处理这些需求。它使用 Python 中的属性定义特性创建了一个只写特性 `radius`。它也创建了一个作为只读特性的 `area` 方法。就像在解释器中练习使用类看到的那样，用户在使用 `Circle2` 时的代码更加直观。这关键在于你调用的 `property()` 类型函数，如下所示：

```
radius = property(None, __setRadius)
```

这行代码接受的参数是一组函数，它们用来执行读、写和删除操作(以及文档字符串)。每个函数的默认值都是 `None`。这里创建了 `radius` 属性，设置读函数为 `None`，但是写函数为 `__setRadius()` 方法(现在是私有的)。其他参数值都被默认地设置为 `None`。这样做的结果是：用户可以把它当成一个公共的数据特性访问，但是当给 `radius` 赋值时，就必须调用 `__setRadius()` 方法来保护这个值。任何试图读取(或删除)这个特性的操作都会被忽略，因为这些操作实际上操作的都是 `None`。

`area` 属性稍有差别，它使用了一个 Python 属性修饰符(`@property`)。属性修饰符是一种创建只读属性的快捷方式。这是属性的一种常用方式。

下面介绍交互式会话，我们创建了一个 `Circle1` 实例，并且使用 `area()` 方法打印出面积。然后试图通过访问 `__radius` 直接打印半径，但是 Python 假称它没有这个特性(由于双下划线的私有设置)并且抛出一个 `AttributeError` 异常。但是当使用 `setRadius()` 方法时，一切就很顺利。再次打印面积时，修改已经生效了。最后，我们试图把一个负值赋给半径，正如我们所料，`setRadius()` 方法抛出了一个 `ValueError` 异常。这个异常还带有一个自定义的错误消息：“Value must be positive”。

在使用 `Circle2` 的会话中，我们可以看到代码变得更加简洁。我们仅使用 `area` 属性名就可以获得面积，并且也可以给 `radius` 属性名赋值。当试图将一个负值赋给它时，赋值方法抛出了一个 `ValueError` 异常。尽管错误消息会稍有不同，但直接打印半径又产生了一个 `AttributeError` 异常。

属性需要程序员这边做少量的额外工作，但却可以极大地简化类的使用。

在这一节中，我们已经看到了如何使用函数和类来扩展 Python 的功能。在下一节中，你将看到如何把这些扩展封装在可复用的模块和包中。

1.6 创建和使用模块和包

在大多数编程环境中，模块对于正式的编程任务来说是非常重要的。它们允许程序被拆分为易于管理的块并且提供项目间代码复用的机制。在 Python 中，模块只是简单的以 `.py` 结尾的源文件，它们位于任何 Python 可以找到的地方。在实际中，这意味着文件必须位于当前的工作目录或在 `sys.path` 变量列出的文件夹中。通过在系统的环境变量 `PYTHONPATH` 中指定你自己的文件夹或在运行时动态地添加，可以把它们添加到这个路径中。

尽管模块提供了一个很有用的包装方法来封装小数量的源文件用于复用，但是它们并不能完全满足更大的项目，例如，GUI 框架或数学函数库。对于这些，Python 提供了包(package)的概念。一个包本质上是包含许多模块的文件夹。唯一的要求是，文件夹需要包含一个名为 `__init__.py` 的文件，这个文件可以为空。对于用户来说，包看起来特别类似于模块，包中的子模块看起来就像模块的特性。

1.6.1 使用和创建模块

我们使用 `import` 关键字来访问模块。在 Python 中，它有很多变形，最常见的形式如下：

```
import aModule
import aModule as anAlias
import firstModule, secondModule, thirdModule...
from aModule import anObject
from aModule import anObject as anAlias
from aModule import firstObject, secondObject, thirdObject...
from aModule import *
```

最后一种形式把 `aModule` 模块中所有可见的名称都导入到当前的命名空间中(很快你会学到如何控制可见性)。该操作可能存在命名冲突的严重风险，导入的名称可能会与内置名称或者你已经或将要在本地定义的名称相冲突。因此，最后一种导入形式只推荐在 Python 命令提示符中测试模块时使用。相比于命名冲突带来的混乱来说，使用其他导入形式带来的小量额外输入的代价其实是很小。其他 `from...import` 形式要更加安全，因为你只是导入指定的名称，而且在必要时还可以使用别名来重命名它们。这样可以大大降低与本地名称发生命名冲突的概率。

一旦使用前三种形式中的任意一种来导入模块，就可以通过在需要的名称前加上模块名(或别名)和点符号来访问它的内容。在前面，你已经学习了有关这方面的示例。例如，`sys.path` 就是 `sys` 模块的一个特性。

既然已经介绍了模块实际上仅仅是源文件，那么你就应该了解一下在创建模块时的一些注意事项。也许除了初始化那些可能依赖本地环境的变量外，你应该避免使用顶层代码，它们会在模块被导入时运行。这意味着你想要复用的代码应该被包装成函数或者类。通常，也会提供测试函数来练习使用模块中的所有函数和类。模块的名称习惯上也都是小写字母。



注意：有一个称为 PEP8 的 Python 风格指南，它提供了一些命名习惯和代码布局规则方面的指导。这些指导并不是强制的，但是强烈推荐你使用它，尤其在你提交的代码被包含在标准库时。PEP8 可以在下面网站中找到：<http://legacy.python.org/dev/peps/pep-0008/>。

有两种方法可以控制模块中对象的可见性。第一种方法类似于类中的私有机制，可以在一个名称前面加一个下划线前缀。当使用 `from x import *` 风格的语句时，这些名称并不会被导出。另一种控制可见性的方法就是在一个名为 `__all__` 的顶层变量中列出你想要导出的名称。这种方法可以保证只有那些你特别想要被导出的名称才会被导出。如果可见性对你来说很重要的话，我们推荐你使用这种方法，而不是下划线方法。



注意：模块中一个重要的陷阱是 `sys.path` 列表是按序搜索的。这通常意味着，你创建的任何模块都会在内置或标准库模块之前被找到。所以你不要使用一个标准模块名作为你自己模块文件的名称，这一点很重要。否则，会发生奇怪的事情，并且即使你意识到那是你的模块，其他读者读取它时也很容易被捉弄。

在下一个“试一试”中，你会练习大部分这些内容，但是首先需要了解一下包以及它们和模块之间的区别。

1.6.2 使用和创建包

在本节的开始，你发现一个 Python 包其实仅仅是包含一个名为 `__init__.py` 文件的文件夹。所有在这个文件夹中的其他 Python 文件都是这个包的模块。Python 把包仅看成另一个类型的模块。这意味着 Python 包可以包含其他包到任意深度。只要每个子包都有自己的 `__init__.py` 文件，它就是个有效的包。



注意：刚才说包被定义为拥有一个 `__init__.py` 文件，但严格说来，这不是真的。包的真正定义特征是它有一个 `__path__` 特性。然而，在实际中，你不需要提供它，因为 Python 会为你完成这件事。所以，如果创建一个 `__init__.py` 文件，一切就都就绪了。

`__init__.py` 文件本身并不是非常特殊，它只是另一个 Python 文件。当导入包时，它就会被加载。这意味着这个文件可以为空，这种情况下，导入包的操作仅仅是让你能够访问包里的模块。`__init__.py` 文件也可以像任何其他模块一样包含 Python 代码。特别是，它可以像之前描述的那样定义一个 `__all__` 列表来控制可见性，这可以有效地让包拥有私有的实现文件。当一个客户导入包时，这些文件不会被导出。

在创建包时，一个常见的需求是需要有一组函数或数据，能够在所有被包含的模块之间分享。可以把被分享的代码放入一个名为 `common.py` 的模块文件中并把它放在包的顶层，然后让所有其他模块都 `import common`，这样就能满足这个需求。它会作为包的一部分而对它们可见，但如果不是显式地被包含在 `__all__` 列表中，包的外部消费者不会看到它。

当使用包时，可以像对待任何其他模块一样对待它。你使用所有常用的风格来导入变量。但是命名模式被扩展了，需要使用点符号来指定在包的层级中哪个子模块是需要的。当使用点符号导入子模块时，实际上定义了两个新的名称：包名和模块名。例如，考虑一下这条语句：


```
import os.path
```

这行代码导入了 `os` 包的 `path` 子模块。但是它同时也让 `os` 模块作为一个整体可见。你不需要再为 `os` 本身使用单独的导入语句就可以继续访问 `os` 的函数。可以推断：Python 自动执行了导入层级中的所有 `__init__.py` 文件。所以在 `os.path` 的示例中，它先执行了 `os.__init__.py` 文件，然后执行了 `path.__init__.py` 文件。

另一方面，如果按如下方式在导入之后使用别名：

```
import os.path as pth
```

则只有 `os.path` 模块是被暴露的。如果想使用 `os` 模块函数，就需要一个显式的导入。尽管只有 `path` 作为名称 `pth` 是被暴露的，但 `os` 和 `path` 的 `__init__.py` 文件还是都被执行了。

Python 标准库包含很多个包，其中就包含了刚才提到的 `os` 包。其他值得注意的包包括 UI 框架 `tkinter` 和 `curses`、`email` 包，以及关注于网络的 `urllib`、`http` 和 `html` 包。你将会在本书的后面部分使用它们中的一些。

命名空间包：

Python 3.3 引入了一种新类型的包，称为命名空间包(namespace package)。一个命名空间包包含数个部分。每一部分都指向一个对象。这个对象有或者没有一个物理的表示形式，并且可能位于网络上或者在本地文件系统的不同部分中。命名空间包并不使用 `__init__.py` 文件技术，而是依赖于 `sys.path` 的定义。它会在导入时寻找模块。

在撰写本书时命名空间包的版本还很新，它们有什么其他的扩展用法还未可知。就目前来看，你可能在实际中不会遇到它们。而这意味着对于用户而言，它们看起来与传统的包区别不会很大。

现在你已经了解了所有关于模块和包的原理。在下一节中，我们会创建一些模块和一个包。这会帮助你熟悉这些原理。

1.7 创建示例包

既然你已经了解了相关原理知识，是时候把它们付诸实践。在本节中，将创建两个模块，并且把它们绑定在同一个包中。你会使用在布尔类型部分提到的按位逻辑操作符。包的目的是为那些操作符提供一个基础的接口，并且扩展它们的应用范围，以包括测试单个的位值。在实现这一点时，你也会看到几个之前介绍过的 Python 的核心语言特性。你开发的模块并不是用来优化性能，而是被设计来说明概念。然而，把它们完善成真正有用的工具并非难事。

试一试：创建模块(bits.py)

在这个“试一试”中，首先会创建一个简单的、传统的基于整数输入模块。然后，会创建另一个模块。它定义了一个可以用来表示一块二进制数据的类。这个类提供了一些

按位函数作为类方法。最后，创建了一个包来包含这两个模块。

(1) 新建一个名为 `bitwise` 的文件夹。它最终会成为你的包。

(2) 在这个文件夹中，创建一个名为 `bits.py` 的 Python 脚本文件。这个文件包含下面的代码(或者加载本书的下载文件中名为 `bits.py` 的文件):

```
#!/bin/env python3
''' Functional wrapper around the bitwise operators.
Designed to make their use more intuitive to users not
familiar with the underlying C operators.
Extends the functionality with bitmask read/set operations.

The inputs are integer values and
return types are 16 bit integers or boolean.
bit indexes are zero based

Functions implemented are:
    NOT(int)          -> int
    AND(int, int)     -> int
    OR(int,int)       -> int
    XOR(int, int)     -> int
    shiftleft(int, num) -> int
    shiftright(int, num) -> int
    bit(int,index)    -> bool
    setbit(int, index) -> int
    zerobit(int,index) -> int
    listbits(int,num) -> [int,int...,int]
'''

def NOT(value):
    return ~value

def AND(val1,val2):
    return val1 & val2

def OR(val1, val2):
    return val1 | val2

def XOR(val1,val2):
    return val1^val2

def shiftleft(val, num):
    return val << num

def shiftright(val, num):
    return val >> num

def bit(val,idx):
```



```

    mask = 1 << idx # all 0 except idx
    return bool(val & 1)

def setbit(val, idx):
    mask = 1 << idx # all 0 except idx
    return val | mask

def zerobit(val, idx):
    mask = ~(1 << idx) # all 1 except idx
    return val & mask

def listbits(val):
    num = len(bin(val)) - 2
    result = []
    for n in range(num):
        result.append( 1 if bit(val,n) else 0 )
    return list( reversed(result) )

```

(3) 保存文件，就在你的 bitwise 文件夹中，启动 Python 解释器。

(4) 输入下面的代码，测试你的新模块：

```

>>> import bits
>>> bits.NOT(0b0101)
-6
>>> bin(bits.NOT(0b0101))
'-0b110'
>>> bin(bits.NOT(0b0101) & 0xF)
'0b1010'
>>> bin(bits.AND(0b0101, 0b0011) & 0xF)
'0b1'
>>> bin(bits.AND(0b0101, 0b0100) & 0xF)
'0b100'
>>> bin(bits.OR(0b0101, 0b0100) & 0xF)
'0b101'
>>> bin(bits.OR(0b0101, 0b0011) & 0xF)
'0b111'
>>> bin(bits.XOR(0b0101, 0b11) & 0xF)
'0b110'
>>> bin(bits.XOR(0b0101, 0b0101) & 0xF)
'0b0'
>>> bin(bits.shiftright(0b10,1))
'0b100'
>>> bin(bits.shiftright(0b10,4))
'0b100000'
>>> bin(bits.shiftright(0b1000,2))
'0b10'
>>> bin(bits.shiftright(0b1000,6))
'0b0'
>>> bits.bit(0b0101,0)
True

```

```
>>> bits.bit(0b0101,1)
False
>>> bin(bits.setbit(0b1000,1))
'0b1010'
>>> bin(bits.zerobit(0b1000,1))
'0b1000'
>>> bits.listbits(0b10111)
[1, 0, 1, 1, 1]
```

示例说明

这个模块是一个非常简单的函数列表。这些函数包装了内置的按位操作符，包括非(~)、与(&)、或(|)、异或(^)、左移(<<)和右移(>>)。这些操作作用于二进制数据，也就是简单的由 1 和 0 组成的、并且在计算机中作为一个存储单元被存储起来的一个序列。基本上，所有数据都是以二进制形式被存储在计算机中的。

基于这些包装函数，又补充了一组函数。它们的功能分别是测试某个位的值是否是 1，设置一位的值为 1，以及设置一位的值为 0(或者称为重置位值)。位数是从右边开始查的，起始于 0。测试是通过使用一个位模式(或者称为位掩码(bitmask))完成的。在除了 zerobit() 之后的所有情况中，这个位模式就是除了你想要测试或设置的位之外，其他位都是 0 的序列。可以通过将 1 左移一定的数位来创建掩码。zerobit() 对普通的掩码进行按位补码，这样可以创建一个除了你想要重设的位为 0，其他位都是 1 的序列。

最后，使用一个函数列出给定值的每一位。最后一个函数稍微有一些复杂，它体现了一些 Python 的编程特色。首先，通过 bin() 把它转换成一个二进制字符串，并把这个字符串的长度减 2(为了去掉开头的 0b 字符)来计算出数字的长度。然后，创建了一个空的结果列表，并且在得到的位组上进行循环。对于每一位，你会根据该位是否被设置来向列表中添加 1 或 0，这使用了 Python 的条件表达式结构。

对模块的测试引入了一些有趣的话题。最开始，导入了创建的新模块。由于当前位置与文件在同一个文件夹中，所以不需要修改 sys.path 的值 Python 就可以查找到它。在测试的开始，测试了 NOT() 函数(当然，前面加了模块名 bits 作为前缀)，然后马上就能看到一個反常现象：Python 解释器打印了十进制的表达式作为结果。为了解决这个问题，可以使用 bin() 函数将数字转换成一个二进制的字符串表达式。然而，还有个问题，因为数字是负的。Python 在内部会使用最左边的位来表示符号。通过反转所有位，我们也可以反转它的符号。可以使用一个掩码 0xF(或者十进制 15，如果喜欢的话)来避免这种混淆。这个掩码帮助只恢复最右的四位。可以使用 bin() 来进行这个转换，你现在可以看到预期中的反转位模式。很明显，如果反转的值大于 16，你可能需要使用更长的掩码。只需要记住每个十六进制数都是 4 位。所以需要做的就是掩码后面额外加一个 F。

下一组测试非常简单，测试的函数范围从 AND() 到 shiftright()。可以在视觉上检查输入位模式和输出来检查结果。shiftright() 的示例展示了一个有趣的输出，它把位向右移了太多，以至于产生了一个 0 的结果。换言之，Python 使用 0 去填充移位操作留下的空隙。

下面介绍下一个新功能，你使用了 bit()、setbit() 和 zerobit() 来测试和修改给定值的单

独位。同样，可从视觉上检查输入和输出的模式来确定它们是否产生了正确的结果。记住，索引参数从右边开始记数并且起始于 0。

最后，测试了 `listbits()` 函数。再一次，可以轻松地对比二进制输入模式和数的结果列表。

所以，现在你已经有了一个可以被导入的工作模块。可以像 Python 中任何其他模块一样使用它。可以通过提供一个测试函数来进一步增强模块。并且如果愿意，还可以把这个测试函数包装在一个 `if __name__` 子句中。但是现在，你将继续了解如何从一个单独的模块开发成一个包。

试一试：创建包(bitmask.py)

在这个“试一试”中，构建了一个类。它复制了所有在 `bits.py` 中的函数作为一组方法。然后将两个模块都绑定到一个包中。

(1) 进入 `bitwise` 文件夹。

(2) 创建一个名为 `bitmask.py` 的新文件。这个文件中包含下面的代码(或者加载本书的下载文件中名为 `bitmask.py` 的文件)：

```
#!/bin/env python3
''' Class that represents a bit mask.
It has methods representing all
the bitwise operations plus some
additional features. The methods
return a new BitMask object or
a boolean result. See the bits
module for more on the operations
provided.
'''

class BitMask(int):
    def AND(self, bm):
        return BitMask(self & bm)
    def OR(self, bm):
        return BitMask(self | bm)
    def XOR(self, bm):
        return BitMask(self ^ bm)
    def NOT(self):
        return BitMask(~self)
    def shiftleft(self, num):
        return BitMask(self << num)
    def shiftright(self, num):
        return BitMask(self > num)
    def bit(self, num):
        mask = 1 << num
        return bool(self & mask)
    def setbit(self, num):
        mask = 1 << num
        return BitMask(self | mask)
```

```

def zerobit(self, num):
    mask = ~(1 << num)
    return BitMask(self & mask)
def listbits(self, start=0, end=-1):
    end = end if end < 0 else end+2
    return [int(c) for c in bin(self)[start+2:end]]

```

(3) 现在保存它，可以在 Python 解释器中测试它。

(4) 留在 bitwise 文件夹中，启动 Python 并输入下面的代码：

```

>>> import bitmask
>>> bm1 = bitmask.BitMask()
>>> bm1
0
>>> bin(bm1.NOT() & 0xf)
'0b1111'
>>> bm2 = bitmask.BitMask(0b10101100)
>>> bin(bm2 & 0xFF)
'0b10101100'
>>> bin(bm2 & 0xF)
'0b1100'
>>> bm1.AND(bm2)
0
>>> bin(bm1.OR(bm2))
'0b10101100'
>>> bm1 = bm1.OR(0b110)
>>> bin(bm1)
'0b110'
>>> bin(bm2)
'0b10101100'
>>> bin(bm1.XOR(bm2))
'0b10101010'
>>> bm3 = bm1.shiftleft(3)
>>> bin(bm3)
'0b110000'
>>> bm1 == bm3.shiftright(3)
True
>>> bm4 = bitmask.BitMask(0b11110000)
>>> bm4.listbits()
[1, 1, 1, 1, 0, 0, 0]
>>> bm4.listbits(2,5)
[1, 1, 0]
>>> bm4.listbits(2, -2)
[1, 1, 0, 0]

```

(5) 退出解释器。

既然你已经证明新模块可以正常工作，那就可以继续把 bitwise 目录转换成一个 Python 包。

(6) 新建一个空的 `__init__.py` 文件。

(7) 为了测试包是否可以正常工作，需要将你的工作目录改成 bitwise 的上一级目录。

现在就做这件事。

现在需要测试你能否导入包和它的内容，并访问它的功能。

(8) 启动 Python 解释器并输入下面的测试代码：

```
>>> import bitwise.bits as bits
>>> from bitwise import bitmask
>>> bits
<module 'bitwise.bits' from 'bitwise/bits.py'>
>>> bitmask
<module 'bitwise.bitmask' from 'bitwise/bitmask.py'>
>>> bin(bits.AND(0b1010,0b1100))
'0b1000'
>>> bin(bits.OR(0b1010,0b1100))
'0b1110'
>>> bin(bits.NOT(0b1010))
'-0b1011'
>>> bin(bits.NOT(0b1010) & 0xFF)
'0b11110101'
>>> bin(bits.NOT(0b1010) & 0xF)
'0b101'
>>> bm = bitmask.BitMask(0b1100)
>>> bin(bm)
'0b1100'
>>> bin(bm.AND(0b1110))
'0b1100'
>>> bin(bm.OR(0b1110))
'0b1110'
>>> bm.listbits()
[1, 1, 0]
```

示例说明

创建了一个基于内置整数类型 `int` 的类。由于仅仅为类提供新方法而不存储任何额外的数据特性，因此不需要提供一个 `__new__()` 构造函数或 `__init__()` 初始化函数。这些方法与 `bits.py` 中写的函数非常类似，但是这里你创建了一个 `BitMask` 实例作为返回类型。`listbits()` 方法也展示了另一种获得位列表的方法。它首先使用 `bin()` 把值转化为字符串表达式。然后进行列表推导而创建一个列表，它利用了 `int()` 进行字符转整型的转换。同时，`listbits()` 方法也被扩展并接受一对 `start` 和 `end` 参数。这对参数的默认值就覆盖了这个二进制数的全部长度。但是它不能被用来获得位的子集。这里面有一小部分的工作是根据索引的正负来调整 `end` 的值。负的下标值不需要加两个字符，因为它们会自动地从右边索引。这样就使用了一个 Python 条件赋值语句来保证 `end` 值是正确的。

既然已经创建了类，那么就可以把它从本地目录中导入，并把它作为一个标准模块进行测试。随后，你重复了一组与对 `bits.py` 进行的测试相类似的测试。值得注意的一点是，可以混合和匹配传统按位操作符和新的函数版本。正如你在 `shiftright()` 示例中看到的那样，也可以就像任何其他整型那样去比较 `BitMask` 对象。最后，你证实新的 `listbits()` 算法可以

正常工作，并且新增的额外参数也可以在正值和负值的情况下都如预期的那样工作。

在这个阶段，你已经在文件夹中创建了两个标准模块。然后，创建了一个空白的 `__init__.py` 文件，把这个文件夹转变成为一个 Python 包。为测试它能否正常工作，回到上一级目录。这样，这个包对于 Python 解释器就是可见的了。然后，你确认可以导入包、模块，并访问一些功能。恭喜，现在你已经拥有了一个包含两个模块的包。

了解如何创建和使用标准的以及那些你自己创建的模块和包是一个了不起的开端。然而，网上还有更多可用的模块和包在等待下载。下一节将要解释你该如何做。

1.8 使用第三方包

在 Python 中，很多第三方的包都是可用的。很多这些包的二进制发布包连同安装程序对于大多数常用的操作系统都是可用的。如果一个二进制安装包可用的话，或者在包网站上或者对于 Linux 用户来说在你的包管理工具中，你应该使用它。因为这是让事情启动并运行的最简单的方式。如果一个二进制包不可用，则需要下载并安装基本包。

可在 Python 包索引(Python Package Index, PyPI)中找到很多这些第三方包。PyPI 的网站是 <https://pypi.python.org/pypi>。它们以一种特殊的格式被分发，这样它本身也需要安装一个第三方包。这种先有蛋还是先有鸡的情况经常会困扰初学者，所以本节描述如何搭建可以访问这些第三方包的环境。

PyPI 包附带了一种叫做蛋(egg)的格式。一个 Python 蛋既可以传送一个标准 Python 包，也可以传送一个用 C 语言或 Python 和 C 代码混合编写的二进制包。

Python 包装的未来：

蛋格式有很多问题，并且它自己也正在被称为轮子(wheel)的东西替代。这是更广泛的策略的全部。它想要让多种发布 Python 和应用的方法合理化。Python Package Authority 正在主导这个项目。最终，所有创建和安装 Python 包所需要的工具都应该在一个标准的 Python 安装中。这个路线图开始在 Python3.4 中起作用了，它把 pip 包含在标准的发布中。

如果想要创建自己的可发布包，应该去 Python Package Authority 的网站(<https://python-packaging-user-guide.readthedocs.org/en/latest/>)上查看最新指导。

安装一个蛋需要一个称为 pip 的工具。幸运的是，安装 pip 并不需要 pip。在 Python3.4 版本中，pip 被包含在标准库中，这在一定程度上简化了流程。如果 pip 没有被包含在你的 Python 版本中，可以自己安装 pip。浏览网站 https://pip.pypa.io/en/latest/reference/pip_install.html 并按照说明去做。

通过网页上的链接将 `get-pip.py` 文件下载到你电脑中任意方便的文件夹中。把工作环境换到那个文件夹，确认你连接到了互联网，并且执行下面的命令：

```
python get-pip.py
```


这可能会要花一些时间并且从互联网下载一些东西。你将会看到一些类似这样的消息：

```
$ python3 get-pip.py
Downloading/unpacking pip
  Downloading pip-1.5.2-py2.py3-none-any.whl (1.2MB): 1.2MB downloaded
Installing collected packages: pip
Successfully installed pip
Cleaning up...
```

一旦 `pip` 被安装好，就可以使用它来安装 PyPI 包，使用下面这个命令：

```
pip install SomePackageName
```

这个命令会安装指定包的最新版本。也可以用这个命令轻松卸载一个包：

```
pip uninstall SomePackageName
```

还有很多其他的选项可以使用，它们在 `pip` 文档页有详细的描述：<https://pip.pypa.io/en/latest/reference/index.html>。

并非所有的包都使用 `pip`，同时也存在其他的安装选项和工具。包文档应该解释需要做什么。

1.9 本章小结

在本章中回顾了 Python 的核心语言特性，了解了解释器环境、核心数据类型，以及语言控制结构和语法。也知道了如何通过写函数、类、模块和包来扩展 Python。

核心数据类型包括布尔(`bool`)、整数(`int`)、浮点数(`float`)以及特殊的 `None` 类型。Python 也支持多个容器类型，包括字符串、字节、列表、元组、字典和集。

控制结构涵盖了所有的结构化编程概念，包括序列、选择和迭代。选择是通过使用 `if/elif/else` 和一个条件表达式来完成的。有两种循环结构体支持迭代：`for` 和 `while`。`for` 会迭代一个集合或可迭代对象。`while` 则更加普通，还可能产生无限循环。Python 也通过 `try/except/finally` 结构体支持异常管理。

除了大量的内置函数和标准模块库，Python 也允许你通过使用 `def` 或 `lambda` 关键字写自己的函数来扩展它的功能。也可以创建自己的数据类型并创建这些类的实例来扩展标准数据类型。类的定义使用 `class` 关键字。函数和类可以保存在单独的文件中。这样就构成了 Python 模块。它们可以被导入到其他代码中，因此方便了跨程序复用。模块还可以被整理成包。包其实就是包含 `__init__.py` 文件的文件夹。

练习题

1. 如何在不同的 Python 数据类型间进行转换？在转换数据类型时，会产生什么数据质量问题？

- 2. 哪些 Python 容器类型可以作为字典中的键使用？哪些 Python 数据类型可以作为字典中的值使用？
- 3. 使用一个 if/elif 链编写一个示例程序，需要包含至少 4 个不同的选择表达式。
- 4. 编写一个 Python for 循环，重复一个消息 7 次。
- 5. 如何使用一个 Python while 循环创建一个无限循环？这可能会产生什么问题？如何解决这个问题？
- 6. 编写一个函数，计算一个给定底和高的三角形面积。
- 7. 编写一个类，并实现一个从 0 到 9 的循环计数器。也就是说，这个计数器从 0 开始，一直递增到 9，再被重置为 0，这样无限地重复这个循环。该类应该有 increment()和 reset()方法。reset()方法会返回当前的计数，然后将计数设回为 0。

练习题的答案在附录 A 中。

本章所学知识

主 题	关 键 概 念
Python 基础结构	在无参数的情况下可以调用 Python 解释器，得到一个交互式 shell。如果把一个文件名作为参数，解释器会执行它并退出
简单数据类型	Python 支持整型、浮点型、布尔型和 None 数据类型。类型名可以被用作类型转换函数。Python 的类型是对象，并且支持很多操作
容器数据类型	Python 支持 Unicode 和字节字符串，以及列表、元组、字典和集。字符串和元组是不可变的(不可以被修改)，而字典和集需要不可变的类型作为键。大多数集合都是可迭代的，并且可以用在 for 循环中
基本控制结构	Python 支持序列、选择和重复。序列只是简单的代码行；不需要块标识符或语句结束符。可以通过 if/elif/else 结构来进行选择。Python 提供了两种循环：for 和 while。 代码块是通过前一行结尾的冒号标明的。代码块在行下面需要被缩进。重新回到原来的缩进级别意味着代码块的结束
错误处理	Python 通过 try/except/else/finally 结构来支持异常处理。 用户可以定义自己的异常或为内置的错误添加参数
输入/输出	使用 input()函数，可以将用户输入作为字符串读取。 通过 print()函数，用户输出可以被显示出来。 文本文件可以被打开、读取和写入。使用 tell()和 seek()可以实现文件导航
定义函数	使用 def 或 lambda 关键字可以定义新函数。函数可以通过参数来接收输入，通过 return 关键字提供结果
定义类	类是使用关键字 class 定义的。类支持单继承和多重继承、多态、操作符重载和方法重写。类特性可以被当成属性和/或插槽 特性可以通过点符号来访问。类也是对象

(续表)

模块和包	模块只是包含 Python 代码的、任何被列在 sys.path 中文件夹内的文件。包就是包含一些模块和一个名为__init__.py 文件(可能为空)的文件夹。包也是模块。模块中的名称是通过点符号访问的
------	--

第 2 章

Python 脚本

本章主要内容：

- 通过操作系统访问和管理计算机资源
- 处理常见文件格式，比如 CSV 和 XML
- 操作日期和时间
- 应用自动化和访问它们的 API
- 在标准库的功能之外，使用第三方模块来扩展自动化

从 wrox.com 下载本章代码

可以在 wrox.com 网站找到本章涉及的代码。代码可以在 www.wrox.com/go/python-projects 的 Download Code 选项卡下找到。第 2 章代码位于 Chapter 2 download，名为 Chapter2.zip，每个文件都是根据本章提到的代码文件名命名的。

有时，你可能发现自己在处理一些涉及很多重复操作的任务。为了避免重复工作，我们可以在单个应用里编写宏来自动化那些操作。但是，如果操作跨越多个应用，宏就很难有效了。例如，如果对一个大的多媒体 Web 应用进行备份和存档，则可能必须去处理一个或多个媒体工具产生的内容、集成开发环境的代码和可能的数据库文件。相比于宏，你更需要一个外部的编程工具驱动每个应用或工具来执行它们各自的工作。

Python 非常适合这种类似于管弦乐编曲的角色。

什么是脚本？

对于不同的人来说，脚本意味着不同的事物。所以在这章看到它之前，我们必须首先要明确它的意思。它协调其他程序或应用的行为来执行一个任务，比如批量文件打印、或一个自动化工作流程，比如增加一个新用户。你可能在使用操作系统工具或类似办公生产力套装的大型通用包。想象一下，戏剧中的剧本告诉演员台词、如何站位以及何时进出场。

的方式。这就是 Python 脚本所做的。它协调其他程序的行为。

在本章，你会学到如何使用 Python 模块检查用户设置以及目录和文件的访问级别，为一个操作设置正确的环境，以及在脚本中启动和控制外部程序。你也会学到 Python 模块如何帮助你访问常用文件格式中的数据、如何处理日期和时间，以及如何使用非常强大的 `ctypes` 模块和专门用于 Windows 的 `pywin32` 包直接访问外部应用的低级别编程接口。

特定系统问题

从讨论的本质上来说，本章大部分的内容都是特定操作系统的。一些模块试图可移植到多个操作系统，然而其他模块在一些系统上只支持部分操作，还有一些对特定平台维护特定的模块。如果发现不好用的示例，确保它不是专为另一个不同的操作系统工作的。可能的话，我们会试着指出哪些模块是专为特定系统的。

同样，在“试一试”部分，输出通常也是特定于用户和操作系统的。所以不要期待会获得一模一样的结果。例如，在检查用户详情时，你应该会看到你自己的用户名，而不是示例中的用户名。

2.1 访问操作系统

大部分典型程序员使用操作系统执行的一些任务，比如收集用户信息或浏览文件系统，都可以使用 Python 的标准库模块以通用的方式完成(回想一下，模块是可以在多个程序间分享的可复用代码片段)。主要的模块都是以这种方式编写的：单个操作系统行为的特殊性都被隐藏在更高层的对象和操作之后。在本节中，你将研究模块 `os/path`、`pwd`、`glob`、`shutil` 和 `subprocess`。这些资料将专注于如何在一些常见的场景中使用这些模块。它并没有试图涵盖所有可能的情况或选项。

顾名思义，`os` 模块为许多操作系统特性提供访问。实际上，它拥有 `os.path` 子模块。这个子模块负责管理文件路径、名称和类型。在逐步完成本章的各种主题时，你会遇到 `os` 模块的其他子模块。这些不计其数的模块被整体地称为 `OS`(大写)模块，而实际的 `os` 模块被称为 `os`(小写)。如果熟悉 UNIX 系统编程，或者甚至使用 UNIX shell，比如 Bash，你会熟悉许多这样的操作。

操作系统主要用来管理计算机硬件的访问，包括 CPU、内存、存储和网络。它管理这些资源的访问以及进程的创建、调度和销毁。`OS` 模块函数可以了解和管理这些操作系统行为。接下来，你会看到如下常见的任务：

- 收集用户和系统信息
- 管理进程
- 确定文件信息
- 操纵文件
- 浏览文件夹



注意：Python 版本 3 中最大的改变是它持续地关注 Unicode 的国际化、识别和使用。os 模块也不例外。它的函数接受 Unicode 字符串。然而，底层的 OS 函数和文件名并不需要处理 Unicode。在那些情况下，Python 会使用 `sys.getfilesystemencoding()` 提供的系统编码把它转换为字节。这并不保证在所有情况下都好用，偶尔可能会抛出一个 `UnicodeError` 异常。在那些情况下，或者必须在调用函数之前把字符串转换成一个更加操作系统友好的格式，或者寻找其他方法解决问题。

2.1.1 获得关于用户和他们的电脑的信息

在探索 OS 模块时，首先要找出它们能够告诉你哪些用户信息。具体地说，可以找出用户 ID、登录名和一些默认设置。

就像 Python 中的大多数新事物一样，熟悉它们的最佳方式就是通过交互式提示符。所以启动 Python 解释器来试一试吧。

试一试：确定用户

在这个“试一试”中，你会找出当前用户的一些信息。完成以下步骤：

- (1) 启动 Python 解释器。
- (2) 在解释器中输入下面的代码：

```
>>> import os
>>> os.getlogin()
'agauld'
>>> os.getuid()                # Not Windows
1001
>>> import pwd                 # Not Windows
>>> pwd.getpwuid(os.getuid()) # Not Windows
pwd.struct_passwd(pw_name='agauld', pw_passwd='unused', pw_uid=1001,
pw_gid=513, pw_gecos='Alan Gauld,U-DOCUMENTATION\\agauld,
S-1-5-21-2472883112-933775427-2136723719-1001',
pw_dir='/home/agauld', pw_shell='/bin/bash')
>>> for id in pwd.getpwall():
...     print(id[0])
...
SYSTEM
LocalService
NetworkService
Administrators
TrustedInstaller
Administrator
```

```

agauld
Guest
HomeGroupUser$
????????
>>>

```

示例说明

在第一行导入 `os` 模块之后，你获得了登录名称字符串。通常这对于创建个性化的提示语和屏幕信息来说是最有用的。遗憾的是，对于 Windows 用户来说，到此为止了。其他部分代码只适合基于 UNIX 的系统。然而，也并非一无所有，因为也可以从环境变量中获得一些这样的信息。你会在 2.1.2 节了解到这些。

如果有一个基于 UNIX 的系统，那么使用 `os.getuid()` 可以获得操作系统产生的用户 ID，即一个数字值。之后可以在各种其他函数中使用它。下面代码导入了密码模块 `pwd` 并使用其中的函数将操作系统用户 ID 翻译成一组更加完整的信息，包括真实名称、默认 shell 以及主目录。很显然这样信息量更大。但是它需要使用首先从 `os.getuid()` 获得的 UID。另一个可用的函数 `os.getpwnam()` 接受登录名作为参数并返回同样的信息。最后，你使用 `pwd.getpwall()` 和一个 `for` 循环提取系统的所有用户名。

接下来，你会看到用户在他们创建的文件上有哪些权限。这是非常重要的，因为它影响着任何代码生成的文件。你可能需要暂时改变权限。例如，如果需要创建一个之后在程序中执行的文件，它需要有执行权限。在 UNIX 中，这些设置都被保存在一个名为 `umask` 或用户掩码的东西中。它是一个位掩码，就像你在第 1 章最后使用的那些。每一位代表着一个用户访问数据点，如下所述。

Python 允许你使用 `os.umask()` 函数查看 `umask` 值，甚至在 Windows 上也可以。但是，`os.umask()` 函数有一个小怪癖。它期待你向函数传入一个新值。然后它会设置这个值并返回旧值。但如果只是想找出当前的值，就不能这样做。相反，需要将 `umask` 设置为一个临时值，读取旧值，然后再将它设置为原始值。掩码的格式非常紧凑，包含三组三位的值，每一组分别对应着 Owner、Group 和 World 权限。



注意：3 位可以被简洁地表示为八进制数。出于这个原因，你经常会发现 UNIX 文档使用八进制数表示这些值。通过在八进制数前面加上 `0o` 来表示它们，Python 也可以使用八进制值，所以 `0o777` 表示所有位都为 1 的 9 位。

在组内，3 位分别对应着每一种访问——读、写或执行。这些是用显式的二进制符号最方便地表示出来。表 2-1 展示了每一个 3 位二进制值是如何映射到权限的。