第7章

Python高级特性

7.0 引言

在本章中，我们将介绍Python语言的一些高级概念，特别是面向对象的Python、文件的读写、异常处理、使用模块和因特网编程。

7.1 格式化数字

面临问题

你想把数字格式化为指定的小数位。

解决方案

你可以对数字应用格式串，具体如下所示。

```
>>> x = 1.2345678
>>> "x={:.2f}".format(x)
'x=1.23'
>>>
```

进一步探讨

用于格式化的字符串可以由常规文本和标志组成，并且需要用{和}进行分隔。函数format的参数（可以有任意多个）会根据格式说明符来取代标志。

在前面的示例中，格式说明符为:.2f，表示数字被指定小数点后面保留两位数，同时其类型为浮点型f。

如果你想把数字格式化为总长度为7位（不够用空格补齐）的话，那么就需要在小数

点前面再加上一个数字，具体如下所示。

```
>>> "x={:7.2f}".format(x)
'x=   1.23'
>>>
```

就本例而言，由于该数字长度只有 3 位，所以会在 1 之前填充 4 个空格。如果你想使用前导零的方式进行补齐的话，可以使用下列代码。

```
>>> "x={:07.2f}".format(x)
'x=0001.23'
>>>
```

下面是一个稍微复杂的例子，这里要使用摄氏度数和华氏度数来表示温度，具体代码如下所示。

```
>>> c = 20.5
>>> "Temperature {:5.2f} deg C, {:5.2f} deg F.".format(c, c * 9 / 5 + 32)
'Temperature 20.50 deg C, 68.90 deg F.'
>>>
```

参考资料

Python 语言中的格式化技术有一套完整的格式化语言（http://bit.ly/cwzPp5）。

7.2 格式化时间和日期

面临问题

你想把一个日期转化为字符串，并且按照特定的方式进行格式化。

解决方案

你可以将格式串应用到日期对象上面。

举例来说：

```
>>> from datetime import datetime
>>> d = datetime.now()
>>> "{:%Y-%m-%d %H:%M:%S}".format(d)
'2015-12-09 16:00:45'
>>>
```

进一步探讨

Python 格式化语言为格式化日期提供了许多特殊符号。

%Y、%m 和%d 分别对应于年、月和日。

其他常用于格式化日期的符号还有%B 和%Y，前者用于提供月份的全称，后者提供 4

位数字表示的年份，具体如下所示。

```
>>> "{:%d %B %Y}".format(d)
'09 December 2015'
```

参考资料

关于数字格式化的内容，请参考 7.1 节。

Python 语言中的格式化技术，涉及到一套完整的格式化语言。这方面的完整说明请参考 http://strftime.org/。

7.3 返回多个值

面临问题

你想编写一个返回多个值的函数。

解决方案

你可以返回 Python 元组，并使用多变量赋值语法。元组是一种 Python 数据结构，与列表比较相似，只不过元组是用小括号而非方括号括起来的。此外，两者的大小都是固定的。

例如，你可以建立一个函数，来将温度的绝对温标转换为华氏温标和摄氏温标。你可以让这个函数同时返回两种温标表示的温度，为此可以使用逗号将多个返回值隔开，具体如下所示。

```
>>> def calculate_temperatures(kelvin):
...     celsius = kelvin - 273
...     fahrenheit = celsius * 9 / 5 + 32
...     return celsius, fahrenheit
...
>>> c, f = calculate_temperatures(340)
>>>
>>> print(c)
67
>>> print(f)
152.6
```

调用该函数时，你只需在=前面提供跟返回值数量一致的变量即可，并且每个返回值都会按照同样的位置赋给相应的变量。

进一步探讨

当只有一两个值需要返回时，上面介绍的这种返回多个值的方式是比较合适的。但是，如果数据非常复杂的话，那么你会发现，使用 Python 面向对象特性来定义一个存放数据的类

将是更加优雅的一种解决方案。这样的话，你可以返回一个类的实例，而非元组。

参考资料

至于类的定义方法，请参考 7.4 节。

7.4 定义类

面临问题

你需要将相关数据和函数整合到一个类中。

解决方案

你可以定义一个类，并根据需要提供相应的成员变量。

下面的代码定义了一个类，用来表示通迅录记录。

```
class Person:
    '''This class represents a person object'''

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

在上面的类定义中的第一行，使用了三引号来指明这是一个文档性质的字符串。这个字符串通常是用来介绍该类的用途的。当然，这个字符串是完全可选的，但是为类添加了文档字符串之后，可以帮助人们了解这个类是做什么的。尤其是当定义的类是供别人使用的时候，这个字符串将格外有用。

与普通注释字符串不同的是虽然注释字符串本身不是有效代码，但是却与类紧密相联，所以，无论何时，你都可以通过下列命令来读取类的文档字符串。

```
Person. __doc__
```

在类定义的内部是构造函数，每当为该类新建一个实例的时候，都会自动调用这个函数。类与模板比较类似，因此在定义名为 Person 的类的时候，我们并没有创建任何实际的 Person 对象，直到像下面这样生成实例为止。

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

构造函数的命名，必须像上面那样，在单词 init 前后分别加上一个下画线。

进一步探讨

Python 与大部分面向对象编程语言的一个不同之处在于在类内部定义的所有方法，

都必须包含特殊变量 self 来作为其参数。在这种情况下，变量 self 就是对新建实例的引用。

从概念上讲，变量 self 与 Java 等其他编程语言中的特殊变量 this 是等价的。

这个方法中的代码会将提供给它的参数传递给成员变量。成员变量无需提前声明，但是，必须带有前缀 self.。

所以，下面的代码：

```
self.name = name
```

将会生成一个名为 name 的变量，它可以供类 Person 的所有成员访问，同时，该变量是利用传递给创建实例的调用的值来初始化的，具体如下所示。

```
p = Person("Simon", "1234567")
```

这样，我们就可以利用下面的代码来查看新建的 Person 对象 p 的名称是否为“Simon”了。

```
>>> p.name
Simon
```

对于复杂的程序来说，可以将每个类单独放到一个文件中，并且以该类的名称给文件命名，这是一种非常好的做法。此外，这种做法也使得将类转换为模块更加容易（见 7.11 节）。

参考资料

关于定义方法的内容，请参考 7.5 节。

7.5 定义方法

面临问题

你需要为类添加方法。

解决方案

下面的示例代码展示了如何在类的定义中包含一个方法。

```
class Person:
    '''This class represents a person object'''
    def __init__ (self, first_name, surname, tel):
        self.first_name = first_name
        self.surname = surname
        self.tel = tel

    def full_name(self):
        return self.first_name + " " + self.surname
```

方法 full_name 的作用是把一个人的姓和名连起来，并以空格作为间隔。

进一步探讨

实际上，你可以把方法看作是绑定到特定类上的一些函数，它们既可以使用这个类的成员变量，也可以不使用这个类的成员变量。所以，正如函数那样，你不仅可以在方法内部写入任何所需代码，也可以从一个方法中调用其他方法。

如果一个方法调用了同一个类中的其他方法，那么调用时必须在被调用方法之前添加前缀 self。

参考资料

关于定义类的内容，请参考 7.4 节。

7.6 继承

面临问题

你需要为一个现有的类制作一个特殊版本。

解决方案

你可以利用继承为现有的类创建一个子类，并为其添加新的成员变量和方法。

在默认的情况下，你新建的所有类都是 object 的子类。要想改变这种状况，在定义类的时候，可以在类名称之后的括号中加入想要使用的超类。下面的示例代码定义了一个 Person 的子类（Employee），并添加了一个新的成员变量（salary）和另外一个方法（give_raise）。

```
class Employee(Person):

    def __init__ (self, first_name, surname, tel, salary):
        super().__init__ (first_name, surname, tel)
        self.salary = salary
    def give_raise(self, amount):
        self.salary = self.salary + amount
```

需要注意的是上面的示例代码适用于 Python 3。对于 Python 2 来说，不允许通过这种方式来使用 super，而应该采用下列所示的方式。

```
class Employee(Person):

    def __init__ (self, first_name, surname, tel, salary):
        Person. __init__ (self, first_name, surname, tel)
        self.salary = salary

    def give_raise(self, amount):
```

```
        self.salary = self.salary + amount
```

进一步探讨

在上面的两个示例中，子类的初始化方法都是首先调用父类（超类）的初始化方法，然后才添加成员变量。这样做的好处在于无需在新的子类中重复初始化代码。

参考资料

关于定义类的内容，请参考 7.4 节。

Python 的继承机制实际上非常强大，并且支持多重继承，也就是说一个子类可以继承自多个超类。要想了解继承的更多内容，请参考 Python 的官方文档（http://bit.ly/17Y2c2k）。

7.7 向文件中写入内容

面临问题

你想向一个文件中写入某些内容。

解决方案

你可以分别使用 open、write 和 close 函数来打开一个文件，并向其中写入某些数据，然后关闭该文件。

```
>>> f = open('test.txt', 'w')
>>> f.write('This file is not empty')
>>> f.close()
```

进一步探讨

文件一旦打开，在其被关闭之前，你可以随意向其中写入内容，次数不限。需要注意的是使用 close 函数关闭文件是非常重要的， 因为，虽然每次写操作都应该即时更新文件，但是写入的内容也可能被缓存在内存中，所以这些数据还是有可能会丢失的。此外，如果不关闭文件的话，会导致该文件被锁住，以及其他程序无法打开该文件。

方法 open 需要两个参数，第一个参数是需要进行写操作的文件的所在路径。

这个路径既可以是基于当前工作目录的相对路径，也可以是从/目录开始的绝对路径。

第二个参数是文件的打开模式。如果要覆写现有的文件，或者文件不存在时按照指定的文件名创建文件的话，可以使用参数 w。

表 7-1 给出了文件模式字符的完整列表。你可以利用+来组合各种模式字符。

所以，如果要以二进制读的模式来打开一个文件的话，你可以使用下列代码。

```
>>> f = open('test.txt', 'r+b')
```

表 7-1　　文件模式模式说明

R	读
W	写
A	追加——将内容追加到现有文件末尾，而非覆盖现有文件
B	二进制模式
T	文本模式（默认）
+	r+w 组合模式的缩写

二进制模式允许你对二进制数据流进行读写操作，二进制数据流包括图像等数据，但是文本不属此列。

参考资料

若要读取文件的内容，请参考 7.8 节。若想进一步了解异常处理，请参考 7.10 节。

7.8　读文件

面临问题

你想把文件的内容读入到一个字符串变量中。

解决方案

为了读取文件的内容，你需要使用文件操作有关的 open、read 和 close 方法。下面的示例代码会把文件的所有内容读取到变量 s 中。

```
f = open('test.txt')
s = f.read()
f.close()
```

进一步探讨

此外，对于文本文件来说，你还可以使用 readline 方法以每次一行的方式来读取文件。

对于上例来说，如果文件不存在，或由于某种原因而处于不可读状态的话，系统就会抛出一个异常。为了处理这种情况，你可以在代码中加入 try/except 结构来处理异常，具体如下所示。

```
try:
    f = open('test.txt')
```

```
    s = f.read()
    f.close()
except IOError:
    print("Cannot open the file")
```

参考资料

关于文件写操作以及文件的各种打开模式，请参考 7.7 节。

关于异常处理的详细介绍，请参考 7.10 节。

7.9 序列化（Pickling）

面临问题

你想把一个数据结构的所有内容全部保存到一个文件中，以便在程序下次运行时可以将其读取出来。

解决方案

你可以使用 Python 的序列化功能，按照某种格式将数据结构转存到文件中，并且，这种格式能在今后自动将转存的内容以原来的形式恢复到内存中。

在下面的示例代码中，我们将会把一个复杂的列表结构保存到一个文件中，具体如下所示。

```
>>> import pickle
>>> mylist = ['some text', 123, [4, 5, True]]
>>> f = open('mylist.pickle', 'w')
>>> pickle.dump(mylist, f)
>>> f.close()
```

为了将一个文件的内容反序列化到一个新的列表中，可以使用下列代码。

```
>>> f = open('mylist.pickle')
>>> other_array = pickle.load(f)
>>> f.close()
>>> other_array
['some text', 123, [4, 5, True]]
```

进一步探讨

序列化技术几乎能够完美适用于任何你能够想到的数据结构上面，而非仅限于列表上。

被序列化的内容，通常都会以文本的形式保存到文件中，尽管这些文本的可读性非常强，但是通常情况下，你根本无需查看或编辑这些文本文件。

参考资料

关于文件写操作以及文件的各种打开模式，请参考 7.7 节。

7.10 异常处理

面临问题

如果程序运行期间出现某些错误，你不仅希望能够捕获这些错误，并且还能够以对用户更加友好的形式来展示这些错误信息。

解决方案

你可以使用 Python 提供的 try/except 结构。

下面的示例代码取自 7.8 节，它能够捕获文件打开期间的所有问题。

```
try:
    f = open('test.txt')
    s = f.read()
    f.close()
except IOError:
    print("Cannot open the file")
```

由于你把可能容易出错的命令都封装到了 try/except 结构中，所以，发生任何错误时，你能够在任何错误信息被显示之前就提前捕获它们，这样的话，你就有机会利用自己的方式来处理它们了。就本例来说，你就可以利用更加友好的消息“Cannot open the file”来通知出错情况了。

进一步探讨

一个常见的运行时异常的情况（文件访问期间除外），是当你访问列表时下标出现越界的时候。例如，如果一个列表只有 3 个元素，当你试图访问第 4 个元素的时候，就会出现这种情形：

```
>>> list = [1, 2, 3]
>>> list[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

错误和异常是以层级结构的形式来组织的，并且你可以根据需要来捕获特殊或一般的异常。

Exception 类非常接近层次树的顶部，并且几乎可以用来捕获所有的异常。此外，你还可以使用单独的 except 段来捕获不同类型的异常，并使用不同的方式来进行处理。如

果你不规定任何异常类型的话，就会捕获所有类型的异常。

另外，Python 还允许在错误处理中使用 else 和 finally 子句。

```
list = [1, 2, 3]
try:
    list[8]
except:
    print("out of range")
else:
    print("in range")
finally:
    print("always do this")
```

如果没有发生异常，else 子句就会被执行，但是无论是否发生异常，finally 子句都是要被执行的。

每当发生异常时，你都可以通过异常对象获取有关的详细情况，需要注意的是只有使用了 as 关键字之后才能如此，具体如下所示。

```
>>> list = [1, 2, 3]
>>> try:
...     list[8]
... except Exception as e:
...     print("out of range")
...     print(e)
...
out of range
list index out of range
>>>
```

这不仅允许你使用自己的方式来处理错误，同时还保留了原始的错误消息。

参考资料

关于 Python 异常类的层次结构的详细内容，你可以参考 Python 的相关文档（http://bit.ly/1cqqHV7）。

7.11 使用模块

面临问题

你希望在自己的程序中使用 Python 的模块。

解决方案

你可以使用 import 命令。

```
import random
```

进一步探讨

对于 Python 语言来说，有非常多的模块（有时候称为库）可供选用。并且，大部分的模块都是作为标准程序库的一部分而包含在 Python 中了，至于剩下的其他模块，你仍然可以通过下载来安装到 Python 中。

标准 Python 库中的模块提供了随机数、数据库访问、各种互联网协议、对象序列化等功能。

模块如此之多的一个后果是有可能发生冲突，例如，两个模块是否会包含同名的函数等。为了避免这些冲突，当导入模块时，你可以规定模块的哪些部分是可以供外部访问的。

例如，如果你只是使用如下所示的命令：

```
import random
```

这样是不会出现任何冲突的，因为你只有在被访问的任何函数或变量前面加上前缀 random 的情况下，才能够访问它们。（顺便提一句，下一节将会介绍 random 包。）

反之，如果你使用了下列命令的话，那么无需使用任何前缀，就可以访问模块内的所有内容。除非你对所用模块内的所有函数了然于胸，否则产生冲突的可能性就会非常大。

```
from random import *
```

在这两种极端情况之间，你还可以选择显式规定自己程序所需的模块中的组件，以便在无需使用前缀的情况下也可以访问它们。

举例来说：

```
>>> from random import randint
>>> print(randint(1,6))
2
>>>
```

第三种选择是通过关键字 as 为模块提供一个更简洁或更有意义的名称，然后利用这个名称来引用该模块。

```
>>> import random as R
>>> R.randint(1, 6)
```

参考资料

关于 Python 所提供模块的权威清单，请参考 http://docs.Python.org/2/library/。

7.12 随机数

面临问题

你需要在指定范围内生成一个随机数。

解决方案

你可以使用 random 库。

```
>>> import random
>>> random.randint(1, 6)
2
>>> random.randint(1, 6)
6
>>> random.randint(1, 6)
5
```

生成的随机数的大小会介于两个参数之间（其中包括这两个参数）——就本例来说，我们是在模拟掷骰子。

进一步探讨

这里生成的数字并非真正意义上的随机数，而是大家所熟悉的伪随机数序列。也就是说，它们是一个很长的数列，当长度足够大的时候，如果从中取数字的话，就会表现出统计意义上的随机分布。对于游戏来说，这种特性已经足够好了，但是如果你想生成彩票号码的话，则需要求助于专门的定制化硬件了。电脑不擅于随机，因为随机的确不是电脑的天性。

随机数通常用于从列表中随机选择元素。为此，你可以先生成一个位置下标，然后使用这个下标访问元素。此外，random 模块还专门为此提供了一个函数。你可以尝试下列代码。

```
>>> import random
>>> random.choice(['a', 'b', 'c'])
'a'
>>> random.choice(['a', 'b', 'c'])
'b'
>>> random.choice(['a', 'b', 'c'])
'a'
```

参考资料

若要进一步了解 random 包，请访问其官方文档（http://docs.python.org/2/library/random.html）。

7.13 利用 Python 发送 Web 请求

面临问题

你需要使用 Python 将网页内容读取到一个字符串中。

解决方案

Python 提供了一个扩展库，可以用来发送 HTTP 请求。

下面的 Python 2 代码会将 Google 主页内容读取到字符串变量 contents 中。

```
import urllib2
contents = urllib2.urlopen("https://www.google.com/").read()
print(contents)
```

进一步探讨

如果你使用的是 Python 3 而非 Python 2 的话，那么你还需要将代码修改为下面的样子。

```
import urllib.request
response = urllib.request.urlopen('http://python.org/')
contents = response.read()
print(contents)
```

在读取 HTML 之后，你很可能想要搜索它，并从中提取真正需要的文本部分。为此，你可以使用相应的字符串处理函数（参见 5.14 节和 5.15 节）。

参考资料

利用 Python 与因特网交互的相关内容，请参考第 15 章。

7.14 Python 的命令行参数

面临问题

你想从命令行运行一个 Python 程序，并向其传递某些参数。

解决方案

你可以导入 sys 模块，并使用其 argv 属性，具体如下例所示。这样的话，会返回一个数组，数组的第一个元素就是程序的名称。该数组的其他元素，则是命令行中在程序名之后输入的所有参数（需要用空格分隔）。

```
import sys

for (i, value) in enumerate(sys.argv):
    print("arg: %d %s " % (i, value))
```

从命令行运行该程序，并在后面带上某些参数，结果如下所示。

```
$ python cmd_line.py a b c
arg: 0 cmd_line.py
arg: 1 a
```

```
arg: 2 b
arg: 3 c
```

进一步探讨

在命令行中指定参数的能力对于在开机期间（见 3.21 节）或在指定时间（见 3.23 节）自动运行 Python 程序来说是非常有用的。

参考资料

关于从命令行运行 Python 的基本知识，请参考 5.4 节。

如果想输出 argv 的话，可以使用列表枚举（见 6.8 节）。

7.15 从 Python 运行 Linux 命令

面临问题

你想从自己的 Python 程序中运行一个 Linux 命令或程序。

解决方案

你可以使用 system 命令。

举例来说，如果想要删除一个名为 myile.txt 的文件（该文件位于 Python 的启动目录中）的话，可以使用下列代码。

```
import os
os.system("rm myfile.txt")
```

进一步探讨

有时，你不仅要像上例那样“摸黑”执行命令，而且还得捕获命令的响应。比如，假设你想使用“hostname”命令来查看树莓派的 IP 地址（参见 2.2 节）。在这种情况下，你可以利用 subprocess 库中的 check_output 函数。

```
import subprocess
ip = subprocess.check_output(['hostname', '-I'])
```

这里，变量 ip 将用来存放树莓派的 IP 地址。与 system 不同的是，check_output 要求将命令本身及其所有参数都作为数组的单独元素来提供。

参考资料

关于 OS 库的相关文档，请访问 http://www.pythonforbeginners.com/OS/pythons-OS-module。

有关 subprocess 库的详细信息，请访问 https://docs.python.org/3/library/subprocess.html。

在 14.4 节中，你将会看到利用 subprocess 库在液晶屏幕上显示树莓派的 IP 地址、主机名和时间的示例代码。

7.16 从 Python 发送电子邮件

面临问题

你想从一个 Python 程序中发送电子邮件。

解决方案

Python 提供了一个支持简单邮件传输协议（SMTP）的库，你可以利用它来发送电子邮件。

```
import smtplib

GMAIL_USER = 'your_name@gmail.com'
GMAIL_PASS = 'your_password'
SMTP_SERVER = 'smtp.gmail.com'
SMTP_PORT = 587

def send_email(recipient, subject, text):
    smtpserver = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)
    smtpserver.ehlo()
    smtpserver.starttls()
    smtpserver.ehlo
    smtpserver.login(GMAIL_USER, GMAIL_PASS)
    header = 'To:' + recipient + '\n' + 'From: ' + GMAIL_USER
    header = header + '\n' + 'Subject:' + subject + '\n'
    msg = header + '\n' + text + ' \n\n'
    smtpserver.sendmail(GMAIL_USER, recipient, msg)
    smtpserver.close()

send_email('destination_email_address', 'sub', 'this is text')
```

要想使用上面的代码向指定的地址发送电子邮件，首先要根据自己的电子邮件登录凭证来修改 GMAIL_USER 和 GMAIL_PASS。如果使用的不是 Gmail 的话，那么你还需要修改 SMTP_SERVER 的值，同时 SMTP_PORT 的值也可能需要修改。

此外，你还需要修改最后一行中电子邮件的目的地。

进一步探讨

Send_email 将 smtplib 库的用法简化为单个函数，以便于在自己的项目中重复使用。

从 Python 发送电子邮件的能力为各种项目开启了机会之门。例如，你可以使用 PIR 之类的传感器，以便在侦测到移动发生时发送电子邮件。

参考资料

关于使用 IFTTT Web 服务发送电子邮件的示例代码，请参考 15.3 节。

关于利用树莓派发送 HTTP 请求的内容，请参考 7.13 节。

关于 smtplib 的详细介绍，请参考 http://docs.python.org/2/library/smtplib.html。

更多与互联网有关的示例代码，请参考第 15 章。

7.17 利用 Python 编写简单 Web 服务器

面临问题

你需要创建一个简单的 Python Web 服务器，但是又不想运行完整的 Web 服务器栈。

解决方案

你可以使用 Python 库 bottle 来运行一个纯 Python 的 Web 服务器来响应 HTTP 请求。

为了安装 bottle，可以使用下列命令。

```
sudo apt-get install python-bottle
```

下面的 Python 程序（在图书资源中名为 bottle_test，资源地址 http://www.raspberrypicookbook.com）提供了显示树莓派时间的简单功能。图 7-1 展示了从网络任意位置上的浏览器连接到树莓派时所显示的页面内容。

```
from bottle import route, run, template
from datetime import datetime

@route('/')
def index(name='time'):
    dt = datetime.now()
    time = "{:%Y-%m-%d %H:%M:%S}".format(dt)
    return template('<b>Pi thinks the date/time is: {{t}}</b>', t=time)

run(host='0.0.0.0', port=80)
```

为了运行该程序，你需要作为超级用户来执行它。

```
$ sudo python bottle_test.py
```

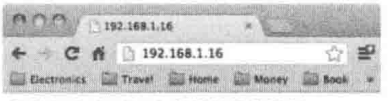

图 7-1 浏览 Python bottle Web 服务器

这个示例代码还需要做进一步的解释。

在 import 命令后，命令@route 会把 URL 路径/与其后的处理函数关联起来。

该处理函数将格式化日期与时间，然后返回供浏览器渲染的 HTML 字符串。就本例来说，它使用了一个模板，该模板随后会被相应的值替换掉。

最后面的 run 所在行的代码才是启动该 Web 服务进程的实际代码。由于 80 端口是 Web 服务的默认端口，所以，如果你希望使用其他端口的话，就需要在服务器地址之后利用：添加相应的端口号。

进一步探讨

只要你喜欢的话，你可以在程序中定义任意数量的路由和处理程序。

对于小而简单的 Web 服务器项目来说，bottle 是理想之选，同时，由于它是用 Python 语言编写的，所以非常易于编写一个处理函数来控制硬件，使其通过浏览器页面作为接口来响应用户的请求。在第 15 章中，你还会看到其他利用 bottle 的示例代码。

参考资料

如果希望进一步了解 bottle 的话，可以参考该项目的相关文档（http://bottlepy.org/docs/dev/）。

关于利用 Python 格式化日期和时间的详细内容，请参考 7.2 节。

更多与互联网有关的示例代码，请参考第 15 章。

7.18 同时进行多件事情

面临问题

当 Python 程序忙于某件事情的时候，你希望还能同时进行其他事情。

解决方案

你可以使用 Python 的 thread 库。

下面的代码将建立一个线程，并且该线程运行时将中断主线程的计算。该示例代码可以从 *Raspberry Pi Cookbook* 网站的下载区获取，网站具体地址为 http://www. raspberrypicookbook.com/。

```
import thread, time, random

def annoy(message):
    while True:
        time.sleep(random.randint(1, 3))
```

```
        print(message)

thread.start_new_thread(annoy, ('BOO !!',))

x = 0
while True:
    print(x)
    x += 1
    time.sleep(1)
```

控制台上将输出类似于下面的内容。

```
$ python thread_test.py
0
1
BOO !!
2
BOO !!
3
4
5
BOO !!
6
7
8
```

当你使用 Python 的 thread 库让一个新线程运行起来时，必须指定让哪个函数作为该线程来运行。在本例中，这个函数名为 annoy，它包含一个循环语句，在一个介于 1 到 3 秒的随机间隔之后打印输出一则消息，并且该循环将一直进行下去。

为了启动该线程，需要调用 thread 模块中的 start_new_thread 函数。

这个函数需要用到两个参数：第一个参数是需要运行的函数的名称（本例为 annoy），第二个参数是一个元组，存放传递给该函数的所有参数（本例为'BOO!!'）。

你会发现，当主线程正在忙着计数的时候，每隔几秒就会被作为 annoy 函数运行的线程所打断。

进一步探讨

像上面这些线程，有时候也被称为轻量级进程，因为从效果上看，类似于同时运行多个程序或进程。不过，对于线程来说，其优点在于在同一个程序中运行的多个线程可以访问相同的变量，并且当程序的主线程退出时，在主线程中启动的任何进程也照常如此。

参考资料

有一篇关于 Python 线程入门的好教材，地址为 http://www.tutorialspoint.com/Python/python_multithreading.htm。

7.19 让 Python 无所事事

面临问题

你希望让 Python 消磨一段时间。比如，当向终端发送消息的这段延迟时间内，你可能会想让 Python 消磨时间。

解决方案

你可以使用 time 库中的 sleep 函数，具体如下面的示例代码所示。该代码可以从本书的网站（http://www.raspberrypicookbook.com）上面下载，程序名称为 sleep_test.py。

```
import time

x = 0
while True:
    print(x)
    time.sleep(1)
    x += 1
```

在打印下一个数字之前，该程序的主循环会延迟几秒时间。

进一步探讨

函数 time.sleep 会以秒数作为其参数，如果你想减少延迟时间的话，可以使用小数。例如，为了推迟 1 毫秒，你可以使用 time.sleep（0.001）。

对于任何持续时间不确定或者只持续零点几秒的循环来说，为其设置一个较小的延迟是个不错的主意，因为当 sleep 被调用的时候，处理器就会被释放出来供其他进程使用。

参考资料

关于 time.sleep 降低 Python 程序的 CPU 负载机制的有趣论述，请参考 http://raspberrypi.stackexchange.com/questions /8077/how-can-i-lower-the-usage-of-CPU-For-this-Python-program。

7.20 将 Python 应用于树莓派版 Minecraft

面临问题

你已经对 Python 有了深入了解，现在，你想将其应用于 Minecraft 上面。

解决方案

利用树莓派版 Minecraft 提供的 Python 接口，你可以在该服务器运行期间通过 Python

语言与其进行交互。

虽然可以在 LXTerminal 会话和 Minecraft 游戏之间来回切换，但是每次窗口失去焦点时游戏都会暂停，因此最好还是从另一台计算机上使用 SSH 连接树莓派（见 2.7 节）。这种做法的额外好处是你可以在游戏中实时观察 Python 脚本的运行情况。

下面的示例代码可以在当前位置创建楼梯（见图 7-2），具体代码可以从 *Raspberry Pi Cookbook* 的网站（http://www.raspberrypicookbook.com/）的下载区下载。

```
from mcpi import minecraft, block
mc = minecraft.Minecraft.create()

mc.postToChat("Lets Build a Staircase!")

x, y, z = mc.player.getPos()

for xy in range(1, 50):
    mc.setBlock(x + xy, y + xy, z, block.STONE)
```

上面的 Python 库是 Raspbian 预安装的，如果你的系统中没有该库的话，可以更新自己的系统（见 1.5 节）。

导入该库之后，变量 mc 将被赋予一个 Minecraft 类的新实例。

方法 postToChat 会向玩家的屏幕发送一则消息，告诉玩家即将建造一个楼梯。

变量 x、y 和 z 将绑定到玩家的位置上，随后，for 循环每次将 x 和 y（y 是高度）的值增加 1 的时候，都会调用 setBlock 方法来创建一级楼梯（见图 7-2）。

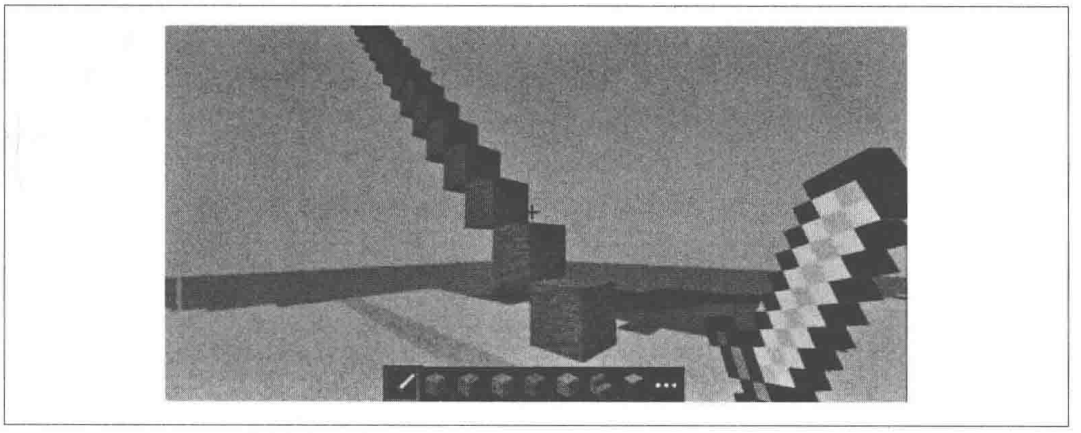

图 7-2　利用 Python 在树莓派版 Minecra 中建造楼梯

进一步探讨

该 mcpi 库不仅可以用来进行聊天、发现其他玩家位置和摆放建筑零件，而且还提供了

许多其他方法，以便让你可以：

- 发现指定坐标处的建筑组件的 ID；
- 找出谁在玩和瞬移它们；
- 确定玩家位置；
- 确定玩家面向的方向。

对于这个类，尚未有权威的文档，只有一篇比较有用的博客文章（http://www.stufaboutcode.com/2013/04/minecra -pi-edition-api-tutorial.html）。

参考资料

要想了解树莓派版 Minecraft 的更多信息，请参考 4.7 节。

关于在树莓派上面运行 Minecraft 服务器的方法，请参考 4.8 节。