

---

# Python UNIX和Linux系统 管理指南

*Noah Gift & Jeremy M. Jones* 著

杨明华 谭励 等译



O'REILLY®

*Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo*

O'Reilly Media, Inc.授权机械工业出版社出版

机械工业出版社

# 目录

<b>序 .....</b>	<b>1</b>
<b>前言 .....</b>	<b>3</b>
<b>第1章 Python简介 .....</b>	<b>11</b>
为什么要选Python .....	11
学习的动力 .....	17
一些基础知识 .....	18
在Python中执行命令 .....	19
在Python中使用函数 .....	23
通过Import语句实现代码复用 .....	26
<b>第2章 IPython .....</b>	<b>31</b>
安装IPython .....	32
基础知识 .....	33
从功能强大的函数获得帮助 .....	40
UNIX Shell .....	44
信息搜集 .....	61
自动和快捷方式 .....	74
本章小结 .....	79

<b>第3章 文本</b>	<b>80</b>
Python的内建功能及模块	80
ElementTree	124
本章小结	127
<b>第4章 文档与报告</b>	<b>129</b>
自动信息收集	129
手工信息收集	132
信息格式化	141
信息发布	147
本章小结	151
<b>第5章 网络</b>	<b>152</b>
网络客户端	152
远程过程调用	163
SSH	169
Twisted	171
Scapy	177
使用Scapy创建脚本	179
<b>第6章 数据</b>	<b>182</b>
引言	182
使用OS模块与Data进行交互	183
拷贝、移动、重命名和删除数据	184
使用路径、目录和文件	186
数据比较	189
合并数据	192
对文件和目录的模式匹配	197
包装rsync	199
元数据：关于数据的数据	200
存档、压缩、映像和恢复	202
使用tarfile模块创建TAR归档	203
使用tarfile模块检查TAR文件内容	205

<b>第7章 SNMP .....</b>	<b>208</b>
引言 .....	208
对SNMP的简要介绍 .....	208
IPython与Net-SNMP .....	211
查找数据中心 .....	214
使用Net-SNMP获取多个值 .....	217
创建混合的SNMP工具 .....	222
Net-SNMP扩展 .....	224
SNMP设备控制 .....	227
整合Zenoss的企业级SNMP .....	228
<b>第8章 操作系统什锦.....</b>	<b>229</b>
引言 .....	229
Python中跨平台的UNIX编辑 .....	230
PyInotify .....	240
OS X .....	241
Red Hat Linux系统管理 .....	246
Ubuntu管理 .....	246
Solaris系统管理 .....	247
虚拟化 .....	247
云计算 .....	248
使用Zenoss从Linux上管理Windows服务器 .....	255
<b>第9章 包管理.....</b>	<b>258</b>
引言 .....	258
SetupTools和Python Egg .....	259
使用easy_install .....	259
easy_install的高级特征 .....	261
创建egg .....	267
进入点及控制台脚本 .....	271
使用Python包索引注册一个包 .....	272
Distutils .....	274

Buildout .....	276
使用Buildout .....	277
使用Buildout进行开发.....	280
virtualenv.....	280
EPM包管理 .....	285
EPM总结：真的非常简单 .....	289
<b>第10章 进程与并发 .....</b>	<b>290</b>
引言 .....	290
子进程 .....	290
使用Supervisor来管理进程.....	299
使用Screen来管理进程 .....	301
Python中的线程.....	302
进程 .....	313
Processing模块 .....	314
调度Python进程.....	317
daemonizer.....	318
本章小结 .....	321
<b>第11章 创建GUI .....</b>	<b>322</b>
GUI创建理论 .....	322
生成一个简单的PyGTK应用 .....	323
使用PyGTK创建Apache日志浏览器 .....	325
使用Curses创建Apache日志浏览器.....	329
Web应用 .....	332
Django .....	333
本章小结 .....	351
<b>第12章 数据持久性 .....</b>	<b>353</b>
简单序列化.....	353
关系序列化.....	372
本章小结 .....	381

<b>第13章 命令行</b>	<b>382</b>
引言	382
基本标准输入的使用	383
Optparse简介	384
简单的Optparse使用模式	385
Unix Mashups：整合Shell命令到Python命令行工具中	392
整合配置文件	397
本章小结	399
<b>第14章 实例</b>	<b>400</b>
使用Python管理DNS	400
使用OpenLDAP、Active Directory以及其他Python工具实现LDAP	402
Apache日志报告	404
FTP镜像	410
<b>附录 回调</b>	<b>415</b>

# Python简介

## 为什么要选Python

如果你是一位系统管理员，那么可能已对Perl、Bash、ksh或其他一些脚本语言有所了解，甚至已经使用了其中的一种或几种。我们通常利用脚本语言来完成一些重复、乏味的工作，使用后，工作完成的速度和准确性都远远高于不使用它们时的情况。实际上，所有的语言都是工具，它们为完成工作提供了便捷的手段，而这些工具的价值也正是体现在它们能够帮助人们把工作做得更好。我们有理由相信Python是一个非常有价值的工具，它会帮助你更为高效地完成工作。

选择Python是因为它比Perl、Bash、Ruby或其他语言更好么？事实上，我们很难对各种编程语言的优劣进行排序，因为这些工具与使用它们的程序员的思维习惯有着紧密联系。编程是一种主观性很强的活动，与程序员直接相关。一门优秀的编程语言，必须适合于使用它的人。因此，在这里我们不会争论Python是否比其他语言更好，但我们将给出理由说明为什么Python是个不错的选择，也将说明为什么Python尤其适合完成系统管理任务。

Python之所以是一门优秀的编程语言，第一个原因就在于Python十分容易学习。如果一门语言无法在很短的时间内帮助人们迅速提高工作效率，那么它作为语言的魅力也就消失了。想一想在使用一种程序语言去完成一些工作之前，需要花费几周甚至几个月的时间去学习这门语言，值得吗？答案恐怕是否定的，对于系统管理员尤其如此。如果你是一位系统管理员，就会感觉到总有处理不完的工作，花费大量时间学习一门语言，简直难以想象。而使用Python则不同，在几个小时内，你就可以写出规范的脚本程序，完全不用花上几天或是几周时间。的确如此，对于一门语言，如果人们不能很容易就掌握它，并很快地学会编写该语言的代码，那么就应该问一问是否有学习这门语言的必要了。

然而，如果一门语言过于简单，不具备完成复杂工作的能力，它同样也没有太高的价

值。因此，Python之所以被称为一门非常优秀的编程语言，第二个原因就在于Python既简单易学，又可以完成你能够想象到的任何复杂任务。当需要一行一行地读取某个日志文件，然后输出其中一些基本信息时，Python可以胜任。或是当需要解析一个日志文件，提取出其中的信息，将其中的IP地址与过去三个月中每个日志文件里的IP地址相比较（这些日志文件都存储在关系数据库中），然后将比较的结果存储在关系数据库中时，Python同样也可以处理。实际上，Python正用于处理一些非常复杂的问题，如基因序列分析、多线程Web服务和繁重的统计分析任务。也许你并不会遇到上述那样复杂的工作任务，但是当我们需要去完成一些比较复杂的任务时，有Python这门语言相助终归是一件好事情。

此外，虽然你能够使用一门语言编写一些复杂的代码，但这些代码的维护却让你伤透脑筋，这同样不是一件好事。尽管Python也有代码维护问题，但是Python能够帮助我们用简单的语法结构来实现复杂的编程思想。在编写代码过程中，代码简化是一个极为重要的因素，因为简化的代码能够使后续的代码维护工作变得简单、轻松。Python在代码简化方面做得非常出色，即使我们需要维护几个月都没再看过的那些代码，也并非难事。甚至当我们使用之前从未见过的陌生Python代码时，也同样简单方便。由于其语法和通用词汇十分简洁、精练，Python语言很容易让人一经学习就能够长时间使用。

Python之所以是一门非常优秀的编程语言的另一个原因是其可读性非常好。Python使用空格（whitespace）来分隔代码块的开始和结尾。代码的缩进格式帮助人们很容易理解程序的流程。此外，Python可以说是“基于词”（word-based）的语言，也就意味着，Python语言在使用一些特殊字符的同时，特征经常是以关键词或词库的形式实现的。强调词而不是强调特殊字符，使得Python代码具有很好的可读性，并且十分易于理解。

以上，我们总结了Python的一些优点，接下来，我们将分别对Python、Perl和Bash编写的代码示例进行对比分析。通过对比，我们将可以更清楚地看到Python的优势。以下是一个简单的Bash示例程序，显示数字1、2与字母a、b的所有组合，代码如下所示：

```
#!/bin/bash

for a in 1 2; do
    for b in a b; do
        echo "$a $b"
    done
done
```

对比下面使用Perl完成的代码：

```
#!/usr/bin/perl

foreach $a ('1', '2') {
    foreach $b ('a', 'b') {
        print "$a $b\n";
```

```
    }  
}
```

我们可以看到，这是一个非常简单的嵌套循环。以下是使用Python中的for循环完成同样循环嵌套机制的代码：

```
#!/usr/bin/env python  
  
for a in [1, 2]:  
    for b in ['a', 'b']:  
        print a, b
```

接下来，我们再对Bash、Perl和Python中条件语句的使用进行对比。以下代码是Bash中一个简单的if/else条件结构，用于检测指定的文件路径是否为一个目录。

```
#!/bin/bash  
  
if [ -d "/tmp" ] ; then  
    echo "/tmp is a directory"  
else  
    echo "/tmp is not a directory"  
fi
```

下面是使用Perl编写的具有相同功能的代码：

```
#!/usr/bin/perl  
  
if (-d "/tmp") {  
    print "/tmp is a directory\n";  
}  
else {  
    print "/tmp is not a directory\n";  
}
```

最后是使用Python编写的代码，实现相同的检测功能：

```
#!/usr/bin/env python  
  
import os  
  
if os.path.isdir("/tmp"):  
    print "/tmp is a directory"  
else:  
    print "/tmp is not a directory"
```

Python语言的另一个优势是它对面向对象编程（OOP）的支持。事实上，如果你不想使用OOP来编写程序，完全可以不用，而如果你想使用OOP，就会发现，在Python中使用OOP编程是相当容易的。利用OOP技术，可以简单、清晰地将问题进行分解，将一系列功能模块组成独立的“事物”或“对象”。Bash不支持面向对象技术，Perl和Python则完全支持。下面的代码是采用Perl语言定义的一个类：

```
▶ package Server;
use strict;

sub new {
    my $class = shift;
    my $self = {};
    $self->{IP} = shift;
    $self->{HOSTNAME} = shift;
    bless($self);
    return $self;
}

sub set_ip {
    my $self = shift;
    $self->{IP} = shift;
    return $self->{IP};
}

sub set_hostname {
    my $self = shift;
    $self->{HOSTNAME} = shift;
    return $self->{HOSTNAME};
}

sub ping {
    my $self = shift;
    my $external_ip = shift;
    my $self_ip = $self->{IP};
    my $self_host = $self->{HOSTNAME};
    print "Pinging $external_ip from $self_ip ($self_host)\n";
    return 0;
}

1;
```

在下面的Perl代码中，使用了上述定义的类：

```
▶ #!/usr/bin/perl

use Server;

$server = Server->new('192.168.1.15', 'grumbly');
$server->ping('192.168.1.20');
```

上述代码直接使用了OO模块，看起来十分简单。但如果对Perl如何使用OOP技术进行编程不太熟悉，那么你仍需要花费一些工夫来弄清Perl是如何处理OOP问题的。

接下来，我们看看在Python中定义并使用类的代码：

```
▶ #!/usr/bin/env python

class Server(object):
```

```

def __init__(self, ip, hostname):
    self.ip = ip
    self.hostname = hostname
def set_ip(self, ip):
    self.ip = ip
def set_hostname(self, hostname):
    self.hostname = hostname
def ping(self, ip_addr):
    print "Pinging %s from %s (%s)" % (ip_addr, self.ip, self.hostname)

if __name__ == '__main__':
    server = Server('192.168.1.20', 'bumble')
    server.ping('192.168.1.15')

```

在上述Perl与Python的示例中，分别演示了两种语言实现面向对象设计的基本方法。这两个示例程序说明，这两种语言在实现各自的目标时具有不一样的优势。它们可以完成同样的工作，但完成的方式各不相同。因此，如果你希望使用OOP，那么Python正好提供了对OOP的支持，可以简单方便地将OOP技术应用到你所编写的程序中。

Python的另外一个优势并不是来自语言本身，而是来自Python社区。在Python社区中，你可以找到许多完成各类工作的成功经验，还可以学到许多能够选择使用（或是不应使用的）的语言特性。Python语言自身支持的一些句式（phrasing）虽然能够实现某些功能，但在社区中已形成的共识会帮助你远离一些不好的句式。例如，在Python模块的顶端使用“`from module import *`”是有效的，但是在社区中并不赞同这种写法，而是建议使用“`:import module`”或“`:from module import resource`”。因为，将一个模块的全部内容导入到另一个模块的命名空间，有时会引起很多麻烦。例如当你试图了解模块是如何工作的、调用了什么函数和从哪里获得这些函数时，问题就会变得十分复杂。社区中这些特殊的约定会帮助你写出更为清晰的代码，有助于其他人对你的代码进行维护。在编写代码时遵循这些公共约定，会使你在编程过程中少走许多弯路。应该说这是一件非常值得重视的事情。

Python标准库（Python Standard Library）是Python另一个非常值得称道的方面。你可能曾经听说过有人在形容Python时，说它是“连电池都包括在内的”（“batteries included”）。之所以这样评价Python，正是因为Python标准库可以帮助你完成几乎所有的工作，无须再到别的地方去寻找支持模块。Python标准库并不是直接内置（built-in）在语言中的，但并不影响其提供各种各样丰富的功能支持，例如正则表达式功能、套接字、线程、日期/时间功能，XML解析、配置文件解析，文件和目录功能、数据持久性、单元测试功能，还包括http、ftp、smtp和nntp客户端库文件等。因此，一旦Python安装完毕，支持所有这些功能的模块就可以根据需要加载到脚本中。你将获得以上列出的所有功能，而这些功能都是来自Python自身的，无须再从其他地方获取。所有这些功能将在使用Python完成工作的过程中，为你提供有力的帮助。

Python另一个真正的优势在于能够方便地获取各种各样的第三方软件包。除了Python标准库中提供的大量库文件外，在互联网上也有大量的库和实用工具可以使用，通过简单的Shell命令行就可以进行安装。在Python包索引（Python Package Index, PyPI, <http://pypi.python.org>）上，任何人都可以将编写的Python包上传到该网站，分享给其他人使用。在我们写这本书的时候，已经有超过3800个包文件可供下载使用。其中包括将在接下来的几章中讲述的IPython, Storm（一个对象-关系映射器，将在第12章中讲述）和Twisted（一个网络框架，将在第5章中讲述），这仅仅是3800多个包中的其中三个。一旦开始使用PyPI，你会发现它是查找和安装有用软件包几乎不可或缺的途径。

我们所看到Python的许多优点，都是源于Python的中心哲学。当我们在Python提示符下输入“import this”时，将会看到Tim Peters所写的“The Zen of Python”，内容如下：

```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

译文如下：

Python的禅宗，作者Tim Peters

美丽优于丑陋。

清楚优于含糊。

简单优于复杂。

复杂优于繁琐。

平坦优于曲折。

宽松优于密集。

重要的是可读性。

特殊的案例不足以特殊到破坏规则。

尽管实践可以打破真理。

错误却不可置之不理。

除非另有明确要求。

面对模棱两可，拒绝猜测。

总会有一个——最好是只有一个——显而易见的方式来明辨。

哪怕这种方式在开始的时候可能并不明显 —— 除非你是荷兰人（译注1）。  
现在有比没有好。

尽管没有经常好于现在。

如果如何实现很难被解释清楚，那么这个想法就是一个坏想法。

如果如何实现可以被很好的解释，那么这是一个好想法。

命名空间就是一个非常好的想法 —— 让我们在这方面多做些工作吧！

尽管这些描述并不是在各个层次的语言开发中都能教条式地强制贯彻的，但其思想精髓却可以渗透到语言的方方面面。我们发现这些思想非常有用，这或许也是我们日复一日选择使用Python的理由，而这些思想中的哲学，也会在我们使用语言的过程中不断显现。如果你也能体会到这些思想精髓，Python对你来说或许就是一个非常不错的选择。

## 学习的动力

如果你刚从书店拿起这本书，或是正在网上在线阅读本书的简介，你或许会问自己学习Python会有多难，甚至会问Python是否值得一学。目前，尽管Python发展得如火如荼，已经吸引了相当一部分人的眼球，但许多系统管理员正在使用的仍是Bash或Perl。如果你就是这些已经掌握了Bash或Perl的系统管理员中的一分子，那么在得知Python非常好学易懂时，也就完全没有必要再担心Python难学了。事实上，Python被许多人认为是最容易讲授和学习的一门语言，尽管这只是来自个人的体会，但的确就是这样！

如果你已经对Python有所了解，或者是使用另一种编程语言的大师级人物，那么你可以跳过下面的简介，直接阅读以后各章，通过使用我们的示例，立即开始工作。经过努力，我们编写了一些示例程序，也许可以帮助你完成一些工作。这些示例程序包括：如何使用SNMP自动发现和监测子网，如何转换到交互式Python的Shell（即IPython），如何构建数据处理管道，如何使用对象-关系映射器编写自定义元数据管理工具，如何实现网络编程，如何编写命令行工具等等。

如果你有Shell编程或脚本编程背景，那么学习Python时就一点不用担心学不会了。你可以相当轻松地完成Python的学习，需要的仅是学习的动力、好奇心和毅力，这些因素也会促成你拿起这本书，开始阅读简介部分。

你可能还会有一些顾虑，也许会受到一些与编程相关的负面消息的影响。例如，一个常见的巨大误解就是，仅有小部分思维奇特且属于精英类型的人才适合学习编程。而事实的真相是，任何人都可以学习编程。另一个同样巨大的误解是，只有攻读并获得计算机专业学位才是一个人成长为真正的软件工程师的必经之路。实际上，一些非常有成就的软件开发人员并没有获得任何工程方面的学位。他们中的许多人获得的是哲学、新闻

---

译注1： 荷兰人被认为具有非常外向和直接的性格。

学、营养学和英语专业的学位，却成了非常优秀的Python程序员。因此，具有计算机专业的学位并不是学习Python的必要条件，当然，有计算机专业的学位也并不坏。

另外一个有趣的、也是经常被误解的事情是认为学习编程必须从十几岁开始，否则就永远错过了学习的时机。这一误解使得一些人倍感欣慰，因为他们有幸在人生中遇见了鼓励他们从小就开始学习编程的人。然而这简直是完全没有事实根据的观点，尽管在年青时就开始学习编程对以后进一步学习确实是有帮助，但是，年龄并不是学习Python的必要条件。学习Python确实不仅仅是年青人的事，正如我们所听到的，实际上有着数不清的在年近30岁、年近40岁、年近50岁，甚至在更大年龄段的人也同样能够成功学习编程的案例。

至此，如果你已经有了学习Python的动力，那么可以说，作为读者，你已经具备了许多人没有的优势。如果你已经决定拿起这本书并开始学习Python，那么你最想知道的可能是如何通过Shell执行命令。掌握了如何在控制终端执行命令之后，本章对Python做简要介绍的任务也就完成了。如果你已经非常确信需要学习Python编程了，那么可以立即开始阅读下面的章节。如你不是很确信，那么请重新阅读本节，它会让你相信自己有能力学会Python编程。Python编程确实很简单，如果你决定学习Python，它将会改变你的生活。

## 一些基础知识

接下来我们将学习一些基础知识，我们将要对Python所做的简要介绍，与之前我们所能看到的其他介绍不同，我们将同时使用被称为IPython的交互式Shell和常规的Bash shell。你需要打开两个终端窗口，一个是IPython，另一个是Bash。在每一个示例中，我们都将会对Python和Bash程序进行对比。首先需要根据你的平台，下载并安装IPython的正确版本。下载的地址为<http://ipython.scipy.org/moin/Download>。如果由于一些原因，无法下载并完成安装，你也可以使用普通的Python Shell。你可以下载一个包括本书所需软件的虚拟机，虚拟机中包括了一个预先配置并可以直接使用的IPython。只需输入“ipython”，就会看到一个命令提示符。

一旦安装了IPython，并且出现了IPython shell提示符，将会看到下面的内容：

```
[ngift@Macintosh-7]:~$ ipython
Python 2.5.1 (r251:54863, Jan 17 2008, 19:35:17)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.2 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.
```

In [1]:

IPython shell与普通的Bash shell有些相似，可以执行ls、cd、pwd这样的命令。通过阅读接下来的章节，你会学习到更多的命令。本章主要讲述如何学习Python，因此不会对命令做更多讲解。

如果在Python终端上输入下面的内容，可以看到如下结果：

→ In [1]: print "I can program in Python"  
I can program in Python

如果使用的是Bash终端，输入下面的内容，可以看到如下结果：

→ [ngift@Macintosh-7][H:10688][J:0]# echo "I can program in Bash"  
I can program in Bash

在这两个示例中，看不出Python与Bash有什么太大的区别，而这正是Python的神奇之处。

## 在Python中执行命令

如果一天中会花费很多时间在终端里输入各种命令，那么你可能需要学会执行一些语句，例如将处理结果重定向，输出到文件或输出给另一个UNIX命令。接下来我们进行一些比较，从而了解一些Bash下执行的命令在Python中是如何执行的。在Bash终端中，键入如下内容：

→ [ngift@Macintosh-7][H:10701][J:0]# ls -l /tmp/  
total 0  
-rw-r--r-- 1 ngift wheel 0 Apr 7 00:26 file.txt

而在Python终端中，需要键入的内容如下：

→ In [2]: import subprocess

In [3]: subprocess.call(["ls","-l ","/tmp/"])  
total 0  
-rw-r--r-- 1 ngift wheel 0 Apr 7 00:26 file.txt  
Out[3]: 0

在上述Bash示例中，可以很清晰地看到，仅使用了一个非常简单的ls命令。但对于Python示例则不同，如果之前从没有见过Python代码，Python示例看起来可能会令人觉得有些怪异。你可能会想问“真见鬼，import subprocess究竟是什么意思”？Python之所以强大，原因之一就是Python可以载入其他模块，或包含其他文件，并在新的程序中做到代码复用。如果你比较熟悉Bash中的“sourcing”一个文件，就会发现相似之

处。在上例所示的特定情况下，你只需要知道加载了模块subprocess，并且知道使用该模块的语法。我们之后会具体解释subprocess和import是如何工作的，现在直接复制下面的代码即可：

```
▶ subprocess.call(["some_command", "some_argument", "another_argument_or_path"])
```

在Python中，可以就像在Bash中一样使用shell命令。给一点提示：你可以创建Python版本的ls命令。在另一个终端或另一个终端窗口中打开你经常使用的文本编辑器，将上述代码写入到该文件中，且将该文件命名为pyls.py，最后使用命令“chmod +x pyls.py”将该文件修改为可执行文件。参见例1-1。

#### 例1-1：Python包装ls命令

```
▶ #!/usr/bin/env python
#Python wrapper for the ls command

import subprocess
subprocess.call(["ls", "-l"])
```

如果现在运行该脚本，将获得与在命令行使用ls -l命令完全相同的结果，如下所示：

```
▶ [ngift@Macintosh-7]:[H:10746][J:0]# ./pyls.py
total 8
-rwxr-xr-x 1 ngift staff 115 Apr 7 12:57 pyls.py
```

虽然这个示例看起来十分简单（事实上也确实简单），却可以给出一个在系统编程中使用Python的通用思路。我们经常需要使用Python对脚本或Unix命令进行“包装”(wrap)。实际上，如果在文件中一行接一行地写下命令，然后运行该文件，就可以说你已经开始在写一些基本的脚本了。接下来让我们看一个简单示例。编写例1-2所示的代码，或是直接将以下代码进行剪切和粘贴，然后运行脚本文件pysysinfo.py和bashsysinfo.sh。这些脚本文件都可以在本章的源代码中找到。例1-2和1-3如下所示：

#### 例1-2：显示系统信息脚本——Python

```
▶ #!/usr/bin/env python
#A System Information Gathering Script
import subprocess

#Command 1
uname = "uname"
uname_arg = "-a"
print "Gathering system information with %s command:\n" % uname
subprocess.call([uname, uname_arg])

#Command 2
diskspace = "df"
diskspace_arg = "-h"
print "Gathering diskspace information %s command:\n" % diskspace
subprocess.call([diskspace, diskspace_arg])
```

### 例1-3：显示系统信息脚本——Bash

```
► #!/usr/bin/env bash  
#A System Information Gathering Script  
  
#Command 1  
UNAME="uname -a"  
printf "Gathering system information with the $UNAME command: \n\n"  
$UNAME  
  
#Command 2  
DISKSPACE="df -h"  
printf "Gathering diskspace information with the $DISKSPACE command: \n\n"  
$DISKSPACE
```

如果把这两个脚本都读一遍，会发现它们看起来非常相似。如果分别运行这两个脚本，所看到的输出结果也是完全相同的。需要注意的是，在使用`subprocess.call`时，将命令与参数完全分开的写法并不是必需的，也可以像下面这样写：

```
► subprocess.call("df -h", shell=True)
```

到目前为止，我们已经学习了很多，但仍没能将`import`和`subprocess`完全解释清楚。在Python脚本程序中我们加载了`subprocess`模块，`subprocess`模块已经包含了使用Python实现系统调用的代码。

正如之前提到的，加载模块（例如加载`subprocess`）仅仅是将可以使用的代码文件加载进来，也可以创建自己的模块或文件，供以后重复使用，这与加载`subprocess`模块的方式相同。模块并没什么神奇，只不过是一个写有代码的文件罢了。IPython shell的一个非常好的优点就是可以对模块或文件进行检查，查看其内部可用的属性。对于Unix，这非常类似于在`/usr/bin`目录下运行`ls`命令。如果你恰好刚开始使用Ubuntu或是Solaris系统，而已经用惯了Red Hat系统，那么你可能会在`/usr/bin`下使用`ls`命令去查看`wget`、`curl`或是`lynx`命令是否可用。现在如果你想使用在`/usr/bin`目录下找到的某个工具，只要简单地输入命令名称即可，例如`/usr/bin/wget`。

像`subprocess`这样的模块是非常简单的。使用IPython，你可以通过tab自动完成功能来查看模块中是否有可用的工具。让我们使用tab自动完成功能来查看`subprocess`中所有可用的属性。注意，一个模块仅仅是包含了代码的文件。下面是在IPython中，使用tab自动完成功能对`subprocess`进行查询的示例。

```
► In [12]: subprocess.  
subprocess.CalledProcessError subprocess._hash_ subprocess.call  
subprocess.MAXFD subprocess._init_ subprocess.check_call  
subprocess.PIPE subprocess._name_ subprocess.errno  
subprocess.Popen subprocess._new_ subprocess.fcntl  
subprocess.STDOUT subprocess._reduce_ subprocess.list2cmdline  
subprocess._all_ subprocess._reduce_ex_ subprocess.mswindows
```

subprocess._builtins_	subprocess._repr_	subprocess.os
subprocess._class_	subprocess._setattr_	subprocess.pickle
subprocess._delattr_	subprocess._str_	subprocess.select
subprocess._dict_	subprocess._active	subprocess.sys
subprocess._doc_	subprocess._cleanup	subprocess.traceback
subprocess._file_	subprocess._demo_posix	subprocess.types
subprocess._getattribute_	subprocess._demo_windows	

如果需要执行相同的命令，只需要输入：

→ import subprocess

然后输入：

→ subprocess.

接下来按Tab键使用其自动完成功能来查看可用属性。在示例中的第三列出现了 subprocess.call。现在如果想查看更多如何使用subprocess.call的信息，输入如下内容：

→ In [13]: subprocess.call?

```
Type:      function
Base Class: <type 'function'>
String Form: <function call at 0x561370>
Namespace:  Interactive
File:      /System/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/
           subprocess.py
Definition: subprocess.call(*popenargs, **kwargs)
Docstring:
Run command with arguments. Wait for command to complete, then
return the returncode attribute.

The arguments are the same as for the Popen constructor. Example:
retcode = call(["ls", "-l"])
```

上例中，在属性的后面使用问号来调用manpage手册页查询相关信息。如果想查询UNIX下某个工具如何使用，也可以直接简单地输入：。

→ man name\_of\_tool

如果希望查询模块中的其他属性，也可以使用与上例中查询subprocess.call类似的方法。在IPython中，在一个属性之后输入问号，就会找到该属性的相关信息，而且包含该属性的文档也会被打印出来。如果在标准库中使用该方法查询更多的属性，你会找到正确使用这些属性的十分有用的帮助信息。记往，也可以使用Python标准库文档来完成查询。

查看文档时，“Docstring”是正式的官方词汇，我们通过示例演示了如何查询 subprocess.call，以及subprocess.call的功能描述。

## 小结

现在你已经掌握了足够的知识，可以称自己是一名Python程序员了。你知道了如何编写一个简单的Python脚本，如何将Bash脚本翻译成Python脚本并执行它，最后，你知道了如何查询不熟悉的模块或属性的文档，从而获得帮助。在下一节中，你将看到如何将一系列连续的命令组织到函数中。

## 在Python中使用函数

在前面一节中，我们学会了一条接一条地连续执行多个命令，这一点非常有用，因为这意味着可以让一些以前必须手工完成的事情变得可以自动完成。进一步来讲，如果要自动执行这些代码，就需要创建函数。如果你对Bash或其他语言中的函数还不太了解，那么可把函数想象成一个小脚本。函数允许创建一个代码块，代码块中的每一行代码属于该函数，并且在被调用时，代码块中的所有代码是被一起调用的。这与我们写的封装了两行命令的Bash脚本有点相似。Bash脚本与函数的不同之处在于，你可以包含许多函数脚本。最后你可以在脚本中编写多个函数，每个函数包含一组代码，那么这组代码就会在适合的时间被调用执行。

现在我们需要进一步讨论一下空格（whitespace）的问题。在Python中，嵌套代码是通过统一的缩进格式来维护的。在其他的语言中，例如Bash，当定义了一个函数，需要使用括号将函数代码包括起来。在Python中，必须在括号内缩进你的代码。如果你是一个初学者，这可能会造成一点点混乱，但过不了很久，你就会熟悉这种写法，而且还会意识到这对于提高代码的可读性多么有帮助。如果在使用这些交互式示例程序过程中遇到麻烦，仔细查看一下这些代码是否正确地缩进了。最普通的经验就是使用tab来实现缩进，一个tab可以缩进4个空格。

让我们看看在Python和Bash中都是如何处理的。如果还有一个打开的IPython shell，则无须再创建一个Python脚本文件，当然如果你愿意的话，也可以创建一个。在IPython提示符下，交互地输入下面的内容：

```
In [1]: def pyfunc():
...:     print "Hello function"
...:
...:

In [2]: pyfunc
Out[2]: <function pyfunc at 0x2d5070>

In [3]: pyfunc()
Hello function

In [4]: for i in range(5):
...:     pyfunc()
```

```
...:  
...:  
Hello function  
Hello function  
Hello function  
Hello function  
Hello function
```

在这个示例中，可以看到在函数中放了一个打印语句。之后不仅调用了该函数，而且根据需要还进行了多次调用。在第[4]行中，我们使用了一个编程习惯或技巧，实现了对函数的5次执行。如果之前没看到过这种用法，只需要明白它让函数执行了5次即可。

在Bash shell中也可以完成类似的工作。下面是其中的一种实现方式：

```
bash-3.2$ function shfunc()  
> {  
>     printf "Hello function\n"  
> }  
bash-3.2$ for (( i=0 ; i < 5 ; i++ ))  
> do  
>     shfunc  
> done  
Hello function  
Hello function  
Hello function  
Hello function  
Hello function
```

在这个Bash示例中，正如之前在Python代码示例中所做的一样，我们创建了一个简单的函数shfunc，然后调用它连续执行了5次，值得注意的是，与Python相比，Bash示例更为繁琐。注意一下Bash的for循环与Python的for循环之间的差异。如果是第一次接触Bash或IPython函数，你应该在IPthon窗口中多做一些函数的练习，然后再开始之后的学习。

函数没有什么神奇的，如果是第一次接触函数，练习交互式地编写多个函数是理解函数，去掉其神秘面纱最好的办法。下面是一些函数的简单示例：

```
In [1]: def print_many():  
...:     print "Hello function"  
...:     print "Hi again function"  
...:     print "Sick of me yet"  
...:  
...:
```

```
In [2]: print_many()  
Hello function  
Hi again function  
Sick of me yet
```

```
In [3]: def addition():  
...:     sum = 1+1
```

```
...: print "1 + 1 = %s" % sum
...:
...:

In [4]: addition()
1 + 1 = 2
```

除了上面的几个示例外，这里还有一些简单示例，可以在你的机器上一起运行试试。现在，我们回到显示系统信息的脚本，并将其转变成函数。见例1-4。

#### 例1-4：转换Python显示系统信息脚本：pysysinfo\_func.py

```
#!/usr/bin/env python
#A System Information Gathering Script
import subprocess

#Command 1
def uname_func():

    uname = "uname"
    uname_arg = "-a"
    print "Gathering system information with %s command:\n" % uname
    subprocess.call([uname, uname_arg])

#Command 2
def disk_func():

    diskspace = "df"
    diskspace_arg = "-h"
    print "Gathering diskspace information %s command:\n" % diskspace
    subprocess.call([diskspace, diskspace_arg])

#Main function that call other functions
def main():
    uname_func()
    disk_func()

main()
```

鉴于我们对函数已有的经验，这个转换示例将我们之前脚本中的代码简单地放到函数中，然后使用main函数进行一次性调用。如果不太熟悉这种风格，你可能不知道，这种在脚本中定义多个函数然后由一个main函数进行调用的方法十分普遍。主要原因之一就是如果你想在其他程序中再次使用这个脚本，可以独立地调用该函数，也可以使用main方法进行联合调用。关键取决于你加载模块之后的决定。

如果没有流程控制或main函数，所有代码在被加载时就会被立即执行。这对于一次性执行的脚本非常适用，但是如果你计划创建一个可重复使用的工具，则应该创建函数，封装指定的功能操作，然后通过main函数来执行整个脚本。

为了进行对比，我们将之前的Bash显示系统信息脚本也转化为函数，参见例1-5。

### 例1-5：转换Bash显示系统信息脚本——bashsysinfo\_func.sh

```
#!/usr/bin/env bash
#A System Information Gathering Script

#Command 1
function uname_func ()
{
    UNAME="uname -a"
    printf "Gathering system information with the $UNAME command: \n\n"
    $UNAME
}

#Command 2
function disk_func ()
{
    DISKSPACE="df -h"
    printf "Gathering diskspace information with the $DISKSPACE command: \n\n"
    $DISKSPACE
}

function main ()
{
    uname_func
    disk_func
}

Main
```

看完Bash示例之后，你可能会觉得它与Python有相当多的相似之处。我们创建了两个函数，然后通过main函数进行调用。如果这是你第一次使用函数，我们强烈建议你将Bash和Python脚本中的main方法注释掉（在Bash或Python脚本的每一行行首加上一个#），然后再运行一次。当再次运行时，你会发现输出结果彻底没有了，这是因为程序在执行时没有调用这两个函数。

学习到现在，你已经是一名能够使用Bash或Python编写简单函数的程序员了。作为程序员，编写程序的过程就是学习的过程。所以，此时我们强烈建议你修改这两个Bash和Python代码中的系统调用程序，将其变成自己的内容。如果可以在脚本中添加一些新的函数，并成功通过main函数实现了调用，那么应该说你干得已经相当不错了，可以好好犒劳一下自己。

## 通过Import语句实现代码复用

在学习新东西的过程中，总会有这样一个问题，那就是如果它十分抽象，类似微积分，那么我们很难有正当的理由去留意它。恐怕你压根不会记得什么时候曾在便利店中使用了高中时期所学的数学知识。在之前的示例中，我们向你演示了如何在脚本中创建函数，从而取代一行接一行的shell命令。我们也曾提到，模块实际上就是一段脚本，或者

说是文件中的一段代码。这没有什么特别，但是确实需要按特定的方式进行组织，以便于将来在别的程序中进行复用。这也正是为什么你应该特别留意的原因。接下来，我们载入之前的显示系统信息的Bash和Python脚本，并且执行。

如果你已经关闭了IPython或Bash窗口，那么重新打开，我们会非常快地向你展示为什么函数对于代码复用如此重要。我们采用Python创建的第一个脚本是由一系列命令组成的文件，文件名为*pysysinfo.py*。在Python中，一个模块对应一个文件，反之亦然，我们将脚本文件载入到IPython中。记住你不需要特别指定载入文件的扩展名为*.py*。事实上，如果这样做了，载入反倒不会成功。下面是我们在Noah的Macbook Pro笔记本上所做的操作：

```
In [1]: import pysysinfo
Gathering system information with uname command:

Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2: /
Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386
Gathering diskspace information df command:

Filesystem      Size   Used  Avail Capacity Mounted on
/dev/disk0s2    93Gi  88Gi  4.2Gi  96%   /
devfs          110Ki  110Ki  0Bi   100%  /dev
fdesc          1.0Ki  1.0Ki  0Bi   100%  /dev
map -hosts     0Bi   0Bi   0Bi   100%  /net
map auto_home  0Bi   0Bi   0Bi   100%  /home
/dev/disk1s2    298Gi 105Gi 193Gi  36%   /Volumes/Backup
/dev/disk2s3    466Gi 240Gi 225Gi  52%   /Volumes/EditingDrive
```

哇，结果非常棒，是吧？如果你加载了一个完全是Python代码的文件，看起来运行得很好。但是，事实上这里面也存在一些问题。如果你计划执行Python代码，通常是作为脚本或程序的一部分从命令行来执行。使用“import”进行载入操作可以帮助我们实现代码复用。那么问题就来了：如果只想执行一部份脚本，打印输出有关磁盘容量部分代码执行结果，应该怎么做呢？你的回答也许是这不可能。实际并非如此，这个问题完全可以解决，这也正是我们使用函数的原因。如本例所示，函数允许我们对在什么时间执行代码以及执行脚本中哪个部分的代码进行控制，而不是一次将所有代码都执行一遍。不要仅仅听我说，自己也应亲自去试试。只要你加载一个包含多个命令的函数脚本，就会明白我的意思。

下面是IPython终端的输出结果：

```
In [3]: import pysysinfo_func
Gathering system information with uname command:

Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2:
Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386
Gathering diskspace information df command:
```

Filesystem	Size	Used	Avail	Capacity	Mounted on
/dev/disk0s2	93Gi	88Gi	4.1Gi	96%	/
devfs	110Ki	110Ki	0Bi	100%	/dev
fdesc	1.0Ki	1.0Ki	0Bi	100%	/dev
map _hosts	0Bi	0Bi	0Bi	100%	/net
map auto_home	0Bi	0Bi	0Bi	100%	/home
/dev/disk1s2	298Gi	105Gi	193Gi	36%	/Volumes/Backup
/dev/disk2s3	466Gi	240Gi	225Gi	52%	/Volumes/EditingDrive

现在我们使用不包含函数的脚本来获得相同的输出结果。你或许会有些迷惑，这是一个好现象。想知道为什么会获得相同的输出结果，只需要查看一下源码即可。如果你是在主目录下，打开另一个终端标签或窗口，查看脚本pysysinfo\_func：

```
#Main function that call other functions
def main():
    uname_func()
    disk_func()

main()
```

问题是之前创建的**main**函数再次出现。一方面我们希望在命令行运行脚本来获得输出结果，但是另一方面，在加载时，我们不希望一次获得所有的输出，而是希望模块既可以作为一个脚本从命令行直接执行，也可以当成一个可复用的模块使用。这种需求在Python中非常普遍。解决方法就是通过像下面这样修改脚本的最后几行从而改变**main**函数的调用方式，如下所示：

```
#Main function that call other functions
def main():
    uname_func()
    disk_func()

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

这是一个解决上述问题的比较常用的技巧。任何缩进在这个表达式之后的代码都可以从命令行执行。通过替换你脚本中相应位置的代码，或者载入脚本pysysinfo\_func\_2.py，可以看到这一应用的效果。

现在，如果回到IPython解释器，并且载入这一新脚本，我们可以看到以下内容：

```
In [1]: import pysysinfo_func_2
```

这时，修改之后的**main**方法没有被调用。因此，在代码中利用了这一技巧，我们就得到了三个可以在其他程序中使用，或是可以在IPython shell中交互使用的函数。还记得之前我们提到的，仅调用打印磁盘容量的函数，而不用调用执行其他命令的函数的想法吗？首先，我们需要重新回顾一下之前已经介绍的技术。还记得可以使用Tab键的自动完成功能吧，它会显示出所有可用的属所，如下所示：

```
In [2]: pysysinfo_func_2.  
pysysinfo_func_2.__builtins__  
pysysinfo_func_2.__class__  
pysysinfo_func_2.__delattr__  
pysysinfo_func_2.__dict__  
pysysinfo_func_2.__doc__  
pysysinfo_func_2.__file__  
pysysinfo_func_2.__getattribute__  
pysysinfo_func_2.__hash__  
pysysinfo_func_2.disk_func  
pysysinfo_func_2.main  
pysysinfo_func_2.py  
pysysinfo_func_2.pyc  
pysysinfo_func_2.subprocess  
pysysinfo_func_2.uname_func
```

在这个例子中，我们将忽略有双下划线的内容，因为这些是特殊方法，已经超出了本章简介所涉及的内容。由于IPython也是一个常规的shell，所示它识别文件名，以及以`.psc`为扩展名的字节编译的（byte-compiled）Python文件。一旦忽略了这些有双下划线的名字，我们就会看到`pysysinfo_func_2.disk_func`。让我们继续下一步，调用函数：

```
In [2]: pysysinfo_func_2.disk_func()  
Gathering diskspace information df command:  


| Filesystem    | Size  | Used  | Avail | Capacity | Mounted on            |
|---------------|-------|-------|-------|----------|-----------------------|
| /dev/disk0s2  | 93Gi  | 89Gi  | 4.1Gi | 96%      | /                     |
| devfs         | 111Ki | 111Ki | 0Bi   | 100%     | /dev                  |
| fdesc         | 1.0Ki | 1.0Ki | 0Bi   | 100%     | /dev                  |
| map -hosts    | 0Bi   | 0Bi   | 0Bi   | 100%     | /net                  |
| map auto_home | 0Bi   | 0Bi   | 0Bi   | 100%     | /home                 |
| /dev/disk1s2  | 298Gi | 105Gi | 193Gi | 36%      | /Volumes/Backup       |
| /dev/disk2s3  | 466Gi | 240Gi | 225Gi | 52%      | /Volumes/EditingDrive |


```

或许你现在已经意识到了，函数总是通过被调用，或是在函数名后加“`()`”来执行的。在这个示例中，仅运行了文件中三个函数中的一个：调用的函数是`disk_func`。最终实现了代码复用。我们可以载入以前写的代码，并且交互式地运行需要的部分。当然，我们也运行了其他两个函数`uname_func`和`main`，这两个函数也是之前分别编写的。让我们看一下：

```
In [3]: pysysinfo_func_2.uname_func()  
Gathering system information with uname command:  
Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2:  
Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386
```

```
In [4]: pysysinfo_func_2.main()  
Gathering system information with uname command:  
Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2:  
Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386  
Gathering diskspace information df command:
```

```
Filesystem      Size   Used   Avail Capacity Mounted on  
/dev/disk0s2    93Gi   89Gi   4.1Gi  96%     /  
devfs          111Ki  111Ki  0Bi   100%    /dev  
fdesc          1.0Ki  1.0Ki  0Bi   100%    /dev  
map -hosts     0Bi    0Bi    0Bi   100%    /net
```

```
map auto_home    0Bi    0Bi    0Bi    100%    /home
/dev/disk1s2    298Gi   105Gi   193Gi   36%    /Volumes/Backup
/dev/disk2s3    466Gi   240Gi   225Gi   52%    /Volumes/EditingDrive
```

如果仔细观察，你会发现我们同时运行了其他两个函数。记住，`main`函数会立即执行全部代码。

通常，编写可复用模块就是为了将来在一个新的脚本中可以一遍又一遍地使用该部分代码。例1-6在另一个脚本中展示了如何使用其中一个函数`disk_func`。

#### 例1-6：使用import进行代码复用：new\_pysysinfo

```
#Very short script that reuses pysysinfo_func_2 code
from pysysinfo_func_2 import disk_func
import subprocess

def tmp_space():
    tmp_usage = "du"
    tmp_arg = "-h"
    path = "/tmp"
    print "Space used in /tmp directory"
    subprocess.call([tmp_usage, tmp_arg, path])

def main():
    disk_func()
    tmp_space()

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

这个例子不仅演示了对之前编写的代码的复用，也展示了如何使用Python特定的语法加载我们需要的函数。代码复用的有趣之处也在于，通过载入之前所编写程序中的函数，可以实现一个完全不同的程序。注意，在`main`方法中包含其他模块中的函数`disk_func()`，以及我们创建的新函数。

在这一节，我们学习了代码复用的知识，了解了代码复用的强大之处，同时也看到，代码复用实现起来又是多么简单。简而言之，我们在文件中加入了一个或两个函数，如果还希望能以脚本方式运行，则需要加入特定的`if __name__ == "__main__"`语法。之后，我们就可以加载这些函数到IPython中，或是简单地在另一个脚本中进行复用。学习上述内容之后，你就可以写一些非常复杂的Python模块，通过多次复用来创建一个新的工具了。这时的你，应该已经成长为一名高手了。

# IPython

Python的优点之一是其交互式解释器，也称为shell。shell提供了一种能够快速实现灵感、检验特性的方法，以及交互式的模块界面，能够将一些需要两三行脚本才能完成的任务一次性完成。通常我们编写代码时，会采用同时运行文本编辑器和Python的方式（稍后会有介绍，这实际上运行的就是IPython），通过交互地使用编辑器和shell，也就是在两者之间切换来完成程序的编写。我们经常需要将代码从编辑器复制到shell或从shell复制到编辑器。这种方式使得我们可以即时看到代码在Python中的处理结果，并且可以快速地在文本编辑器中编写需要的代码。

事实上，IPython集成了交互式Python的诸多优点。IPython具有卓越的Python shell，其性能远远优于标准Python的shell。IPython同时提供了基于控制台命令环境的定制功能，可以十分轻松地将交互式Python shell包含在各种Python应用中，甚至可以当作系统级shell来使用。本章主要介绍如何使用IPython提高\*nix-shell以及Python相关任务的执行效率。

与Python相同，IPython也有着非常活跃的社区支持。可以在 <http://lists.ipython.scipy.org/mailman/listinfo/ipython-user> 注册邮件列表；此外，在 <http://ipython.scipy.org/moin> 有一个极好的wiki。作为wiki的一部分，在 <http://ipython.scipy.org/moin/Cookbook> 还有一个菜单列表。因此，可以根据需要进行阅读或向其中添加资源。也可以在IPython的开发领域贡献力量。最近，IPython的开发已经转变为分布式代码控制方式，因此可以将代码分段下载后再进行处理。如果做了一些有益的工作，还可以向其提交你所做的修改。

## 名人简介：IPython

### Fernando Perez



Fernando Perez在获得物理学博士学位之后，在科罗拉多大学的应用数学系从事数值算法研究。目前，他在加州大学伯克利分校Helen Wills神经科学研究所，主要致力于脑成像问题的分析方法和高级科学计算工具方面的研究。在研究生期间，Fernando Perez就参与了Python工具的开发工作，这些工具都被用于科学计算领域。

2001年，为了寻找能够更为高效地处理每天科研任务的交互式工作流程，Fernando Perez发起了IPython开源项目。该项目得到了社区众多参与者的关注，支持者队伍日渐壮大。经过数年发展，IPython已经不仅局限于科研领域的应用，而且也让并非从事科研工作的程序员受益匪浅。

## 名人简介：IPython

### Ville Vainio



2003年，Ville Vainio在芬兰Satakunta应用科学大学（Satakunta University of Applied Sciences）Pori技术学院获得了软件工程学士学位。在撰写本书时，他是Digia Plc公司智能手机部的软件专家，主要在诺基亚和UIQ的Symbian操作系统平台上从事C++程序开发工作。此前，Ville Vainio曾就职于Cimcorp Oy公司，致力于使用Python语言开发工业机器人通信软件。Ville一直热衷于IPython，自2006年1月起，他就一直在维护0.x系列的稳定版本。Ville最初所做的工作是为IPython实现一系列补丁程序，使其具有比Windows系统shell更优越的性能。至今系统shell用例仍然是Ville关注的重点。Ville和未婚妻现在住在芬兰，在Pori的坦佩雷理工大学完成硕士论文。他的论文是关于ILeo项目的，ILeo试图在IPython和Leo之间架设一座桥梁，使Leo能够成为IPython的full-fledged记事本。

## 安装IPython

安装IPython可以有几种选择，其中最常见也是最常用的方法，是通过IPython发布的源码进行安装。IPython的源码可以在<http://ipython.scipy.org/dist/>下载。编写本书时，IPython的最新发布版本是0.8.2。0.8.3版本也即将完成。安装时需要下载tar.gz文件，例如<http://ipython.scipy.org/dist/ipython-0.8.2.tar.gz>。通过tar zxvf ipython-0.8.2.tar.gz命令解压软件包后，能够看到一个setup.py文件。通过调用带install参数的setup.py文件

(例如, `python setup.py install`) 开始安装Python。该操作将在`site-packages`目录中安装IPython的库文件，并在`scripts`目录中创建一个`ipython`脚本。在UNIX系统中，该目录与`python`的二进制文件目录相同。如果系统中已经安装了`python`包，则IPython将会安装到`/usr/bin`目录下。本书中，我们安装的是IPython最新的开发版源码，因此你可能会在一些例子中看到“0.8.3”。

第二种选择是通过系统的软件包管理器安装IPython软件包。`.deb`安装包可在Debian和Ubuntu获取，运行`apt-get install ipython`命令即可。Ubuntu将IPython的库文件安装到`/usr/share/python-support/ipython`目录下，包括一系列`.pth`文件和符号链接。而IPython的二进制文件则安装在`/usr/bin/ipython`目录下。

第三种选择是通过Python包进行安装。也许你从没有注意到在Python包中包含了IPython。实际上，Python包是一个ZIP文件，解压后包含一个扩展名为`.egg`的文件。Egg文件可以通过`easy_install`工具安装。`easy_install`工具的突出特点之一，是能够检查`egg`文件的配置，从而选择需要安装的内容。大多数时候，`easy_install`工具被人们忽略了，而事实上，它非常简单易用。`easy_install`工具通过Python包索引（Python Package Index，简称PyPI，又被称作Python CheeseShop）确定包的安装。使用`easy_install`工具安装IPython，只需要用户对`site-package`目录具有写权限，直接运行`easy_install ipython`即可。

第四种选择可能会令你感到万分惊讶，那就是IPython不必安装即可使用。当下载了IPython发布的源码，并运行了`ipython.py`安装命令之后，就可以使用该下载版本中的IPython实例了。这种方法能够使`site-packages`目录保持简明，但同时也会带来一些问题，那就是如果没有解压IPython，也没有修改`PYTHONPATH`环境变量，IPython将不能作为一个库文件直接使用。

## 基础知识

IPython安装之后，第一次运行`ipython`命令，将看到如下内容：

```
jmjones@dink:~$ ipython
*****
Welcome to IPython. I will try to create a personal configuration directory
where you can customize many aspects of IPython's functionality in:
/home/jmjones/.ipython

Successful installation!

Please read the sections 'Initial Configuration' and 'Quick Tips' in the
IPython manual (there are both HTML and PDF versions supplied with the
distribution) to make sure that your system environment is properly configured
to take advantage of IPython's features.
```

Important note: the configuration system has changed! The old system is still in place, but its setting may be partly overridden by the settings in "`~/.ipython/ipy_user_conf.py`" config file. Please take a look at the file if some of the new settings bother you.

Please press <RETURN> to start IPython.

此时光标会停留在原处，等待输入。若点击Return键，IPython将显示如下内容：

```
jmjones@dinkgutsy:stable-dev$ python ipython.py
Python 2.5.1 (r251:54863, Mar 7 2008, 03:39:23)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.3.bzr.r96 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

## 与IPython进行交互

当我们第一次接触到一个新的shell提示符，通常有些不知所措，甚至根本就不知道该做些什么。还记得第一次登录到UNIX，看到出现的(`b|k|c|z`)`sh`提示符时的情形吗？既然正在阅读本书，我们假设你已经对Unix shell有一定的了解。如果的确如此，那么对你而言，掌握IPython将变得十分容易。

当你第一次看到IPython提示符，也许同样不知道该做些什么。出现这种情况的原因极有可能是在IPython提示符下，可以做到的事几乎没有任何限制。因此，关键是应该明确想要做些什么。通过IPython提示符，Python语言所有的一切都可以使用。而且，还有许多IPython魔术般神奇的函数可以利用。通过IPython，可以方便地执行任何Unix shell命令，并将执行结果保存到Python变量中。接下来的几个示例，将展示在IPython的默认配置下，可以从IPython中获得什么。

下面是一些简单的输入输出操作：

```
In [1]: a = 1
In [2]: b = 2
In [3]: c = 3
```

这与在标准Python提示符下输入相同的内容看起来没什么不同。我们简单地为a、b、c分别赋值1、2、3。

IPython与标准Python的最大区别在于，Ipython会对命令提示符的每一行进行编号。

现在我们已经将一些数值（1、2和3）分别保存到一些变量中（变量a、b和c），可以查看这些变量所保存的数值：

In [4]: print a

1

In [5]: print b

2

In [6]: print c

3

这是一个设计好的例子。仅需要简单地键入打印输出变量语句（print），就可以直接查看每个变量的赋值，最坏的情况不过是需要向上回滚屏幕。每一个显示的变量会占6个字符，多于显示它们的值所需要的长度。下面是显示变量值的另一种更为简洁的方式：

In [7]: a  
Out[7]: 1

In [8]: b  
Out[8]: 2

In [9]: c  
Out[9]: 3

对比两个例子，其输出变量值似乎相同，其实仍有差别。print语句使用非正式的（unofficial）字符串表达式，而简单变量名（bare variable name）使用了正式的（official）字符串表达式。在处理自定义类而不是内置类时，这种差异会体现得非常明显。下面是使用不同字符串表示方法的示例：

In [10]: class DoubleRep(object):  
....: def \_\_str\_\_(self):  
....: return "Hi, I'm a \_\_str\_\_"  
....: def \_\_repr\_\_(self):  
....: return "Hi, I'm a \_\_repr\_\_"  
....:  
....:

In [11]: dr = DoubleRep()

In [12]: print dr  
Hi, I'm a \_\_str\_\_

In [13]: dr  
Out[13]: Hi, I'm a \_\_repr\_\_

我们创建了一个名为DoubleRep的类，类中有两个方法，一个是\_\_str\_\_，另一个是

`__repr__`, 用来演示使用`print`输出对象与使用正式字符串表达式的差异。在实例化`DoubleRep`对象后, 指定变量`dr`保存该对象。接下来, 使用`print`输出对象`dr`的值, 可以看到`__str__`方法被调用。之后, 简单地输入变量的名称`dr`, 则`__repr__`方法被调用。产生其中差异的原因是, 当输入变量名时, IPython以正式字符串表达式显示结果; 而使用`print`输出变量时, IPython采用非正式字符串表达式。总之, 在Python中, 当调用`str(obj)`或是使用格式化字符串`"%s" % obj`时, `__str__`方法被调用, 当调用`repr(obj)`或是使用格式化字符串`"%r" % obj`时, `__repr__`方法被调用。

实际上, 这并非IPython的特例, 标准Python shell也是如此。下面是使用标准Python shell的相同的`DoubleRep`示例:

```
>>> class DoubleRep(object):
...     def __str__(self):
...         return "Hi, I'm a __str__"
...     def __repr__(self):
...         return "Hi, I'm a __repr__"
...
>>>
>>> dr = DoubleRep()
>>> print dr
Hi, I'm a __str__
>>> dr
Hi, I'm a __repr__
```

你可能已经注意到了, 标准Python的提示符和IPython提示符是不一样的。标准Python的提示符由三个大于号 (`>>>`) 组成, 而IPython的提示符由单词 “In”, 之后是方括号中的数字, 最后是冒号组成 (例如`In [1]:`)。采用这样的提示符, 原因可能是IPython具有命令跟踪功能, 所有输入的命令都被保存在一个名为`In`的列表中。在前一个示例中, 当变量`a`、`b`、`c`赋值为1、2、3之后, `In`列表中的内容如下所示:

```
In [4]: print In
['\n', u'a = 1\n', u'b = 2\n', u'c = 3\n', u'print In\n']
```

IPython的输出提示符与标准Python输出提示符也是不同的。IPython的输出提示符看上去能够区分两种输出: 写输出 (written output) 与求值输出 (evaluated output)。而事实上, IPython并非真正能够区分这两种类型。`print`调用会引起计算的副作用, 因此IPython忽略`print`, 不会对其进行捕获。`print`的副作用会在标准输出`stdout`中反映出来, 这是在调用过程中发送的。而在IPython执行用户代码时, 会检测返回值, 如果返回值不是空 (`None`), 会在提示符 “Out [number]:” 后将返回值打印输出。

标准Python提示符同样不会区分这两种输出类型。如果在IPython提示符后输入了一个语句来对一些数值进行求值, 且值不为空, 则IPython将求得的值输出到新的一行中, 该行以`Out`开始, 后跟方括号和行号, 然后是冒号, 最后是表达式求值的结果 (例如

`Out[1]:1`。下面是一个示例，演示了在IPython中如何将一个整数赋值给变量，然后对变量进行求值运算，最后打印输出变量值。注意这三者之间的差异：为变量赋值、显示对变量求值的结果，以及打印输出变量。首先是IPython的提示符：

```
→ In [1]: a = 1  
In [2]: a  
Out[2]: 1  
  
In [3]: print a  
1  
  
In [4]:
```

接下来是标准Python的提示符：

```
→ >>> a = 1  
>>> a  
1  
>>> print a  
1  
>>>
```

在IPython和Python中，对整数进行赋值的方式没有什么差异。一个是IPython提示符，一个是标准Python提示符，两者都能够快速地向用户返回变量的赋值。但是，在显示正式字符串表示的变量时，IPython和标准Python有所不同。IPython显示一个Out提示符，而Python则直接显示值。对于打印输出变量，两者没有什么区别，都是以无提示符方式显示输出。

这种`In [some number]:`和`Out [some number]:`的方式或许会让人迷惑，是否在IPython与标准的Python之间存在着更深层的差异，还是这种差异纯粹就是表面上的？这种差异毫无疑问是有深层根源的。事实上，这种差异反映了IPython的功能区不同于Python，从而使得IPython这种不同类型的交互式shell与标准Python shell能够相互区分。

在这里，有两个内置变量应该引起注意，它们是`In`和`Out`。前者是IPython输入列表（list）对象，后者是一个字典（dict）对象。以下是`type`对`In`和`Out`的说明：

```
→ In [1]: type(In)  
Out[1]: <class 'IPython.iplib.InputList'>  
  
In [2]: type(Out)  
Out[2]: <type 'dict'>
```

在开始使用`In`和`Out`之后，你会对此习以为常。

接下来，我们看看这两个数据类型保存了什么内容。

```
→ In [3]: print In
['\n', u'type(In)\n', u'type(Out)\n', u'print In\n']

In [4]: print Out
{1: <class 'IPython.lib.InputList'>, 2: <type 'dict'>}
```

正如所期望的，In和Out分别保存了输入，以及非空语句和表达式求值运算的输出。由于每一行必须有输入，这对于跟踪类列表结构的输入非常有效。但是，跟踪类列表（list-like）结构的输出却可能导致一些空字段或所包含的内容为空的字段。因此，并不是每一行都会有可求值的非空输出，采用类字典（dictionary-like）的数据结构或是一个纯粹的字典（dict）对象对输出进行跟踪就显得十分有意义了。

## Tab自动完成

另外一个极为有用的是有关IPython数据输入（data-entry）的特征是Tab自动完成功能，该功能在默认状态下是开启的。标准Python shell如果编译时增加了readline支持特性，将具有tab自动完成功能，但需要做如下处理：

```
→ >>> import rlcompleter, readline
>>> readline.parse_and_bind('tab: complete')
```

经过上述设置，我们可以使用如下功能：

```
→ >>> import os
>>> os.lis<TAB>
>>> os.listdir
>>> os.li<TAB><TAB>
os.linesep os.link os.listdir
```

在加载了rlcompleter和readline，并设置了readline的Tab自动完成选项后，可以载入os，输入os.lis，按Tab键一次，让自动完成功能将其匹配成os.listdir。也可以输入os.li，然后按Tab键两次，将会出现一个所有可能匹配的列表。

在IPython中，可以实现相同的功能而无须进行任何额外的配置。对于Python，该项功能是可选的，而对于IPython，该功能则为默认开启的。下面是在IPython中运行与之前相同的示例：

```
→ In [1]: import os
In [2]: os.lis<TAB>
In [2]: os.listdir
In [2]: os.li<TAB>
os.linesep os.link os.listdir
```

注意，在本示例的最后一行，只需按Tab键一次即可。

这个os.TAB示例仅仅演示了IPython的属性查找和自动完成功能，而另一个更不错的自动完成功能，则体现在模块导入方面。打开一个新的IPython shell，这样可以看到IPython将如何帮助我们找到需要载入的模块：

```
In [1]: import os  
opcode operator optparse os os2emxpath ossaudiodev  
  
In [1]: import xm  
xml xmlllib xmllib
```

注意：所有通过import自动完成功能所列出的项都是模块，不需要为此感到意外，这就是IPython的特点。

IPython提供了两类自动完成功能：完成（complete）与菜单完成（menu-complete）。两者的差别在于“完成”尽可能扩展当前的主题词，并提供一个可能的替换列表，而“菜单完成”会扩展主题词，直接匹配可以替换列表中的一个，并且如果连续按Tab键时，每一次都会切换到下一个可能的替换。IPython的默认自动完成是“完成”。也可以通过简单的设置，轻松修改IPython的原有配置。

## 魔力编辑

最后一个将要涉及的基本输入输出主题是关于魔力编辑（magic edit）功能的。（在接下来的章节中将进一步介绍魔力编辑）。严格地说，使用shell这种面向命令行（line-oriented）的用户交互方式，尽管具有相当大的优势，但是也存在一定限制。这句话听起来有些矛盾，接下来我们一点一点分析解释。在shell中，采用一次输入一行命令的方式。每次输入一行命令，接下来shell会对命令进行处理，有时候你会坐下来等待命令执行的返回结果，然后再输入下一条命令。这个过程就是一个循环。事实上，也是非常有效的。但是，有时候如果一次能够处理多行命令，那将是非常不错的选择。为实现这一功能，使用文本编辑器对多行命令进行编辑是不错的做法。虽然IPython的readline支持在这方面有所改进，如可以使用文本编辑器编写Python模块，但这不是我们想在这里要谈论的内容。我们将要讨论的是在面向命令行的输入方式与文本编辑器输入方式之间进行整合，然后向shell提供需要执行的命令。可以说，有了对多行代码处理的支持，严格面向命令行的输入方式就显得功能有些受限了。这就是为什么我们说面向命令行的方式尽管具有相当大的优势，但是也存在一定限制的原因。

魔力编辑功能类似于上面提到的在Python shell的纯命令行交互方式与使用文本编辑器方式之间的折中。其好处是可以利用手边的资源，尽情享用你选择的编辑器的全部优点。可以简便地编辑代码块，对代码块中的循环或是函数的方法进行修改。而且，还具有与shell直接交互所带来的便捷和灵敏。当这两种来编写代码的方法可以整合时，其各自的

优点也被整合了。你可以保留shell环境，在编辑器中暂停、编辑和执行代码。当你继续使用熟悉的shell工作时，可以看到在编辑器中刚刚做的修改所带来的变化。

## 配置IPython

最后需要学习的基础知识是如何配置IPython。如果第一次运行IPython时没有指定其他信息，它会在home目录下创建一个*.ipython*目录。在*.ipython*目录中的是一个名为*ipy\_user\_conf.py*的文件。这就是使用Python语法的简单用户配置文件。为了让你可以随心所欲地使用IPython，配置文件中包含了大量的配置项，用户可以自行定制。例如，可以选择所使用shell的颜色，选择用于shell提示的组件，设置用%edit编辑文件时所使用的默认文件编辑器。这里，我们不再进一步描述其中的细节了。你只需要知道一点，那就是IPython有一个配置文件，并且它值得你仔细阅读一下，因为这有利于确定它是否包含了你所需要的配置项或希望设置的配置项。

## 从功能强大的函数获得帮助

正如已经提到的，IPython有着强大的功能。原因之一是它具有非常多的、内建的(built-in)魔力函数。什么是魔力函数？在IPython的文档中是这样描述的：

IPython会将任何第一个字母为%的行，视为对魔力函数的特殊调用。这样你就可以控制IPython，为其增加许多系统级的特征。魔力函数都是以%为前缀，并且参数中不包含括号或者引号。

例如：输入“%cd mydir”（不包括引号），表示如果mydir存在的话，将当前工作目录修改到mydir。

有两个魔力函数可以帮助你查看所有的函数，并且排序输出那些有用的函数。第一个魔力函数是lsmagic。lsmagic可以列出所有的魔力函数。下面是运行lsmagic的结果示例：

```
In [1]: lsmagic
Available magic functions:
%Exit %Pprint %Quit %alias %autocall %autoindent %automagic %bg
%bookmark %cd %clear %color_info %colors %cpaste %debug %dhist %dirs
%doctest_mode %ed %edit %env %exit %hist %history %logoff %logon
%logstart %logstate %logstop %lsmagic %macro %magic %p %page %pdb
%pdef %pdoc %pfile %pinfo %popd %profile %prun %psearch %psource
%pushd %pwd %pycat %quickref %quit %r %rehash %rehashx %rep %reset
%run %runlog %save %sc %store %sx %system_verbose %time %timeit
%unalias %upgrade %who %who_ls %whos %xmode

Automagic is ON, % prefix NOT needed for magic functions.
```

正如你所看到的，这里有非常多的函数可供使用。事实上，在写这本书时，已经有69个魔力函数可供使用。也可以像下面这样列出所有的魔力函数：

```
In [2]: %<TAB>
%Exit          %debug          %logstop        %psearch       %save
%Pprint         %dhist          %lsmagic         %psource       %sc
%Quit          %dirs           %macro          %pushd         %store
%alias         %doctest_mode   %magic          %pwd           %sx
%autocall      %ed             %p              %pycat         %system_verbose
%autoindent    %edit           %page           %quickref     %time
%automagic     %env            %pdb            %quit          %timeit
%bg             %exit           %pdef           %r              %unalias
%bookmark      %hist           %pdoc           %rehash        %upgrade
%cd             %history        %pfile          %rehashx      %who
%clear          %logoff          %pinfo          %rep           %who_ls
%color_info    %logon           %popd           %reset         %whos
%colors         %logstart        %profile        %run           %xmode
%cpaste         %logstate       %prun          %runlog
```

输入%然后按Tab键，可以看到69个magic函数的列表。使用lsmagic函数或输入%然后按Tab，都是为了快速查看所有可用的函数，当你正在寻找某一特定的函数时就可以这样做。或者你可以使用上述方法快速浏览所有的函数，看看都有哪些函数可用。但是除非看到具体的函数说明，否则仅靠列表，不足以帮助你了解每个函数的具体功能。

而这正是神奇的魔力函数发挥作用的另一个地方。魔力函数的名字magic本身就具有魔力。运行magic可以打开一个分页的帮助文档，其中记录了所有IPython内建函数的用法。这个帮助文档包括函数名，函数的用法（适用于何处），以及函数工作方式的描述。以下是魔力page函数的帮助说明：

```
%page:
Pretty print the object and display it through a pager.

%page [options] OBJECT

If no object is given, use _ (last output).

Options:

-i: page str(object), don't pretty-print it.
```

你可以在执行magic函数之后搜索或前后滚动，找到需要的内容。如果知道需要查找的具体函数，并且想直接跳到该函数所在位置而不是上下滚动进行查找，这么做非常有效。所有的函数按照字母顺序排列，因此无论是搜索还是滚动，都非常方便。

在本章的后面将介绍另外一种使用帮助的方法。在键入希望获得帮助信息的魔力函数名字之后，在其后输入问号（?），能够得到与使用%magic几乎相同的帮助信息。下面是使用%page?之后的结果：

```
In [1]: %page ?
Type:      Magic function
Base Class: <type 'instancemethod'>
```

```
String Form:    <bound method InteractiveShell.magic_page of
                 <IPython.iplib.InteractiveShell object at 0x2ac5429b8a10>>
Namespace:     IPython internal
File:          /home/jmjones/local/python/psa/lib/python2.5/site-packages/IPython/
                Magic.py
Definition:    %page(self, parameter_s='')
Docstring:
Pretty print the object and display it through a pager.

%page [options] OBJECT

If no object is given, use _ (last output).

Options:

-r: page str(object), don't pretty-print it.
```

这是IPython帮助文档中的一部分，用它非常适于生成一个总结，也可以用于生成魔力函数自身的总结。在IPython提示符下输入%quickref后，可以看到一个分页的参考文档信息，如下所示：

```
IPython -- An enhanced Interactive Python - Quick Reference Card
=====
obj?, obj??      : Get help, or more help for object (also works as
                    ?obj, ??obj).
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
%magic           : Information about IPython's 'magic' % functions.

Magic functions are prefixed by %, and typically take their arguments without
parentheses, quotes or even commas for convenience.

Example magic function calls:

%alias d ls -F   : 'd' is now an alias for 'ls -F'
alias d ls -F    : Works if 'alias' not a python name
alist = %alias   : Get list of aliases to 'alist'
cd /usr/share    : Obvious. cd -<tab> to choose from visited dirs.
%cd??            : See help AND source for magic %cd

System commands:

!cp a.txt b/      : System command escape, calls os.system()
cp a.txt b/      : after %rehashx, most system commands work without !
cp ${f}.txt $bar : Variable expansion in magics and system commands
files = !ls /usr  : Capture system command output
files.s, files.l, files.n: "a b c", ['a','b','c'], 'a\nb\nc'
```

其结束部分内容如下所示：

```
%time:
Time execution of a Python statement or expression.

%timeit:
Time execution of a Python statement or expression.
```

```
%unalias:  
    Remove an alias  
%upgrade:  
    Upgrade your IPython installation  
%who:  
    Print all interactive variables, with some minimal formatting.  
%who_ls:  
    Return a sorted list of all interactive variables.  
%whos:  
    Like %who, but gives some extra information about each variable.  
%xmode:  
    Switch modes for the exception handlers.
```

%quickref的起始部分是一个对IPython各种用法的引用。%quickref的其余部分是对%magic函数的迷你总结，包括全部帮助信息的首行。例如，下面是一个对%who的全部帮助信息：

```
In [1]: %who ?  
Type: Magic function  
Base Class: <type 'instancemethod'>  
String Form: <bound method InteractiveShell.magic_who of  
             <IPython.iplib.InteractiveShell object at 0x2ac9f449da10>>  
Namespace: IPython internal  
File: /home/jmjones/local/python/psa/lib/python2.5/site-packages/IPython/  
      Magic.py  
Definition: who(self, parameter_s='')  
Docstring:  
Print all interactive variables, with some minimal formatting.
```

If any arguments are given, only variables whose type matches one of these are printed. For example:

```
%who function str
```

will only list functions and strings, excluding all other types of variables. To find the proper type names, simply use type(var) at a command line to see how python prints type names. For example:

```
In [1]: type('hello')  
Out[1]: <type 'str'>
```

indicates that the type name for strings is 'str'.

%who always excludes executed names loaded through your configuration file and things which are internal to IPython.

This is deliberate, as typically you may load many modules and the purpose of %who is to show you only what you've manually defined.

%quickref中的%ho帮助行同使用标准的%who?返回的Docstring部分的第一行信息是一样的。

# UNIX Shell

使用UNIX shell，毫无疑问有它的优点。UNIX shell提供了一个处理问题的统一方法，具有丰富的工具集，相当简炼容易的语法、标准I/O流、管道、以及重定向等功能。如果能够在我们所熟悉的UNIX中增加Python的功能，将会非常有用。IPython就具有一些兼顾两者优点的特征。

## alias

起到衔接Python与UNIX shell功能的第一个特征是alias魔力函数。通过alias，可以创建一个IPython的快速方式，用以执行系统命令。定义别名，只需要简单地输入alias，后跟系统命令（也可以附加该命令的参数）。例如：

```
In [1]: alias nss netstat -lptn  
In [2]: nss  
(Not all processes could be identified, non-owned process info  
will not be shown, you would have to be root to see it all.)  
Active Internet connections (only servers)  
Proto Recv-Q Send-Q Local Address          Foreign Address      State  
tcp      0      0 0.0.0.0:80              0.0.0.0:*            LISTEN  
tcp      0      0 127.0.0.1:631           0.0.0.0:*            LISTEN
```

别名还有一些不同的输入方式，其中之一是do-nothing方法。如果传递给命令的所有附加参数都能够组织在一起，就可以采用do-nothing方法。例如，如果想查找netstat命令中有关80的执行结果，可以这样输入：

```
In [3]: nss | grep 80  
(Not all processes could be identified, non-owned process info  
will not be shown, you would have to be root to see it all.)  
tcp      0      0 0.0.0.0:80              0.0.0.0:*            LISTEN -
```

这样做没有传递附加的选项，但也无法显示参数是如何进行处理的。

还有一个do-everything方法。该方法与do-nothing方法相似，但采用的是隐含参数传递的方式。需要显示地操作所有后续参数。下面是一个示例，显示了如何以一个组的方式处理后续参数。

```
In [1]: alias achoo echo "%1"  
In [2]: achoo  
||  
In [3]: achoo these are args  
|these are args|
```

这里演示了%1（百分号后跟字母l）的语法，该方法用于将行的其他部分插入到alias中。

事实上，使用该方法几乎可以在别名之后（在别名所代表的实现命令的中间位置）插入任何内容。

下面是一个do-nothing示例，重新处理全部参数：

```
In [1]: alias nss netstat -lptn %l  
In [2]: nss  
(Not all processes could be identified, non-owned process info  
will not be shown, you would have to be root to see it all.)  
Active Internet connections (only servers)  
Proto Recv-Q Send-Q Local Address          Foreign Address      State  
tcp        0      0 0.0.0.0:80              0.0.0.0:*            LISTEN  
tcp        0      0 127.0.0.1:631           0.0.0.0:*            LISTEN  
  
In [3]: nss | grep 80  
(Not all processes could be identified, non-owned process info  
will not be shown, you would have to be root to see it all.)  
tcp        0      0 0.0.0.0:80              0.0.0.0:*            LISTEN
```

在这个示例中，事实上不需要加入%l。即使没有加入，也会得到相同的结果。

还可以通过命令字符串插入不同的参数，这里使用%s表示字符串。下面示例显示参数如何插入：

```
In [1]: alias achoo echo first: "|%s|", second: "|%s|"  
In [2]: achoo foo bar  
first: |foo|, second: |bar|
```

然而，这存在一点问题。如果仅提供了一个参数，而需要的是两个参数，则会得到错误信息。

```
In [3]: achoo foo  
ERROR: Alias <achoo> requires 2 arguments, 1 given.  
-----  
AttributeError                                Traceback (most recent call last)
```

而另一方面，如果提供的参数个数多于需要的个数，那么执行起来结果是正确的。

```
In [4]: achoo foo bar bam  
first: |foo|, second: |bar| bam
```

可以看到，foo和bar分别插入到各自的位置，而bam被附加到尾部，这正是它应当被放置的位置。

可以用%store魔力函数保留所使用的别名，本章后面会介绍如何实现。继续前面的例子，保留achoo别名，使得下一次打开IPython时，能够继续使用它，如下所示：

```
→ In [5]: store achoo
Alias stored: achoo (2, 'echo first: "|%s|", second: "|%s|")'

In [6]:
Do you really want to exit ([y]/n)?
(psa)jmjones@dinkgutsy:code$ ipython -nobanner

In [1]: achoo one two
first: |one|, second: |two|
```

## Shell的执行

另一个可以简易执行shell命令的方法，是在命令前加一个感叹号（!）：

```
→ In [1]: !netstat -lptn
(Not all processes could be identified, non-owned process info
will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address          Foreign Address      State
tcp        0      0 0.0.0.0:80              0.0.0.0:*            LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*            LISTEN
```

通过美元符 (\$) 前缀，可以将变量传递到shell命令中，例如：

```
→ In [1]: user = 'jmjones'

In [2]: process = 'bash'

In [3]: !ps aux | grep $user | grep $process
jmjones  5967  0.0  0.4  21368  4344 pts/0    Ss+  Apr11   0:01 bash
jmjones  6008  0.0  0.4  21340  4304 pts/1    Ss  Apr11   0:02 bash
jmjones  8298  0.0  0.4  21296  4280 pts/2    Ss+  Apr11   0:04 bash
jmjones 10184  0.0  0.5  22644  5608 pts/3    Ss+  Apr11   0:01 bash
jmjones 12035  0.0  0.4  21260  4168 pts/15   Ss  Apr15   0:00 bash
jmjones 12943  0.0  0.4  21288  4268 pts/5    Ss  Apr11   0:01 bash
jmjones 15720  0.0  0.4  21360  4268 pts/17   Ss  02:37   0:00 bash
jmjones 18589  0.1  0.4  21356  4260 pts/4    Ss+  07:04   0:00 bash
jmjones 18661  0.0  0.0    320     16 pts/15   R+  07:06   0:00 grep bash
jmjones 27705  0.0  0.4  21384  4312 pts/7    Ss+  Apr12   0:01 bash
jmjones 32010  0.0  0.4  21252  4172 pts/6    Ss+  Apr12   0:00 bash
```

可以看到，上例中列出了所有属于jmjones的bash会话。

下面是一个示例，展示了如何保存使用感叹号执行的命令结果：

```
→ In [4]: l = !ps aux | grep $user | grep $process

In [5]: l

Out[5]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: jmjones  5967  0.0  0.4  21368  4344 pts/0    Ss+  Apr11   0:01 bash
1: jmjones  6008  0.0  0.4  21340  4304 pts/1    Ss  Apr11   0:02 bash
2: jmjones  8298  0.0  0.4  21296  4280 pts/2    Ss+  Apr11   0:04 bash
```

```
3: jmjones 10184 0.0 0.5 22644 5608 pts/3 Ss+ Apr11 0:01 bash
4: jmjones 12035 0.0 0.4 21260 4168 pts/15 Ss Apr15 0:00 bash
5: jmjones 12943 0.0 0.4 21288 4268 pts/5 Ss Apr11 0:01 bash
6: jmjones 15720 0.0 0.4 21360 4268 pts/17 Ss 02:37 0:00 bash
7: jmjones 18589 0.0 0.4 21356 4260 pts/4 Ss+ 07:04 0:00 bash
8: jmjones 27705 0.0 0.4 21384 4312 pts/7 Ss+ Apr12 0:01 bash
9: jmjones 32010 0.0 0.4 21252 4172 pts/6 Ss+ Apr12 0:00 bash
```

你或许注意到了，输出结果保存到了变量`l`中，这与之前示例中的输出不同。变量`l`包括了一个类列表（list-like）对象，而之前演示的示例中显示的是命令的原始输出。在之后的字符串处理章节中将进一步讨论类列表对象。

`!!`可以替换`!`，除了使用`!!`无法保存结果到变量之外，两者完全一致。可以使用`_`或`_*[0-9]*`符号访问命令输出的结果，这将在之后的“历史结果”部分进行讨论。

在一个shell命令之前使用`!`或`!!`，无疑要比创建一个别名更为便捷。但是在使用过程中还需要视情况而定，一些情况下应该创建别名，而另一些情况下应该使用`!`或`!!`。例如，如果需要输入的命令时常用到，那么为其创建一个别名或宏比较好。如是仅仅使用一次，或偶尔用一下，那么最好使用`!`或`!!`。

## rehash

IPython中还有另一个使用别名或是执行shell命令的方法：重哈希（rehashing）。从技术上讲，它是为shell命令创建一个别名，但实际上并非如此。`rehash`魔力函数更新了PATH路径中的别名表（alias table）。你或许会问“什么是别名表”？当创建一个别名时，IPython映射别名到你希望关联的shell命令。别名表就是映射发生的地方。

---

**注意：**比重哈希别名表更好的方式是使用`rehashx`魔力函数而不是`rehash`。我们将同时说明这两种方法，并描述两者的不同。

---

IPython提供了大量运行IPython时可以访问的变量，例如`In`和`Out`，这两个变量我们已经介绍了。IPython还提供了一个变量`_IP`。`_IP`实际上是一个交互式shell对象，拥有一个叫做`alias_table`的属性。这就是实现映射别名到shell命令的地方。可以采用与查看任何变量相同的方式来查看别名映射：

```
In [1]: _IP.alias_table
Out[1]:
{'cat': (0, 'cat'),
'clear': (0, 'clear'),
'cp': (0, 'cp -i'),
'lc': (0, 'ls -F -o --color'),
'lmdir': (0, 'ls -F -o --color %l | grep /$'),
'less': (0, 'less')}
```

```
'lf': (0, 'ls -F -o --color %l | grep ^-'),
'lk': (0, 'ls -F -o --color %l | grep ^l'),
'll': (0, 'ls -lR'),
'lrt': (0, 'ls -lart'),
'ls': (0, 'ls -F'),
'lx': (0, 'ls -F -o --color %l | grep ^-..x'),
'mkdir': (0, 'mkdir'),
'mv': (0, 'mv -i'),
'rm': (0, 'rm -i'),
'rmdir': (0, 'rmdir')}
```

这看起来十分像字典：

```
In [2]: type(_IP.alias_table)
Out[2]: <type 'dict'>
```

的确如此。当前，该字典具有16个记录项：

```
In [3]: len(_IP.alias_table)
Out[3]: 16
```

在执行重哈希之后，映射变得更大：

```
In [4]: rehash
In [5]: len(_IP.alias_table)
Out[5]: 2314
```

让我们查找一些之前没有，而现在应该存在的记录项，例如**transcode**工具应该出现在别名表中：

```
In [6]: _IP.alias_table['transcode']
Out[6]: (0, 'transcode')
```

---

**注意：**当看到一个变量或属性以双下划线（`_`）开头时，这实际上表示代码的作者不希望你对其进行修改。我们可以访问变量`_IP`，但这只能展示它的内部结构。如果想去访问IPython的正式API，应该使用`_ip`对象，该对象在IPython提示符下可以被访问。

## rehashx

`rehashx`与`rehash`十分相似，只是`rehashx`在PATH中进行查找，并认为可以将其添加到别名表里。因此，当打开一个新的IPython shell并执行`rehashx`时，我们期望别名表与`rehash`的结果大小相同甚至更小。

```
In [1]: rehashx  
In [2]: len(__IP.alias_table)  
Out[2]: 2307
```

有趣的是，`rehashx`产生的别名表内容比`rehash`的或还要少，7个。以下是这7个不同之处：

```
In [3]: from sets import Set  
In [4]: rehashx_set = Set(__IP.alias_table.keys())  
In [5]: rehash  
In [6]: rehash_set = Set(__IP.alias_table.keys())  
In [7]: rehash_set - rehashx_set  
Out[7]: Set(['fusermount', 'rmmod.modutils', 'modprobe.modutils', 'kallsyms', 'ksyms', /'lsmod.modutils', 'X11'])
```

如果想看看为什么运行`rehashx`时，`rmmod.modutils`没有出现在别名表中，而在运行`rehash`时却确实出现了，可以执行如下操作：

```
jmjones@dinkgutsy:Music$ slocate rmmod.modutils  
/sbin/rmmod.modutils  
jmjones@dinkgutsy:Music$ ls -l /sbin/rmmod.modutils  
lrwxrwxrwx 1 root root 15 2007-12-07 10:34 /sbin/rmmod.modutils -> insmod.modutils  
jmjones@dinkgutsy:Music$ ls -l /sbin/insmod.modutils  
ls: /sbin/insmod.modutils: No such file or directory
```

可以看到，`rmmod.modutils`是`insmod.modutils`的链接，而`insmod.modutils`并不存在。

## cd

如果使用标准的Python shell，你可能注意到妄想判断当前进入的是哪一个目录非常困难。尽管可以使用`os.chdir()`来更改目录，但这并不十分方便。也可以使用`os.getcwd()`来获得当前的目录，但也不是非常方便。如果执行的是Python命令而不是标准Python shell下的shell命令，这或许不是一个大问题。但是，如果正在使用IPython并且需要经常访问系统shell，那么找到一种能够更为方便地访问目录并进行目录浏览的方法，将变得非常重要。

这就是将要介绍的`cd`。输入`cd`不是一个创举，做到这一点也毫无困难。但是想象一下，如果错过，问题将十分严重。

在IPython中，`cd`的作用与Bash中`cd`的作用相同。主要用法是“`cd directory_name`”。如果你有Bash的经验，就知道这样用是可以的。如果没有参数，`cd`命令会让你回到主目

录。如果使用空格加连字符作为参数 “cd -” , 能够让你回到前一个目录。以下三个附加的选项, 是Bash中的cd所不具备的。

第一个是 “-q” 或quiet选项。不使用该选项, IPython会输出你刚刚改变的目录名。下面是一个示例, 演示了使用了这一选项的不同:

```
In [1]: cd /tmp  
/tmp  
  
In [2]: pwd  
  
Out[2]: '/tmp'  
  
In [3]: cd -  
/home/jmjones  
  
In [4]: cd -q /tmp  
  
In [5]: pwd  
  
Out[5]: '/tmp'
```

使用-q会阻止IPython输出曾经进入过的/tmp目录。

另一个IPython的cd命令所包含的选项, 能够切换当前目录到已定义标签所在的目录。(之后会解释如何创建一个标签)。以下示例演示了如何改变目录到创建标签的位置:

```
In [1]: cd -b t  
(bookmark:t) -> /tmp  
/tmp
```

示例中, 假设对目录/tmp设置了标签t。切换到标签所示目录的正式语法是cd -b bookmark\_name。但是, 如果名为“bookmak\_name”的标签已被定义, 或是在当前目录下没有bookmak\_name目录, 则-b标志就是可选的; IPython能够找到你想要进入的用标签标识的目录。

IPython的cd命令所提供的最后一个特征选项, 是能够切换当前目录到指定的目录, 而指定的目录来自于由曾经访问过的目录组成的历史列表。下面是一个示例, 其中就使用了目录历史列表:

```
0: /home/jmjones  
1: /home/jmjones/local/Videos  
2: /home/jmjones/local/Music  
3: /home/jmjones/local/downloads  
4: /home/jmjones/local/Pictures  
5: /home/jmjones/local/Projects  
6: /home/jmjones/local/tmp  
7: /tmp  
8: /home/jmjones
```

```
In [2]: cd -6  
/home/jmjones/local/tmp
```

首先，你会看到在目录历史列表中列出的所有目录。使用该功能可以立即进入到以前访问过的目录。接下来，传递数字参数-6，该参数告诉IPython我们希望进入的目录是目录历史列表中标识为6的目录，即`/home/jmjones/local/tmp`。最后，我们看到当前目录已经成功切换到了`/home/jmjones/local/tmp`。

## bookmark

我们已经展示了如何使用cd选项进入到被标签（bookmark）标识的目录。现在介绍如何创建和管理标签。需要格外注意的是，标签在整个IPython会话过程中都是持久有效的。如果退出IPython，之后再次启动，你的标签仍将存在。有两种方式可以创建标签。下面是第一种方式：

► In [1]: cd /tmp  
/tmp

In [2]: bookmark t

在`/tmp`目录时，输入“`bookmark t`”，一个名为t的标签就创建了，且该标签指向`/tmp`目录。另一种创建标签的方法需要输入更多的参数，例如：

► In [3]: bookmark muzak /home/jmjones/local/Music

这里创建了一个名为muzak的标签，该标签指向一个本地存放音乐的目录。第一个参数是标签的名称，第二个参数是标签指向的目录名。

-l选项可以让IPython显示标签列表。我们已经定义了两个标签。现在看一下全部的标签：

► In [4]: bookmark -l  
Current bookmarks:  
muzak -> /home/jmjones/local/Music  
t -> /tmp

有两个选项可以删除标签：删除所有标签和一次删除一个标签。在以下示例中，我们创建一个标签，随后删除该标签，最后删除全部的标签，执行过程如下：

► In [5]: bookmark ulb /usr/local/bin

In [6]: bookmark -l  
Current bookmarks:  
muzak -> /home/jmjones/local/Music  
t -> /tmp  
ulb -> /usr/local/bin

```
In [7]: bookmark -d ulb  
In [8]: bookmark -l  
Current bookmarks:  
muzak -> /home/jmjones/local/Music  
t -> /tmp
```

-l还有一个可替换的选项-b，可以使用“cd -b”：

```
→ In [9]: cd -b<TAB>  
muzak t txt
```

继续后面的操作：

```
→ In [9]: bookmark -r  
In [10]: bookmark -l  
Current bookmarks:
```

可以看到，上述示例中创建的标签名为ulb，且指向/usr/local/bin目录。之后，使用选项“-d bookmark\_name”删除了该标签。最后使用-r选项删除了全部标签。

## dhist

在前面有关cd的示例中，用到了一个由曾经访问过的目录所组成的历史目录列表。现在向你演示如何查看该列表。使用的命令为dhist。该命令不仅可以保存会话列表，而且可以保存IPython会话过程中使用的目录。下面是不带参数使用dhist命令得到的结果：

```
→ In [1]: dhist  
Directory history (kept in _dh)  
0: /home/jmjones  
1: /home/jmjones/local/Videos  
2: /home/jmjones/local/Music  
3: /home/jmjones/local/downloads  
4: /home/jmjones/local/Pictures  
5: /home/jmjones/local/Projects  
6: /home/jmjones/local/tmp  
7: /tmp  
8: /home/jmjones  
9: /home/jmjones/local/tmp  
10: /tmp
```

一种访问目录历史的快捷方式是使用cd -<TAB>，如下所示：

```
→ In [1]: cd -  
-00 [/home/jmjones] -06 [/home/jmjones/local/tmp]  
-01 [/home/jmjones/local/Videos] -07 [/tmp]  
-02 [/home/jmjones/local/Music] -08 [/home/jmjones]  
-03 [/home/jmjones/local/downloads] -09 [/home/jmjones/local/tmp]  
-04 [/home/jmjones/local/Pictures] -10 [/tmp]  
-05 [/home/jmjones/local/Projects]
```

`dhist`命令有两个选项，可以让使用该命令比`cd-<TAB>`更为灵活。第一个选项允许提供一个数字来定义显示多少个目录。例如，只想查看最近访问过的5个目录，可以输入如下内容：

```
In [2]: dhist 5
Directory history (kept in _dh)
6: /home/jmjones/local/tmp
7: /tmp
8: /home/jmjones
9: /home/jmjones/local/tmp
10: /tmp
```

第二个选项允许指定一个目录范围。例如，要查看第3个到第6个之间的所有目录，可以输入如下命令：

```
In [3]: dhist 3 7
Directory history (kept in _dh)
3: /home/jmjones/local/downloads
4: /home/jmjones/local/Pictures
5: /home/jmjones/local/Projects
6: /home/jmjones/local/tmp
```

注意，结束边界是非包含的，因此必须在设置结束位置时，将其指定为你想要查看的最后一个目录的下一个目录。

## pwd

在目录操作中，一个简单但几乎是必须的函数就是`pwd`，`pwd`能够告诉你当前所在的目录。下面是一个示例：

```
In [1]: cd /tmp
/tmp
In [2]: pwd
Out[2]: '/tmp'
```

## 可变扩展

前面介绍了八个IPython的特征，它们是非常有用的，也是必需的。接下来将要介绍的三个特征会让高级用户感到非常高兴。其中，第一个是可变扩展（Variable Expansion）。到目前为止，我们几乎一直保持着shell是shell，Python是Python。但是现在，我们要跨越边界，将两者进行合并。也就是说，从Python取得一个值，然后把值传递给shell。

```
In [1]: for i in range(10):
...:     !date > ${i}.txt
...:
```

```
...:  
In [2]: ls  
0.txt 1.txt 2.txt 3.txt 4.txt 5.txt 6.txt 7.txt 8.txt 9.txt  
  
In [3]: !cat 0.txt  
Sat Mar 8 07:40:05 EST 2008
```

这个示例可能并不那么实用，因为很难有这样的需要：一下子创建10个文本文件，而且每个文本文件都包含日期。但这个示例却显示了如何将Python代码与shell代码结合。我们通过重复调用range()函数来创建一个列表，并且保存当前的项到变量i中。在每一次循环中，使用shell字符!来执行date命令。注意，这里调用date的语法，等同于已经定义了一个shell变量i，然后调用它。因此，date被调用，并且输出结果被重定向到文件{current list item}.txt中。在创建之后，我们使用ls命令列出了所有的文件，并且使用cat命令显示输出了其中一个文件。从文件的内容可以看到，这是一个日期。

可以将在Python中取得的任何值，传递到系统shell中。如果数值来自通过计算产生的数据库或是一个数据文件、一个XMLRPC服务，或者从文本文件中提取出的数据，可以先将其放到Python中，然后再使用!将其传递给系统shell。

## 字符串处理

IPython另一个强有力的特征是提供了采用字符串方式处理系统shell命令执行结果的功能。如果想查看属于用户jmjones的所有进程的PID值，可以通过输入以下命令实现：

```
ps aux | awk '{if ($1 == "jmjones") print $2}'
```

该命令十分紧凑、简练，而且可读性也很强。接下来，让我们看看如何使用IPython来处理相同任务。首先，提取非过滤命令ps aux的输出结果：

```
In [1]: ps = !ps aux
```

```
In [2]:
```

ps aux的执行结果是一种类列表的结构，保存在变量ps中，其数据项是从系统shell调用返回的结果。这里所说的类列表结构继承了内建的列表类型，所以能够支持这种类型的各种方法。因此，如果有一个函数或是方法的输入是一个列表，你可将这些结果对象传递给它。另外，除了能够对标准的列表方法提供支持外，ps也支持一些非常有趣的方法和方便使用的属性。为了能说明都有哪些有趣的方法，我们将偏离要找到全部属于jmjones所有的进程这个任务一小会儿。第一个十分有趣的方法是指grep()方法。这是一个基本的、非常简单的过滤器，可以决定输出中保留哪些行，删除哪些行。例如，要查看是否在输出中存在一些可以匹配lighttpd的行，可以输入下面的内容：

```
In [2]: ps.grep('lighttpd')
```

```
Out[2]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: www-data 4905 0.0 0.1.....0:00 /usr/sbin/lighttpd -f /etc/lighttpd/l
```

我们调用了`grep()`方法，并向其传递了一个正则表达式`lighttpd`。记住，传递给`grep()`的正则表达式是大小写敏感的。`grep()`的调用结果是一行输出，该行输出表示正则表达式“`lighttpd`”有一个正向匹配。我们可以像下面这样查看除匹配特定正则表达式外的所有记录：

► In [3]: `ps.grep('Mar07', prune=True)`

```
Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND  
1: jmjones 19301 0.0 0.4 21364 4272 pts/2 Ss+ 03:58 0:00 bash  
2: jmjones 21340 0.0 0.9 202484 10184 pts/3 Sl+ 07:00 0:06 vim ipytho  
3: jmjones 23024 0.0 1.1 81480 11600 pts/4 S+ 08:58 0:00 /home/jmjo  
4: jmjones 23025 0.0 0.0 0 0 pts/4 Z+ 08:59 0:00 [sh] <defu  
5: jmjones 23373 5.4 1.0 81160 11196 pts/0 R+ 09:20 0:00 /home/jmjo  
6: jmjones 23374 0.0 0.0 3908 532 pts/0 R+ 09:20 0:00 /bin/sh -c  
7: jmjones 23375 0.0 0.1 15024 1056 pts/0 R+ 09:20 0:00 ps aux
```

在将正则表达式“`Mar07`”传递给`grep()`方法后，可以看到大多数系统进程都是从`Mar07`（3月7日）开始运行的，因此我们决定查看所有不是在`Mar07`创建的进程。为了排除所有包含`Mar07`的记录项，传递了另一个参数给`grep()`，这个关键参数是：`prune=True`。这个关键参数告诉IPython“将匹配正则表达式的任何记录都删除掉”。正如你所看到的，输出结果中没有匹配`Mar07`的记录。

调用返回的结果也可以用于`grep()`。这表示`grep()`可以将函数作为一个参数来调用。它将函数传递给正在工作的列表项记录。如果函数返回值为真，则该项记录包括在过滤集中。例如，要创建一个目录列表，过滤掉文件或是目录：

► In [1]: `import os`

In [2]: `file_list = !ls`

In [3]: `file_list`

```
Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: ch01.xml  
1: code  
2: ipython.pdf  
3: ipython.xml
```

目录列表显示了四个文件。我们无法分辨列表中哪些是文件哪些是目录。但是如果使用`os.path.isfile()`进行过滤检测，就可以分辨出哪些是文件：

► In [4]: `file_list.grep(os.path.isfile)`

```
Out[4]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: ch01.xml
```

```
1: ipython.pdf  
2: ipython.xml
```

这次，名为*code*的文件被过滤掉了，因此*code*根本就不是文件。接下来对目录进行过滤：

```
In [5]: file_list.grep(os.path.isdir)  
Out[5]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: code
```

现在我们看到，*code*事实上就是一个目录。另一个有趣的方法是*fields()*。*fields()*可以在过滤完结果集并获得想要的数据之后（或许是之前），准确地列出希望显示的字段。看一下*non-Mar07*这个示例，在其中，我们输出了包含用户名、pid和登录时间的列：

```
In [4]: ps.grep('Mar07', prune=True).fields(0, 1, 8)  
Out[4]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: USER PID START  
1: jmjones 19301 03:58  
2: jmjones 21340 07:00  
3: jmjones 23024 08:58  
4: jmjones 23025 08:59  
5: jmjones 23373 09:20  
6: jmjones 23374 09:20  
7: jmjones 23375 09:20
```

首先需要注意的是，不管使用*fields()*方法做什么，都是使用该方法对*grep()*方法的执行结果进行处理。之所以可以如此操作，是因为*grep()*返回一个与*ps*对象相同类型的对象，而*fields*本身返回与*grep()*相同类型的对象。基于此，可以将*grep()*与*fields()*联合在一起进行调用。现在，让我们看看这是如何工作的。*fields()*方法使用了多个数字作为参数，这些参数是我们希望输出的列，而且输出行中各列应当是被空白字符分隔的。你可能会想到这非常像*awk*对文本进行处理时所采用的默认分隔。本例调用了*fields()*方法来查看第0、1、8列，也就是USERNAME、PID和STARTTIME字段。

现在，回到显示所有属于*jmjones*的进程的PID值这个示例：

```
In [5]: ps.fields(0, 1).grep('jmjones').fields(1)  
Out[5]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: 5385  
1: 5388  
2: 5423  
3: 5425  
4: 5429  
5: 5431  
6: 5437  
7: 5440
```

```
8: 5444  
<continues on...>
```

这个示例首先将结果集进行削减，只余下第0列和第1列，即用户名字段和PID字段。然后，用`grep()`方法从削减后的结果集中提取出包含`jmjones`的记录。最后通过调用`fields(1)`将过滤结果集中的第2个字段提取出来（注意，列的字段是从0开始编号的）。

关于字符串处理，我们最后想说一说能够直接访问进程列表的对象的`s`属性。这个对象或许不能给出需要寻找的结果。为了使系统shell与输出协同工作，可以在进程列表对象中使用`s`属性。

```
In [6]: ps.fields(0, 1).grep('jmjones').fields(1).s  
Out[6]: '5385 5388 5423 5425 5429 5431 5437 5440 5444 5452 5454 5457 5458 5468  
5470 5478 5480 5483 5489 5562 5568 5593 5595 5597 5598 5618 5621 5623 5628 5632  
5640 5740 5742 5808 5838 12707 12913 14391 14785 19301 21340 23024 23025 23373  
23374 23375'
```

`s`属性给了我们一个空格分隔的PID字符串，这些PID标明了在系统调用时可以使用的进程。我们希望能够将这个字符串字段列保存到一个名为`pids`的变量中，并且可以在IPython中执行类似`kill $pids`的操作。但是这会给用户`jmjones`的所有进程都发送一个SIGTERM信号，杀死它的文本编辑器和IPython会话。

在IPython脚本中使用下面的awk单行命令，就可以成功地实现之前所说的目标：

```
ps aux | awk '{if ($1 == "jmjones") print $2}'
```

介绍上述这些概念之后，我们同样可以通过一系列命令成功完成这一目标。`grep()`方法采用了称为字段（field）的选项参数。如果我们指定了字段参数，那么为了将所需要的内容收集到结果集中，搜索将按顺序匹配字段：

```
In [1]: ps = !ps aux  
In [2]: ps.grep('jmjones', field=0)  
Out[2]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: jmjones 5361 0.0 0.1 46412 1828 ? SL Apr11  
    0:00 /usr/bin/gnome-keyring-daemon -d  
1: jmjones 5364 0.0 1.4 214948 14552 ? Ssl Apr11  
    0:03 x-session-manager  
....  
53: jmjones 32425 0.0 0.0 3908 584 ? S Apr15  
    0:00 /bin/sh /usr/lib/firefox/run-mozilla.  
54: jmjones 32429 0.1 8.6 603780 88656 ? S1 Apr15  
    2:38 /usr/lib/firefox/firefox-bin
```

尽管这准确匹配了想要的行，但却输出了每行所有的列。为了取得指定PID值，应该作如下操作：

```
In [3]: ps.grep('jmjones', field=0).fields(1)  
Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: 5361  
1: 5364  
....  
53: 32425  
54: 32429
```

这样，就实现了与awk过滤器相同的目标。

## sh profile

一个我们没有讲到的概念是profile。一个profile是一个简单的配置集，在启动IPython时被加载。你可以自定义一些profile配置文件，让IPython可以根据会话的需要按不同的方式运行。要激活一个特定的profile配置文件，需要使用-p命令行选项并指定所使用的profile文件。

sh profile或者是shell profile是IPython内建的配置文件之一。sh profile可以使IPython在使用系统调用时更为友好。sh profile有两个配置项与标准IPython不同，sh不但显示当前目录而且rehash你的PATH，这样就可以与在Bash中一样，立刻访问所有的可执行程序。

除了设置一些配置值，sh profile也可以启动一些有助于shell的扩展。例如，启用环境持久性 (envpersist) 扩展。环境持久性扩展可以帮助你简单、持续地修改IPython sh profile中各种各样的环境变量，而无须升级.bash\_profile或.bashrc。

以下是PATH的内容：

```
jmjones@dinkgutsy:tmp$ ipython -p sh  
IPython 0.8.3.bzr.r96 [on Py 2.5.1]  
[~/tmp]|2> import os  
[~/tmp]|3> os.environ['PATH']  
<3> '/home/jmjones/local/python/psa/bin:  
/home/jmjones/apps/lb/bin:/home/jmjones/bin:  
/usr/local/sbin:/usr/local/bin:/usr/sbin:  
/usr/bin:/sbin:/bin:/usr/games'
```

现在添加:/appended到当前PATH变量的后面：

```
[~/tmp]|4> env PATH+=:/appended  
PATH after append = /home/jmjones/local/python/psa/bin:  
/home/jmjones/apps/lb/bin:/home/jmjones/bin:  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:  
/sbin:/bin:/usr/games:/appended
```

在当前PATH变量开始位置添加/prepended:

```
[~/tmp]|5> env PATH=/prepended:  
PATH after prepend = /prepended:/home/jmjones/local/python/psa/bin:  
/home/jmjones/apps/lb/bin:/home/jmjones/bin:/usr/local/sbin:  
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/usr/games:/appended
```

下面显示了使用os.environ的PATH环境变量:

```
[~/tmp]|6> os.environ['PATH']  
<6> '/prepended:/home/jmjones/local/python/psa/bin:  
/home/jmjones/apps/lb/bin:/home/jmjones/bin:  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:  
/bin:/usr/games:/appended'
```

现在退出IPython shell:

```
[~/tmp]|7>  
Do you really want to exit ([y]/n)?  
jmjones@dinkgutsy:tmp$
```

最后，打开一个新的IPython shell，查看PATH环境变量的值:

```
jmjones@dinkgutsy:tmp$ ipython -p sh  
IPython 0.8.3.bzr.r96 [on Py 2.5.1]  
[~/tmp]|2> import os  
[~/tmp]|3> os.environ['PATH']  
<3> '/prepended:/home/jmjones/local/python/psa/bin:  
/home/jmjones/apps/lb/bin:/home/jmjones/bin:/usr/local/sbin:  
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/usr/games:/appended'
```

有趣的是，尽管没有更新配置脚本，但在PATH之前和之后所添加的值都被显示了出来。IPython可以保证对PATH的修改即时生效，并且无须再做额外的工作。现在显示一下所有能够即时生效的环境变量。

```
[~/tmp]|4> env -p  
<4> {'add': [('PATH', ':/appended')], 'pre': [('PATH', '/prepended')], 'set': {}}
```

可以删除对PATH即时生效的设置:

```
[~/tmp]|5> env -d PATH  
Forgot 'PATH' (for next session)
```

可以检查PATH的值:

```
[~/tmp]|6> os.environ['PATH']  
<6> '/prepended:/home/jmjones/local/python/psa/bin:/home/jmjones/apps/lb/bin:  
/home/jmjones/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:  
/sbin:/bin:/usr/games:/appended'
```

可以看到，在告诉IPython删除对PATH的即时生效的设置之后，之前设置的PATH值仍然保留着。实际上，删除的设置已经起作用了，IPython将会删除对这些项的即时生效指令。注意，一些以某些环境变量开始的进程会保留这些值，直到发生了修改。也就是说，当下次IPython shell启动时，PATH的值就不一样了：

```
[~/tmp]|7> Do you really want to exit ([y]/n)?  
jmjones@dinkgutsy:tmp$ ipython -p sh  
IPython 0.8.3.bzr.r96 [on Py 2.5.1]  
[~/tmp]|2> import os  
[~/tmp]|3> os.environ['PATH']  
<3> '/home/jmjones/local/python/psa/bin:/home/jmjones/apps/lb/bin:  
/home/jmjones/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:  
/sbin:/bin:/usr/games'
```

正如所期望的，PATH的值已经恢复到开始修改之前的情况了。

另一个在sh profile中非常有用的特征是mglob。mglob的许多一般性设置语法非常简单。例如，要查找所有的Django目录中的.py文件，可以这样操作：

```
[django/trunk]|3> mglob rec:*py  
<3> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: ./setup.py  
1: ./examples/urls.py  
2: ./examples/manage.py  
3: ./examples/settings.py  
4: ./examples/views.py  
...  
1103: ./django/conf/project_template/urls.py  
1104: ./django/conf/project_template/manage.py  
1105: ./django/conf/project_template/settings.py  
1106: ./django/conf/project_template/__init__.py  
1107: ./docs/conf.py  
[django/trunk]|4>
```

rec指令简单地对它后面的模式执行递归查找。在本例中，\*py就是所谓的“模式”。为了显示Django目录中所有的目录，可以使用下面的命令：

```
[django/trunk]|3> mglob dir:  
<3> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: examples  
1: tests  
2: extras  
3: build  
4: django  
5: docs  
6: scripts  
</3>
```

`mglob`命令返回一个Python列表对象。因此，在Python中可以执行的操作，同样也可以在这个返回文件或目录的列表中执行。

我们以上所介绍的只是`sh profile`中的几个部分。`sh profile`还有一些特征和特征选项，但未能在这里作一一介绍。

## 信息搜集

IPython不仅是一个能够帮助你完成工作的shell，它也可以像工具一样，搜集各种类型的、与正在使用的代码和对象相关的信息。它可以执行信息挖掘，感觉就像是一个调查或侦测工具。本节将简要介绍IPython中能够帮助搜集信息的一些特性。

### page

如果正在处理的对象表示起来太过复杂，无法在一屏中完全显示，可以试试页（`page`）函数。`page`可以用来打印对象并且可以通过一个`pager`来运行。在许多系统中，默认的`pager`是`less`，但也可以使用其他的`pager`。标准用法如下：

```
In [1]: p = !ps aux
==
['USER      PID %CPU %MEM      VSZ      RSS TTY      STAT START      TIME COMMAND',
 'root      1  0.0  0.1      5116     1964 ?      Ss Mar07      0:00 /sbin/init',
 < ... trimmed result ... >
In [2]: page p
['USER      PID %CPU %MEM      VSZ      RSS TTY      STAT START      TIME COMMAND',
 'root      1  0.0  0.1 5116     1964      ? Ss      Mar07      0:00 /sbin/init',
 < ... trimmed result ... >]
```

这里，将系统shell命令`ps aux`的执行结果保存到变量`p`中。之后调用`page`，并且将处理结果对象传递给它。随后，`page`函数启动`less`。

`page`有一个选项`-r`。该选项告诉`page`不要美化打印（译注1）（pretty print）对象，而是通过`pager`运行它的字符串表示（`str()`的执行结果）。结果看起来类似这样：

```
In [3]: page -r p
ilus-cd-burner/mapping-d', 'jmjones 5568 0.0 1.0 232004 10608 ? S
Mar07 0:00 /usr/lib/gnome-applets/trashapplet --', 'jmjones 5593 0.0 0.9
188996 10076 ?      S      Mar07 0:00 /usr/lib/gnome-applets/battstat-apple',
'jmjones 5595 0.0 2.8 402148 29412 ?      S      Mar07 0:01 p
< ... trimmed result ... >
```

这个非美化打印（non-pretty-print）的结果确实是不够完美。我们建议还是从美化打印开始，并在此基础上进行工作。

译注1： 美化打印是指自动格式化输出产生统一的缩进格式。

## pdef

魔力pdef函数能打印输出任何可被调用对象的定义名或是函数声明。这个示例创建了一个函数，并且该函数有注释和返回语句：

```
In [1]: def myfunc(a, b, c, d):
...:     '''return something by using a, b, c, d to do something'''
...:     return a, b, c, d
...:

In [2]: pdef myfunc
myfunc(a, b, c, d)
```

pdef函数忽略了注释和返回语句，而输出了函数的声明部分。可以在任何可调用函数中这样使用。即使函数的源代码不可用，只要能够访问.pyc文件或egg文件，pdef函数就依然可以使用。

## pdoc

pdoc函数可以打印传递给它的函数的注释信息。这里使用pdoc处理在pdef示例中使用的myfunc()函数：

```
In [3]: pdoc myfunc
Class Docstring:
    return something by using a, b, c, d to do something
Calling Docstring:
    x.__call__(...) <==> x(...)
```

这是一个相当完美的自解释（self-explanatory）。

## pfile

pfile函数能够运行对象的文件，但前提是对象所包含的文件能够找得到。例如：

```
In [1]: import os
In [2]: pfile os
r"""OS routines for Mac, NT, or Posix depending on what system we're on.

This exports:
- all functions from posix, nt, os2, mac, or ce, e.g. unlink, stat, etc.

< ... trimmed result ... >
```

本例引入了os模块，并且通过less运行。这个示例能够帮助你理解一个代码段是如何开始运行的。显然，如果所包含的文件仅仅是egg或.pyc文件，pfile函数将不起作用。

注意：从??操作符可以看到与使用魔力函数%`pdef`、%`pdoc`和%`pfile`可以看到的相同的信息。优先选择的方法是??。

## pinfo

`pinfo`函数以及相关的工具使用起来非常方便。很难想象如果没有它们会怎么样。`pinfo`函数提供了诸如类型、基础类、命名空间和注释等信息。例如，有一个模块如下所示：

```
#!/usr/bin/env python

class Foo:
    """my Foo class"""
    def __init__(self):
        pass

class Bar:
    """my Bar class"""
    def __init__(self):
        pass

class Bam:
    """my Bam class"""
    def __init__(self):
        pass
```

我们可以从模块自身获得相应的信息：

```
In [1]: import some_module

In [2]: pinfo some_module
Type:          module
Base Class:    <type 'module'>
String Form:   <module 'some_module' from 'some_module.py'>
Namespace:     Interactive
File:          /home/jmjones/code/some_module.py
Docstring:
<no docstring>
```

也可以获得模块中所包含的类的相关信息：

```
In [3]: pinfo some_module.Foo
Type:          classobj
String Form:   some_module.Foo
Namespace:     Interactive
File:          /home/jmjones/code/some_module.py
Docstring:
    my Foo class

Constructor information:
Definition:    some_module.Foo(self)
```

此外，还可以获得类的实例的相关信息：

```
In [4]: f = some_module.Foo()

In [5]:      pinfo f
Type:           instance
Base Class:     some_module.Foo
String Form:    <some_module.Foo instance at 0x86e9e0>
Namespace:      Interactive
Docstring:
    my Foo class
```

在对象名之前或之后的问号 (?) 提供了与pinfo相同的功能：

```
In [6]: ? f
Type:           instance
Base Class:     some_module.Foo
String Form:    <some_module.Foo instance at 0x86e9e0>
Namespace:      Interactive
Docstring:
    my Foo class
```

```
In [7]: f ?
Type:           instance
Base Class:     some_module.Foo
String Form:    <some_module.Foo instance at 0x86e9e0>
Namespace:      Interactive
Docstring:
    my Foo class
```

但是，在对象名前面或后面的两个问号 (??) 可以提供更多的信息：

```
In [8]: some_module.Foo ??
Type:           classobj
String Form:    some_module.Foo
Namespace:      Interactive
File:          /home/jmjones/code/some_module.py
Source:
class Foo:
    """my Foo class"""
    def __init__(self):
        pass
Constructor information:
Definition:     some_module.Foo(self)
```

可以看到，??不但能够提供pinfo提供给我们的全部信息，而且还能够提供所请求对象的源代码。由于仅仅是对类进行查询，因此??提供的结果是类的源码，而不是整个文件。这正是pinfo函数的特点。也正因为如此，它比其他函数更为常用。

## psource

`psource`函数显示定义的元素的源代码，不论该元素是一个模块或是模块中的类或函数。为将其显示出来，`psource`通过运行`page`显示源代码。以下是一个`psource`针对模块的应用示例：

```
In [1]: import some_other_module

In [2]: psource some_other_module
#!/usr/bin/env python

class Foo:
    """my Foo class"""
    def __init__(self):
        pass

class Bar:
    """my Bar class"""
    def __init__(self):
        pass

class Bam:
    """my Bam class"""
    def __init__(self):
        pass

def baz():
    """my baz function"""
    return None
```

这是一个`psource`针对模块中的一个类的应用示例：

```
In [3]: psource some_other_module.Foo
class Foo:
    """my Foo class"""
    def __init__(self):
        pass
```

接下来是`psource`针对模块中的一个函数的应用示例：

```
In [4]: psource some_other_module.baz
def baz():
    """my baz function"""
    return None
```

## psearch

`psearch`魔力函数不但能够依据名称查找Python对象，还可以使用通配符协助查找。我们这里只简略地描述`psearch`函数，如果你想要知道更多的信息，可以通过在IPython提示符下输入`magic`来查看帮助文档。

让我们从声明下面的对象开始：

```
In [1]: a = 1
In [2]: aa = "one"
In [3]: b = 2
In [4]: bb = "two"
In [5]: c = 3
In [6]: cc = "three"
```

我们可以查找所有以a、b、c开头的对象，例如：

```
In [7]: psearch a*
a
aa
abs
all
any
apply

In [8]: psearch b*
b
basestring
bb
bool
buffer

In [9]: psearch c*
c
callable
cc
chr
classmethod
cmp
coerce
compile
complex
copyright
credits
```

注意，这里能够查到所有的对象，而不仅是a、aa、b、bb、c、cc，并且都是内建对象。

能够快速替换psearch函数的方法就是使用问号（?）操作符，下面是一个示例：

```
In [2]: import os
In [3]: psearch os.li*
os.linesep
os.link
os.listdir
```

```
In [4]: os.li*?  
os.linesep  
os.link  
os.listdir
```

除了`psearch`, 还可以使用`*?`。

`psearch`在执行搜索操作时可以使用`-s`选项, 排除搜索时可以使用`-e`选项, 搜索的范围是内建的命名空间。命名空间包括`builtin`、`user`、`user_global`、`internal`和`alias`。默认情况下, `psearch`搜索`builtin`和`user`空间。如果只是明确地对用户进行搜索, 传递`-e builtin`选项给`psearch`能够排除对内建空间的搜索。这似乎有点违反常规, 但非常有意义。`psearch`的默认搜索路径是`builtin`和`user`, 所以如果我们指定`-s user`, 搜索`builtin`和`user`仍会得到我们想要的结果。在这个示例中, 搜索又执行了一次。注意, 这些结果不包括内建的命名空间:

```
In [10]: psearch -e builtin a*  
a  
aa  
  
In [11]: psearch -e builtin b*  
b  
bb  
  
In [12]: psearch -e builtin c*  
c  
cc
```

`psearch`函数允许搜索指定类型的对象。这里搜索`user`命名空间中的整数类型对象。

```
In [13]: psearch -e builtin * int  
a  
b  
c
```

接下来是对字符串的搜索:

```
In [14]: psearch -e builtin * string  
—  
__name__  
aa  
bb  
cc
```

这里出现的`_`和`_objects`是IPython的缩略表示, 表示之前的返回结果。`__name__`是一个指定的变量, 表示模块的名称。如果`__name__`是“`__main__`”, 这表示模块是从解释器运行而不是从另外一个模块引入的。

## who

IPython还提供了一些能够列出所有交互式对象的方法。第一个就是who函数。下面是之前的一个示例，其中通过who函数显示变量a、aa、b、bb、c、cc：

```
In [15]: who
a      aa      b      bb      c      cc
```

who函数的使用非常直接明了，我们得到了一个返回的简单列表，包括了所有交互定义的对象。可以使用who函数对类型进行过滤，例如：

```
In [16]: who int
```

```
a      b      c
```

```
In [17]: who str
```

```
aa     bb     cc
```

## who\_ls

who\_ls函数与who函数十分相似，但who\_ls函数返回的是一个列表而不是所匹配变量的名称。下面是一个示例，演示了没有参数的who\_ls函数：

```
In [18]: who_ls
```

```
Out[18]: ['a', 'aa', 'b', 'bb', 'c', 'cc']
```

接下来是一个基于对象类型进行过滤的示例：

```
In [19]: who_ls int
```

```
Out[19]: ['a', 'b', 'c']
```

```
In [20]: who_ls str
```

```
Out[20]: ['aa', 'bb', 'cc']
```

由于who\_ls返回名称列表，可以使用\_variable访问名称列表，但这仅是针对“最后一次的输出”。下面是循环显示上次返回的匹配变量名称列表的示例：

```
In [21]: for n in _:
```

```
....:     print n
```

```
....:
```

```
....:
```

```
aa
```

```
bb
```

```
cc
```

## whos

`whos`函数与`who`函数非常相似，只是`whos`打印输出详细信息，而`who`不打印输出。下面是`whos`函数的一个示例，其中使用了非命令行参数：

```
In [22]: whos
Variable  Type  Data/Info
-----
a         int    1
aa        str    one
b         int    2
bb        str    two
c         int    3
cc        str    three
n         str    cc
```

接下来，正如在`who`示例中所做的那样，使用`whos`根据类型进行过滤：

```
In [23]: whos int
Variable  Type  Data/Info
-----
a         int    1
b         int    2
c         int    3

In [24]: whos str
Variable  Type  Data/Info
-----
aa        str    one
bb        str    two
cc        str    three
n         str    cc
```

## 历史

在IPython中，有两种方式可以访问输入的命令历史（History）。第一种是基于行（readline-based）的方式，第二种是基于`hist`函数的方式。

### 行支持（readline support）

到目前为止，你已经了解了IPython许多非常酷的特性，这些特性是在面向行的应用中常常会用得上的。如果习惯于使用Ctrl-s搜索Bash历史，在IPython中使用相同的功能时就不会有任何麻烦。下面，我们定义了一些变量，然后向后搜索整个历史。

```
In [1]: foo = 1
In [2]: bar = 2
In [3]: bam = 3
In [4]: d = dict(foo=foo, bar=bar, bam=bam)
```

```
In [5]: dict2 = dict(d=d, foo=foo)

In [6]: <CTRL-s>

(reverse-i-search)`fo': dict2 = dict(d=d, foo=foo)

<CTRL-r>

(reverse-i-search)`fo': d = dict(foo=foo, bar=bar, bam=bam)
```

首先输入Ctrl-r来启动搜索，然后输入fo作为搜索标准。它返回输入的行，如IPython中In[5]所示。使用行搜索功能，按Ctrl-r，它返回匹配输入的行，如IPython中In[4]所示。

可以通过行(`readline`)操作来完成更多的内容，但这里我们只能简单地做一个介绍。Ctrl-a让你回到行的开始位置，Ctrl-e让光标跳到行的结尾处。Ctrl-f用于删除字符，Ctrl-h能够向后删除一个字符(相当于`backspace`)。Ctrl-p将历史记录中的行向后移动一行，Ctrl-n则是向前移动一行。如果想了解更多的行操作，可以在\*nix系统中输入`man readline`进行查看。

### hist命令 (hist command)

除了具有访问行操作历史的各项功能外，IPython也提供了称为`history`的历史函数。可以使用`hist`代替`history`。如果不带参数，`hist`会连续打印用户输入命令的列表。默认情况下，该列表会被编号。例如设置一些变量，切换目录，然后运行`hist`命令：

```
→ In [1]: foo = 1
          In [2]: bar = 2
          In [3]: bam = 3
          In [4]: cd /tmp
                    /tmp
          In [5]: hist
          1: foo = 1
          2: bar = 2
          3: bam = 3
          4: _ip.magic("cd /tmp")
          5: _ip.magic("hist ")
```

在历史列表的第4和第5项是`magic`函数。注意，它们已经被IPython修改过。你可以看到如何通过Ipython调用`magic()`函数的过程。

如果希望去掉行号，可以使用`-n`选项。下面是`hist`命令使用`-n`选项的示例：

```
→ In [6]: hist -n
          foo = 1
          bar = 2
          bam = 3
```

```
_ip.magic("cd /tmp")
_ip.magic("hist ")
_ip.magic("hist -n")
```

如果在IPython中工作时想往文本编辑器中粘贴一段IPython的代码，这将非常有帮助。

-t选项返回一个被翻译的命令历史视图，历史命令记录了IPython看到的用户输入的命令。这是默认设置。在下面的示例中使用-t选项输出了到目前为止建立起来的命令历史：

```
In [7]: hist -t
1: foo = 1
2: bar = 2
3: bam = 3
4: _ip.magic("cd /tmp")
5: _ip.magic("hist ")
6: _ip.magic("hist -n")
7: _ip.magic("hist -t")
```

“raw history”或是选项-r能够准确显示输入了什么。下面的示例显示了在之前的示例中添加了“raw history”标志后的输出结果：

```
In [8]: hist -r
1: foo = 1
2: bar = 2
3: bam = 3
4: cd /tmp
5: hist
6: hist -n
7: hist -t
8: hist -r
```

IPython的-g标志也提供了一种搜索历史中指定模式的方式。下面的示例使用前面的示例配合-g标志对命令历史进行搜索：

```
In [9]: hist -g hist
0187: hist
0188: hist -n
0189: hist -g import
0190: hist -h
0191: hist -t
0192: hist -r
0193: hist -d
0213: hist -g foo
0219: hist -g hist
===
^shadow history ends, fetch by %rep <number> (must start with 0)
== start of normal history ==
5 : _ip.magic("hist ")
6 : _ip.magic("hist -n")
7 : _ip.magic("hist -t")
```

```
8 : _ip.magic("hist -r")
9 : _ip.magic("hist -g hist")
```

注意，之前的示例中返回了“shadow history”一词。shadow history是包括你输入的每一个命令的历史。“shadow history”从0开始显示在结果集的起始部分。来自会话的历史结果被保存在结果集的最后，但不以0开始。

## 历史结果 (History results)

在Python和IPython中，不仅可以访问曾经输入的命令历史列表，而且可以访问结果的历史。第一种实现此用途的方法是使用“\_”标志，这表示“上次输出”。下面的示例展示了\_函数在IPython中是如何工作的：

```
In [1]: foo = "foo_string"
In [2]: _
Out[2]: ''
In [3]: foo
Out[3]: 'foo_string'
In [4]: _
Out[4]: 'foo_string'
In [5]: a = _
In [6]: a
Out[6]: 'foo_string'
```

当我们在In[1]中定义了foo，在In[2]中的“\_”返回了一个空字符串。当我们在In[3]中输出了foo，便可以使用“\_”在In[4]中获得结果。在In[5]中，能够将结果保存到变量a中。

下面是使用标准Python shell运行相同示例的情形：

```
>>> foo = "foo_string"
>>> _
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_' is not defined
>>> foo
'foo_string'
>>> _
'foo_string'
>>> a = _
>>> a
'foo_string'
```

可以看到，除了在试图访问“\_”时输出了名字错误（NameError）异常，在IPython中的结果与标准Python shell的结果是非常相似的。

IPython在使用“上一次输出”的概念时更进一步：在本书“Shell的执行”部分中，我们描述了如何使用!和!!操作符，并且说明了不能将!!的结果保存到变量中，但是可以随后使用!!的结果。简言之，可以访问任何使用下划线之后跟随任一数字`_ [0-9]*`的语法方式输出的结果。数字必须与想看到的Out[0-9]\*结果相对应。

为了说明这一点，我们首先列出文件，但不对输出做任何处理：

```
In [1]: !!ls apa*py
Out[1]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: apache_conf_docroot_replace.py
1: apache_log_parser_regex.py
2: apache_log_parser_split.py

In [2]: !!ls e*py
Out[2]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: elementtree_system_profile.py
1: elementtree_tomcat_users.py

In [3]: !!ls t*py
Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: test_apache_log_parser_regex.py
1: test_apache_log_parser_split.py
```

首先使用`_1`, `_2`和`_3`访问了Out[1-3]。接下来，为每一项添加一个更为明确的名字：

```
In [4]: apache_list = _1
In [5]: element_tree_list = _2
In [6]: tests = _3
```

现在，`apache_list`, `element_tree_list`和`tests`包括了相同的元素，分别对应输出中的Out[1], Out [2]和Out [3]。

```
In [7]: apache_list
Out[7]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: apache_conf_docroot_replace.py
1: apache_log_parser_regex.py
2: apache_log_parser_split.py

In [8]: element_tree_list
Out[8]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: elementtree_system_profile.py
1: elementtree_tomcat_users.py
```

```
In [9]: tests  
Out[9]: SList(.p, .n, .l, .s, .grep(), .fields() available). Value:  
0: test_apache_log_parser_regex.py  
1: test_apache_log_parser_split.py
```

但是，所有这些的关键全部在于，在IPython中，可以使用`_加指定的变量`或是`使用_加数字`的方式，来访问之前输出的结果。

## 自动和快捷方式

即使IPython没能提高你的工作效率，它还是提供了一系列函数和特征来帮助你实现IPython任务和使用的自动化。

### alias

首先说一说alias（别名）。在这一章的前面已经对别名进行了介绍，因此这里不再老调重谈。但是在这里特别指出的是，别名不仅能帮助你直接在IPython中使用\*nix shell命令，而且可以帮助你将任务自动化。

### macro

macro（宏）函数允许你定义一个代码块，这段代码块可以在之后编写的程序中被内联（inline）执行，无论你正在编写什么代码都可以使用。这不同于创建函数或方法。宏，在某种意义上说，与当前代码运行的环境密切相关。如果有一个频繁在所有文件中执行的公共处理步骤，你就可以在文件中创建一个宏。为了更好地理解宏是如何在一系列文件中发挥作用的，请看下面的示例：

```
→ In [1]: dirlist = []  
In [2]: for f in dirlist:  
...:     print "working on", f  
...:     print "done with", f  
...:     print "moving %s to %s.done" % (f, f)  
...:     print "*" * 40  
...:  
...:  
In [3]: macro procdir 2  
Macro `procdir` created. To execute, type its name (without quotes).  
Macro contents:  
for f in dirlist:  
    print "working on", f  
    print "done with", f  
    print "moving %s to %s.done" % (f, f)  
    print "*" * 40
```

在In[2]中创建了一个循环，而在变量dirlist中没有循环需要的宏列表项，这是因为我们希望将来使用循环时再将循环列表项加入到变量dirlist中。我们创建了一个名为prokdir的宏来遍历列表。创建宏的语法是macro macro\_name range\_of\_lines。这里的range of lines是行的列表，来自想要合并到宏中的历史。宏列表的行应该用以空格为分隔的数字序列或表示数字序列的数字范围来标识（例如1-4）。

在这个示例中，我们创建一个系统文件名，并将其保存到dirlist变量中。通过执行prokdir宏，遍历dirlist中的所有文件名：

```
In [4]: dirlist = ['a.txt', 'b.txt', 'c.txt']

In [5]: prokdir
-----> prokdir()
working on a.txt
done with a.txt
moving a.txt to a.txt.done
*****
working on b.txt
done with b.txt
moving b.txt to b.txt.done
*****
working on c.txt
done with c.txt
moving c.txt to c.txt.done
*****
```

一旦定义了一个宏，就可以打开文本编辑器编辑它。在继续使用它之前，对其进行调试并确定它是否正确是非常必要的。

## store

通过store魔力函数可以一直使用你的宏和一些普通的Python变量。store函数的标准用法就是store variable。然而，store函数也可以包含一些非常有用的参数：-d variable函数能够从持久存储包（persistence store）中删除指定的变量；-z函数能够删除所有存储的变量；-r函数能够从持久存储包中重新加载所有变量。

## reset

reset函数用来从交互命名空间中删除所有变量。在下面的示例中，我们先定义了3个变量，再使用whos来检测它们的设置，之后reset命名空间，最后再次使用whos来验证它们已经被删除：

```
In [1]: a = 1

In [2]: b = 2
```

```
In [3]: c = 3
In [4]: whos
Variable Type Data/Info
-----
a int 1
b int 2
c int 3
In [5]: reset
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
In [6]: whos
Interactive namespace is empty.
```

## run

`run`函数可以在IPython中执行指定的文件。在其他应用中，它允许使用外部文本编辑器修改一个Python模块，并在Ipython中交互式地测试这些修改。在执行指定的程序之后，将会返回到IPython shell。使用`run`的语法是`run options specified_file args`。

-n选项使得模块的`_name_`变量设置成它自己的名称，而不是'`_main_`'。这使得模块运行与简单地载入十分相像。

-i选项在IPython的当前命名空间中运行模块，因此，使用运行的模块可以访问所有定义的变量。

-e选项使得IPython忽略对`sys.exit()`的调用和`SystemExit`异常。如果两者都没有发生，IPython继续执行。

-t选项使IPython输出模块运行的时间信息。

-d选项使得指定的模块运行在Python调试器（`pdb`）中。

-p选项在Python配置下运行指定的模块。

## save

`save`函数会保存指定的输入行到指定的输出文件中。使用`save`的语法为`save options filename lines`。行可以使用与宏相同的范围格式。`save`仅有的选项是-r。-r表示将原始的输入而不是经过转换的内容进行保存。在标准Python中，对输入进行转换是默认的。

## rep

`rep`是自动启用函数。`rep`函数有一些你或许觉得会非常有用的参数。使用不带参数的`rep`可以取回最近处理的结果，并在下一行输出时设置一个字符串进行表示。例如：

```
→ In [1]: def format_str(s):
...:     return "str(%s)" % s
...:

In [2]: format_str(1)

Out[2]: 'str(1)'

In [3]: rep

In [4]: str(1)
```

`rep`在In[3]被调用，这样你看到的文本被放到了In[4]中。这使得通过编程能够产生IPython需要处理的输入。尤其是当你混合使用generators和宏的时候，这非常方便。

普通的不带参数使用`rep`的方法是比较懒散、没有鼠标支持的编辑方式。如果你有一个包含一些值的变量，可以直接编辑这个值。作为一个示例，假设正在使用的函数对于指定的安装包返回到bin目录。我们将`bin`目录保存将在变量`a`中：

```
→ In [2]: a = some_blackbox_function('squiggly')

In [3]: a

Out[3]: '/opt/local/squiggly/bin'
```

如果输入`rep`，可以看到在新输入行中`/opt/local/squiggly/bin`之后出现闪烁的光标等待我们进行编辑：

```
→ In [4]: rep

In [5]: /opt/local/squiggly/bin<blinking cursor>
```

如果想去保存包的基目录，而不是`bin`目录，只需要删除路径最后的`bin`，在路径前加一个新变量名，之后是一个等号和前引号，最后加一个后引号即可：

```
→ In [5]: new_a = '/opt/local/squiggly'
```

现在已经有一个包含字符串的变量，该字符串是包的基目录名。

的确，我们可以用鼠标进行复制和粘贴，但是使用起来比较麻烦。为什么不使用舒适的键盘，而去使用鼠标？现在可以根据包来使用一个新的标识`_a`，作为任何需要进行操作的基目录。

当把一个数字作为参数传递给`rep`时，IPython获得从历史记录得来的指定行，并且放到下一行中，然后将光标在放置行的末尾。这对于执行、编辑和再执行单行甚至一小段代码来说非常有帮助。例如：

```
→ In [1]: map = (('a', '1'), ('b', '2'), ('c', '3'))
```

```
In [2]: for alph, num in map:  
...:     print alph, num  
...:  
...:  
a 1  
b 2  
c 3
```

这里，我们编辑In[2]，并打印输出数字值乘以2的结果，而不是一个非计算的值。我们既可以再次输入for循环，也可以使用rep：

► In [3]: rep 2

```
In [4]: for alph, num in map:  
print alph, int(num) * 2  
...:  
...:  
a 2  
b 4  
c 6
```

rep函数也可以使用数字范围作为参数。数字范围的语法与宏中的数字范围语法相同，这在本章的前面已经讨论过了。当你为rep指定一个范围时，行被立即执行。下面是rep的一个示例：

► In [1]: i = 1

In [2]: i += 1

In [3]: print i  
2

In [4]: rep 2-3  
lines [u'i += 1\nprint i\n']  
3

In [7]: rep 2-3  
lines [u'i += 1\nprint i\n']  
4

我们定义了一个递增计数器和在In[1]到In[3]之间打印输出当前计数值的代码，在In[4]和In[7]中告诉rep重复第2行和第3行。注意，由于第5和第6两行在In[4]之后执行，因此被错过了。

rep的最后一个选项是传递一个字符串。这与“将一个词传递给rep”或是“传递一个非引用搜索字符串到rep”非常相似。下面是一个示例：

► In [1]: a = 1

In [2]: b = 2

```
In [3]: c = 3
```

```
In [4]: rep a
```

```
In [5]: a = 1
```

我们定义了一些变量，并且告诉rep重复包含字母“a”的行。它将In[1]返回给我们来编辑和重新执行。

## 本章小结

IPython是工具包中最常使用的一个工具。掌握了这个神奇的shell，就相当于掌握了一个神奇的文本编辑器：你越精通它，就可以越快速地完成单调乏味的工作。我们在几年前开始使用IPython的时候，就发现了这个工具了不起的强大功能。现在，Ipython得到进一步发展，变得更为强大了。grep函数和对字符串的处理功能是学习IPython时首先需要了解的特性，也是IPython最重要的优点。我们强烈建议你进一步深入学习IPython，你绝不会因为花费时间学习IPython而感到后悔的。

## 第3章

# 文本

几乎所有的系统管理员都需要处理文本，无论其形式是日志、应用程序数据、XML、HTML、配置文件或是某些命令的输出结果。通常，你会使用诸如grep、awk这样的工具，但有时候可能需要一个更富表现力、更完美的工具来处理更为复杂的问题。在你需要利用从其他文件中提取的数据来创建新文件时，经常使用重定向文本处理工具（grep或awk）的输出到一个文件方法。实际上，一个易于扩展的工具可能更适合此项工作。

经验表明，相对Perl、bash或是其他语言而言，Python具有更富表现力，更完美，且易于扩展的特点。关于为什么我们对Python的评价要比Perl或Bash（你同样可以使用sed或awk创建应用程序）更高，请参见第1章的内容。Python的标准库，语言特征和内建类型，对于读取文本文件、操作文本或是从文本文件中提取信息而言，都是非常强大的。Python及其标准库包含了大量灵活、功能强大的函数，适用于利用字符串类型、文件类型和正则表达式模块进行文本处理。标准库中最近新增了一个功能，即ElementTree，在需要处理XML时，ElementTree非常有用。在这一章中，我们将学习如何有效地使用标准库和内建组件来实现文本处理。

## Python的内建功能及模块

### str

字符串由一系列字符组成。如果需要处理文本数据，则很可能需要使用一个字符串对象或是一系列字符串对象。字符串类型（str）是一个强大而灵活的、能够对字符串数据进行操作处理的数据类型。本节将演示如创建一个字符串，以及创建之后如何使用。

#### 创建字符串

最普通的创建字符串的方法是在文本前后加上引号：

```
In [1]: string1 = 'This is a string'  
In [2]: string2 = "This is another string"  
In [3]: string3 = '''This is still another string'''  
In [4]: string4 = """And one more string"""  
In [5]: type(string1), type(string2), type(string3), type(string4)  
Out[5]: (<type 'str'>, <type 'str'>, <type 'str'>, <type 'str'>)
```

单引号、双引号、三引号可以完成相同的事：去掉创建一个str类型的对象。单引号和双引号在创建字符串时是相同的，可以替换使用。这与UNIX shell中引号的使用略有不同。在UNIX中两者是不可以替换使用的，例如：

```
jmjones@dink:~$ FOO=sometext  
jmjones@dink:~$ echo "Here is $FOO"  
Here is sometext  
jmjones@dink:~$ echo 'Here is $FOO'  
Here is $FOO
```

Perl在创建字符串时也使用单、双引号。下面是一个Perl脚本写的对比示例：

```
#!/usr/bin/perl  
  
$FOO = "some_text";  
print "-- $FOO --\n";  
print '-- $FOO --\n';
```

下面是一个简单的Perl脚本的输出结果：

```
jmjones@dinkgutsy:code$ ./quotes.pl  
-- some_text --  
-- $FOO --\n jmjones@dinkgutsy:code$
```

在Python中则不存在这样的差别。Python将差别留给了编程人员进行处理。例如，可以在单引号的字符串中嵌入双引号，并且不使用反斜线（“\”）进行字符转义。相反地，如果需要在字符串中嵌入一个单引号，并且不想对其进行转义，则该字符串应使用双引号，如例3-1。

### 例3-1：Python中单/双引号的比较

```
In [1]: s = "This is a string with 'quotes' in it"  
  
In [2]: s  
Out[2]: "This is a string with 'quotes' in it"  
In [3]: s = 'This is a string with \'quotes\' in it'  
  
In [4]: s  
Out[4]: "This is a string with 'quotes' in it"  
  
In [5]: s = 'This is a string with "quotes" in it'
```

```
In [6]: s  
Out[6]: 'This is a string with "quotes" in it'  
  
In [7]: s = "This is a string with \"quotes\" in it"  
  
In [8]: s  
Out[8]: 'This is a string with "quotes" in it'
```

注意，在In [3]和In [7]中分别嵌入了一对相同类型的转义引号。

在希望一个字符串能够跨越多行时，可以在字符串中你希望分行的地方嵌入“\n”来解决，但是这种方法略显笨拙。另外一种更为简洁的替换方法是使用三引号。三引号允许创建多行字符串。例3-2中先演示了在多行字符串中单引号的使用，之后演示了三引号的使用：

### 例3-2：三引号

```
→ In [6]: s = 'this is  
-----  
File "<ipython console>", line 1  
s = 'this is  
^  
SyntaxError: EOL while scanning single-quoted string  
  
In [7]: s = '''this is a  
...: multiline string'''  
  
In [8]: s  
Out[8]: 'this is a\nmultiline string'
```

对于复杂的处理，在Python中有另一种被称为“原始”字符串的字符串表示方法。在创建一个字符串时，通过在引号之前放置字母r，可以创建一个原始字符串。从根本上讲，创建一个原始字符串与创建一个非原始字符串的区别在于，Python不对原始字符串中的转义字符进行解析，而在处理普通字符串时，则对其进行解析。Python遵循类似于标准C语言对转义序列的一系列规则。例如，在普通字符串中，“\t”被解析成tab字符，“\n”被解析成换行，“\r”被解析成回车。表3-1展示了Python中使用的转义序列。

表3-1：Python的转义序列

转义字符	解释
\	newline Ignored忽略换行符
\\\	Backslash反斜杠
\'	Single quote单引号
\"	Double quote双引号
\a	ASCII Bell响铃

表3-1：Python的转义序列（续）

转义字符	解析为
\b	ASCII backspace退格
\f	ASCII form feed表格换行
\n	ASCII line feed换行
\N{name}	Named character in Unicode database (Unicode strings only) Unicode数据库中命名的字符（仅Unicode字符串）
\r	ASCII carriage return回车
\t	ASCII horizontal tab水平制表符
\uxxxx	Character with 16-bit hex value xxxx (Unicode only) 16位十六进制值表示的字符（仅Unicode）
\Uxxxxxxxxx	Character with 32-bit hex value xxxx (Unicode only) 32位十六进制值表示的字符（仅Unicode）
\v	ASCII vertical tab垂直制表符
\ooo	Character with octal value oo八进制值表示的字符
\xhh	Character with hex value hh十六进制值表示的字符

转义序列的原始字符串方便记忆，尤其是在处理正则表达式的时候，这在本章的后面将进行介绍。例3-3显示了在原始字符串中转义序列是如何使用的。

### 例3-3：转义序列与原始字符串

```
In [1]: s = '\t'  
  
In [2]: s  
Out[2]: '\t'  
  
In [3]: print s  
  
In [4]: s = r'\t'  
  
In [5]: s  
Out[5]: '\\t'  
  
In [6]: print s  
\t  
  
In [7]: s = """\t"""  
  
In [8]: s  
Out[8]: '\t'  
  
In [9]: print s  
  
In [10]: s = r"""\""\t"""
```

```
In [11]: s
Out[11]: '\\t'

In [12]: print s
\t

In [13]: s = r'\t'

In [14]: s
Out[14]: "\t"

In [15]: print s
\t
```

当转义序列被解析时，“\t”表示tab。当转义序列不被解析时，“\t”只是简单的一个由两个字符“\”和“t”组成的字符串。对于使用单引号、双引号或三引号创建的字符串，可以将“\t”解析为tab字符。而同样的字符串如果以r开头，可以将“\t”解析为“\”和“t”两个字符。

从这个示例中可以看到的另外一个有趣的事情就是`_repr_`与`_str_`之间的差别。如果在IPython提示符下输入一个变量名，然后按回车键，它的`_repr_`表示将显示出来。如果输入`print`加变量名，再按回车后，它的`_str_`表示将被显示出来。`print`函数解析字符串中的转义序列，并合理显示其内容。如果想看到对`_repr_`和`_str_`更多的介绍，请查阅第2章中的相关内容。

## 对于str进行数据提取的内建方法

字符串是对象，提供了可以被调用以执行操作的方法。但是提及方法，不能仅限于`str`对象类型提供给我们使用的方法，还应包括所有可以用于从字符串对象中提取数据的可用的方法。这包括所有的`str`方法，也包括前一个示例中使用的`in`和`not in`文本操作。

从技术上讲，在例3-1中，`in`和`not in`调用了`str`对象的一个方法`_contains_()`。要了解更多的详细信息，请参见附录。可以使用`in`和`not in`来检查一个字符串是否是另一个字符串的一部分。参见例3-4。

### 例3-4：In和not in

```
In [1]: import subprocess

In [2]: res = subprocess.Popen(['uname', '-sv'], stdout=subprocess.PIPE)

In [3]: uname = res.stdout.read().strip()

In [4]: uname

Out[4]: 'Linux #1 SMP Tue Feb 12 02:46:46 UTC 2008'

In [5]: 'Linux' in uname

Out[5]: True
```

```
In [6]: 'Darwin' in uname  
Out[6]: False  
  
In [7]: 'Linux' not in uname  
Out[7]: False  
  
In [8]: 'Darwin' not in uname  
Out[8]: True
```

如果string2包含string1，则“string1 in string2”返回的值为真，否则返回的值为假。因此在检查“Linux”是否包含在uname所表示的字符串中时，返回值为真，而当检查“Darwin”是否包含在uname所表示的字符串中时，返回值为假。接下来我们演示了not in的有趣之处。

有时仅仅需要知道一个字符串是否是另一个字符串的子串。但有时还需要知道子字符串出现的具体位置。使用find()和index()可以实现这一目的。参见例3-5。

### 例3-5：find()和index()

```
→ In [9]: uname.index('Linux')  
Out[9]: 0  
  
In [10]: uname.find('Linux')  
Out[10]: 0  
  
In [11]: uname.index('Darwin')  
-----  
<type 'exceptions.ValueError'>           Traceback (most recent call last)  
/home/jmjones/code/<ipython console> in <module>()  
<type 'exceptions.ValueError'>: substring not found  
  
In [12]: uname.find('Darwin')  
Out[12]: -1
```

如果string1在string2中（正如先前看到的示例），string2.find(string1)将返回string1第一个字符的索引，否则，返回-1。（别着急，我们马上会介绍有关索引的内容。）如果string1包括在string2中，string2.index(string1)将返回string1的第一个字符的索引，否则它会抛出一个ValueError异常。在示例中，find()方法在字符串开始处找到字符串“Linux”，所示返回值为0，表示“Linux”的第一个字符的索引为0。然而，find()方法无法找到“Darwin”，因此返回值为-1。当Python搜索“Linux”时，index()方法与find()方法表现类似。但是，当查找“Darwin”时，index()抛出一个ValueError异常，表示它无法找到那个字符串。

如何处理索引号呢？使用它们有什么好处呢？字符串被理解为一系列字符。`find()`和`index()`返回的索引表明从较长字符串中的哪一个字符开始匹配子字符串。参见例3-6。

#### 例3-6：字符串切分

```
In [13]: smp_index = uname.index('SMP')

In [14]: smp_index

Out[14]: 9

In [15]: uname[smp_index:]

Out[15]: 'SMP Tue Feb 12 02:46:46 UTC 2008'

In [16]: uname[:smp_index]

Out[16]: 'Linux #1'

In [17]: uname

Out[17]: 'Linux #1 SMP Tue Feb 12 02:46:46 UTC 2008'
```

使用字符串切分语法“`string[index:]`”，可以查找到从SMP开始到字符串结尾的每一个字符。使用语法`string[:index]`，我们可以看到从uname字符串起始位置到找到SMP所在位置的所有字符。两者之间仅有的差别在于冒号（`:`）在索引的左边或是右边。

上述字符串切分示例，以及`in`与`not in`检测示例主要是为了向你展示字符串是一个序列，因此可以像处理列表这样的序列结构一样进行处理。如果希望查看更多对序列处理相关的讨论，参见由alex Martelli编著的《Python in a Nutshell (O'Reilly)》第4章序列操作，也可以在线访问<http://safari.oreilly.com/0596100469/pythonian-CHP-4-SECT-6>。

另外两个或许会用到的方法是`startswith()`和`endswith()`。正如方法名所表示的，这两个方法可以帮助你判断字符串是否以某一特定子串开始，或是以某一特定子串结束。参见例3-7：

#### 例3-7：startswith()和endswith()

```
In [1]: some_string = "Raymond Luxury-Yacht"

In [2]: some_string.startswith("Raymond")
Out[2]: True

In [3]: some_string.startswith("Throatwarbler")
Out[3]: False

In [4]: some_string.endswith("Luxury-Yacht")
Out[4]: True

In [5]: some_string.endswith("Mangrove")
Out[5]: False
```

可以看到Python返回的信息：字符串“Raymond Luxury-Yacht”以“Raymond”开始，以“Luxury-Yacht.”结束。而不是以“Throatwarbler,”开始，也不是以“Mangrove.”结束。如果使用上文介绍的切分方法，也可以简单地获得相同的结果，但是切分方法使用起来有一些麻烦和枯燥。参见例3-8。

#### 例3-8：使用切分技术实现Startswith()和endswith()功能

```
In [6]: some_string[:len("Raymond")] == "Raymond"
Out[6]: True
In [7]: some_string[:len("Throatwarbler")] == "Throatwarbler"
Out[7]: False
In [8]: some_string[-len("Luxury-Yacht"):] == "Luxury-Yacht"
Out[8]: True
In [9]: some_string[-len("Mangrove"):] == "Mangrove"
Out[9]: False
```

---

**注意：**切分操作可以创建并返回一个新的字符串对象，而不是在行内对字符串进行修改。脚本中切分一个字符串的频率，会对内存和性能有明显的影响。即使没有明显的性能影响，在使用startwith()及endswith()可以满足需要的情况下，应避免使用切分操作。

---

可以看到，通过切分，尽管其中有多个字符，字符串“Paymond”还是出现在了some\_string的开始。换句话说，我们可以看到some\_string以字符串“Raymond”开始，而没有使用startswith()方法。在结尾的“Luxury-Yacht.”也是如此。

如果不带任何参数，lstrip()、rstrip()和strip()分别是用来删除前导空白，结尾空白，和前后空白的方法。空白可以包括tab、空格、回车和换行。不带参数使用lstrip()可以删除在字符串开始处出现的空白，并把去除空白的字符串作为一个新的字符串返回。使用不带参数的rstrip()可以删除出现在字符串结尾的空白，并把去除空白的字符串作为一个新的字符串返回。不带参数使用strip()可以删除在字符串开始及结尾的所有空白，并返回一个新字符串。参见例3-9。

---

**注意：**所有的strip()方法创建并返回新的字符串对象，而不是对字符串进行行内修改。这或许根本不是什么问题，但是却是值得引起注意的地方。

---

#### 例3-9：lstrip()、rstrip() 和strip()

```
In [1]: spacious_string = "\n\t Some Non-Spacious Text\n \t\r"
In [2]: spacious_string
Out[2]: '\n\t Some Non-Spacious Text\n \t\r'
In [3]: print spacious_string
Some Non-Spacious Text
```

```
In [4]: spacious_string.lstrip()  
Out[4]: 'Some Non-Spacious Text\n\t\r'
```

```
In [5]: print spacious_string.lstrip()  
Some Non-Spacious Text
```

```
In [6]: spacious_string.rstrip()  
Out[6]: '\n\t Some Non-Spacious Text'
```

```
In [7]: print spacious_string.rstrip()  
Some Non-Spacious Text
```

```
In [8]: spacious_string.strip()  
Out[8]: 'Some Non-Spacious Text'
```

```
In [9]: print spacious_string.strip()  
Some Non-Spacious Text
```

但是，`strip()`、`rstrip()`以及`lstrip()`都有一个可选参数：待除去的字符所组成的字符串。这意味着`strip()`等方法不只删除空白，还可以根据需要删除任何内容。

```
► In [1]: xml_tag = "<some_tag>"  
In [2]: xml_tag.lstrip("<")  
Out[2]: 'some_tag>'  
In [3]: xml_tag.lstrip(">")  
Out[3]: '<some_tag>'  
In [4]: xml_tag.rstrip(">")  
Out[4]: '<some_tag'  
In [5]: xml_tag.rstrip("<")  
Out[5]: '<some_tag>'
```

以下示例演示如何依次去除XML标签的左尖括号和右尖括号。但是如果想同时删除左右尖括号时，该如何解决呢？可以这样来操作：

```
► In [6]: xml_tag.strip("<").strip(">")  
Out[6]: 'some_tag'
```

既然`strip()`等方法可以返回一个字符串，就可以在`strip()`调用之后直接调用另一个字符串操作。这里，我们将`strip()`调用连续使用。第一个`strip()`调用去除起始字符（左尖括号）并且返回一个字符串，第二个`strip()`函数删除末尾字符（右尖括号）并且返回字符串“`some_tag`”。但是这里有一个更为简单的方法：

```
In [7]: xml_tag.strip("<>")
```

```
Out[7]: 'some_tag'
```

你可能设想strip()能够按照你所输入的字符进行准确删除，但是实际上，strip()将从字符串的适当一侧开始删除这些指定字符的任意顺序组合。在上面的示例中，我们告诉strip()删除“<>”。这不意味着将删除准确匹配的“<>”，也就是左尖括号在前右尖括号在后的组合，而是可以删除任何“<”和“>”的组合。

下面是一个比较清楚的示例：

```
In [8]: gt_lt_str = "<><><>gt lt str<><><>"
```

```
In [9]: gt_lt_str.strip("<>")
```

```
Out[9]: 'gt lt str'
```

```
In [10]: gt_lt_str.strip("><")
```

```
Out[10]: 'gt lt str'
```

我们去除了在字符串两边出现的“<”或“>”，只保留了字母和空格。

下面或许仍不是你希望的结果。例如：

```
In [11]: foo_str = "<oooooooo>blah<foo>"
```

```
In [12]: foo_str.strip("<foo>")
```

```
Out[12]: 'blah'
```

你可能希望strip()可以按照输入字符的顺序进行匹配删除，但是它并不是按照“<”、“f”、“o”和“>”的顺序进行匹配的。它将删除字符串中包含的全部4个字符，绝不会漏下一个“o”。下面是另一个strip()示例，可以清晰的看到这一点：

```
In [13]: foo_str.strip("><of")
```

```
Out[13]: 'blah'
```

在这个示例中，strip()同样也删除了“<”、“f”和“o”，尽管字符根本不是按这个顺序排列的。

upper()方法和lower()方法非常有用，尤其是在需要对两个字符串进行比较，并且不考虑字符是大写或是小写时。upper()方法返回一个字符串，该字符串是大写的原始字符串的大写。lower()方法返回一个字符串，该字符串是小写的原始字符串（参见例3-10）。

### 例3-10：upper()和lower()

```
In [1]: mixed_case_string = "V0rpal BUnny"
```

```
In [2]: mixed_case_string == "vorpal bunny"
Out[2]: False

In [3]: mixed_case_string.lower() == "vorpal bunny"
Out[3]: True

In [4]: mixed_case_string == "VORPAL BUNNY"
Out[4]: False

In [5]: mixed_case_string.upper() == "VORPAL BUNNY"
Out[5]: True

In [6]: mixed_case_string.upper()
Out[6]: 'VORPAL BUNNY'

In [7]: mixed_case_string.lower()
Out[7]: 'vorpal bunny'
```

如果需要根据某个指定的分隔符对一个字符串进行提取，`split()`方法正好可以完成这类任务。参见例3-11。

### 例3-11：`split()`

```
→ In [1]: comma_delim_string = "pos1,pos2,pos3"

In [2]: pipe_delim_string = "pipepos1|pipepos2|pipepos3"

In [3]: comma_delim_string.split(',')
Out[3]: ['pos1', 'pos2', 'pos3']

In [4]: pipe_delim_string.split('|')
Out[4]: ['pipepos1', 'pipepos2', 'pipepos3']
```

`split()`方法的典型用法是把希望作为分割符的分割的字符串作为参数传递给它。通常，分隔符是单个字管道符，例如逗号或符，但是也可以是多于一个字符的字符串。我们能够通过`split()`函数，以逗号分割`comma_delim_string`，以管道符（|）分割`pipe_delim_string`，只需要将逗号和管道符分别传递给`split()`函数即可。`split()`的返回值是一系列字符串，每一个都是一组位于两个指定的分隔符之间的连续字符。当你需要的分割符是多个连续的字符，而不是单个字符时，`split()`方法也提供了支持。在我们写这本书的时候，Python中还没有字符类型，因此，虽然在两种情况下我们传递给`split()`方法的都只是单个字符，但实际上仍是一个字符串。因此在传递给`split()`多个字符时，它仍会正常工作。参见例3-12。

### 例3-12：`split()`多定界符（delimiter）示例

```
→ In [1]: multi_delim_string = "pos1XXXpos2XXXpos3"

In [2]: multi_delim_string.split("XXX")
Out[2]: ['pos1', 'pos2', 'pos3']

In [3]: multi_delim_string.split("XX")
Out[3]: ['pos1', 'Xpos2', 'Xpos3']
```

```
In [4]: multi_delim_string.split("X")
Out[4]: ['pos1', '', '', 'pos2', '', '', 'pos3']
```

注意，首先为`multi_delim_string`定义"XXX"作为定界字符串。正如我们所期望的，返回结果为['pos1', 'pos2', 'pos3']。接下来定义"XX"作为定界字符串，`split()`返回['pos1', 'Xpos2', 'Xpos3']。`split()`会对出现在每对定界字符串"XX"之间的字符进行查找。“Pos1”从字符串起始位置，到第一个"XX"定界符；"Xpos2"出现在从第一次出现"XX"到第二次出现之间。最后一个`split()`使用了单个字符"X"作为定界字符串。值得注意的是，针对有两个紧临的"X"字符的位置，在返回列表里有一个空字符串""。这意味着在字符"X"之间没有字符。

但是，如果只是想在指定定界符第一次出现字符"n"的位置对字符串进行分割，那该怎么办呢？`split()`使用称为`max_split`的第2个参数。当一个整数值作为`max_split`被传递进来，`split()`仅对字符串分割由`max_split`指定的次数。

```
In [1]: two_field_string = "8675309,This is a freeform, plain text, string"
In [2]: two_field_string.split(',', 1)
Out[2]: ['8675309', 'This is a freeform, plain text, string']
```

我们根据逗号对字符串进行分割，并且仅对第一次出现的逗号定界符进行分割。虽然在这个示例中有许多逗号出现，字符串仅依据第一个进行分割。

如果想顺序去除诗歌等文本中的空格，`split()`是一个非常好的工具：

```
In [1]: prosaic_string = "Insert your clever little piece of text here."
In [2]: prosaic_string.split()
Out[2]: ['Insert', 'your', 'clever', 'little', 'piece', 'of', 'text', 'here.']
```

因为没有传递参数，`split()`将空格默认为分隔符。

绝大多数时候，使用`split`会看到期望的结果。然而，如果遇到多行文本，或许会得不到所期望的结果。通常，在处理多行文本时，需要一次处理一行。但是你或许会发现`split`会对字符串中的每一个词进行分割：

```
In [1]: multiline_string = """This
...: is
...: a multiline
...: piece of
...: text"""
In [2]: multiline_string.split()
Out[2]: ['This', 'is', 'a', 'multiline', 'piece', 'of', 'text']
```

在这个示例中，使用`splitlines()`会获得你更想得到的结果：

```
→ In [3]: lines = multiline_string.splitlines()
In [4]: lines
Out[4]: ['This', 'is', 'a multiline', 'piece of', 'text']
```

`splitlines()`返回一个由字符串中的每一行所组成的列表，并且保存为一组。从这里可以看到，你可以迭代每一行，并将每一行内容分割为单词：

```
→ In [5]: for line in lines:
...:     print "START LINE::"
...:     print line.split()
...:     print "::END LINE"
...
START LINE::
['This']
::END LINE
START LINE::
['is']
::END LINE
START LINE::
['a', 'multiline']
::END LINE
START LINE::
['piece', 'of']
::END LINE
START LINE::
['text']
::END LINE
```

有时，并不想分割字符串或从字符串中提取信息，而是希望将多个字符串连接到一起。在这种情况下，`join()`可以帮助完成该工作：

```
→ In [1]: some_list = ['one', 'two', 'three', 'four']
In [2]: ','.join(some_list)
Out[2]: 'one,two,three,four'

In [3]: ', '.join(some_list)
Out[3]: 'one, two, three, four'

In [4]: '\t'.join(some_list)
Out[4]: 'one\ttwo\tthree\tfour'

In [5]: ''.join(some_list)
Out[5]: 'onetwothreefour'
```

例如指定一个列表`some_list`，就可以将字符串“one”、“two”、“three”和“four”组合到一些变量中。我们使用一个逗号，再一个逗号，然后是一个空格和一个tab将列表`some_list`进行连接。`join()`是字符串方法，因此对于一个类似“,”的字符串，调用`join()`是有效的。`join()`会采用一连串的多个字符串作为参数。它将多个字符

串压缩成单个字符串，这样列表中的每个字符串将按顺序排列，但是调用join()的字符串会在序列中每一项前出现。

我们对join()和参数的使用有一些建议。注意，join()需要一个字符串序列。如果你给join()传递的是一系列整数，那会出现什么情况呢？

```
→ In [1]: some_list = range(10)

In [2]: some_list
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: ",".join(some_list)
-----
exceptions.TypeError                                Traceback (most recent call last)

/Users/jmjones/<ipython console>

TypeError: sequence item 0: expected string, int found
```

join()抛出的异常追踪是自解释的，但是由于这是一个常见错误，值得引起注意。只需对列表稍加理解就可以很容易地避免这种问题的发生。这里列出了帮助信息，来帮助理解如何将some\_list中所有整数元素转换为字符串：

```
→ In [4]: ",".join([str(i) for i in some_list])
Out[4]: '0,1,2,3,4,5,6,7,8,9'
```

也可以使用一个表达式：

```
→ In [5]: ",".join(str(i) for i in some_list)
Out[5]: '0,1,2,3,4,5,6,7,8,9'
```

关于列表的更多内容，可以参见《Python in a Nutshell》（也可以在线访问<http://safarible.com/0596100469/pythonian-CHP-4-SECT-10>）第4章“流程控制语句”。

最后要介绍的是用于创建和修改文本字符的replace()方法。replace()有两个参数，分别是被替换的字符串以及替换字符串。下面是一个简单的replace()方法示例：

```
→ In [1]: replacable_string = "trancendental hibernational nation"
In [2]: replacable_string.replace("nation", "natty")
Out[2]: 'trancendental hibernattyal natty'
```

值得注意的是，replace()不关心替换字符串是在一个词的中间或者就是一个完整的单词。因此，在需要使用指定的字符序列去替换另一个字符序列时，replace()是一个非常好的工具。

然而，当需要用一个字符序列去替换另一个字符序列，但需要进行更为精确的控制时，这是不够的。有时需要指定一个字符模式来实现查找和替换。模式可以帮助实现对文本

的搜索，从而实现数据的提取。在一些更适于使用模式的情况下，使用正则表达式是非常有帮助的。接下来我们会介绍正则表达式。

---

注意：与切分操作以及strip()方法一样，replace()会创建一个新字符串，而不是对字符串进行内修改。

---

## Unicode字符串

到目前为止，所有查找字符串的示例全部都使用了内建的字符串类型（str），但是Python还有另一种你可能想要熟悉的字符串类型：Unicode。当你看到在计算机屏幕上显示的字符时，计算机正将其作为数字在内部正进行处理。根据语言和平台的不同，有许多种不同的数字-字符映射集。Unicode是一个标准，提供了数字-字符的单一映射，而无须考虑语言、平台或者是对文本进行处理的程序。在这一章，我们引入Unicode的概念并介绍Python如何对Unicode进行处理。要对Unicode作更深一步的了解，可以参阅A.M.Kuchling的非常不错的Unicode教程，网址是：<http://www.amk.ca/python/howto/unicode>。

创建一个Unicode字符串与创建一个普通的字符串一样简单：

```
► In [1]: unicode_string = u'this is a unicode string'  
In [2]: unicode_string  
Out[2]: u'this is a unicode string'  
In [3]: print unicode_string  
this is a unicode string
```

你也可以使用内建的unicode()函数：

```
► In [4]: unicode('this is a unicode string')  
Out[4]: u'this is a unicode string'
```

使用Unicode字符串看起来并不会给我们增加许多麻烦，尤其是仅对来自一种语言的字符进行处理的时候。但是当字符串包含多种语言的字符时该如何处理呢？Unicode会帮助你处理这种情况。可以使用指定的数值创建一个Unicode字符，你可以使用“\uXXX”或者是“\uXXXXXXXXXX”来表示。例如，下面是一个Unicode字符串，其中包括了拉丁、希腊和俄文字符：

```
► In [1]: unicode_string = u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042f'  
In [2]: unicode_string  
Out[2]: u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042f'
```

Python依据使用的编码产生字符串（str）。对于与Mac标准一致的Python，如果试图从之前的示例中打印字符串，将会返回一个错误，打印内容如下：

```
In [3]: print unicode_string
-----
UnicodeEncodeError                                 Traceback (most recent call last)
/Users/jmjones/<ipython console> in <module>()
UnicodeEncodeError: 'ascii' codec can't encode characters in position 4-6:
ordinal not in range(128)
```

我们不得不告诉它所使用的编码，这样它就知道如何处理我们使用的字符：

```
In [4]: print unicode_string.encode('utf-8')
abc_ΠΣΩΔΦЯ
```

这里，我们采用UTF-8格式对包含拉丁、希腊以及俄文字符的字符串进行编码，这是Unicode数据的常用编码。

Unicode字符串包含了许多功能，其中包括在in的示例中已经提到的正则表达式的用法。

```
In [5]: u'abc' in unicode_string
Out[5]: True

In [6]: u'foo' in unicode_string
Out[6]: False

In [7]: unicode_string.split()
Out[7]: [u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042f']

In [8]: unicode_string.
unicode_string.__add__          unicode_string.expandtabs
unicode_string.__class__        unicode_string.find
unicode_string.__contains__    unicode_string.index
unicode_string.__delattr__      unicode_string.isalnum
unicode_string.__doc__         unicode_string.isalpha
unicode_string.__eq__          unicode_string.isdecimal
unicode_string.__ge__          unicode_string.isdigit
unicode_string.__getattribute__ unicode_string.islower
unicode_string.__getitem__     unicode_string.isnumeric
unicode_string.__getnewargs__   unicode_string.isspace
unicode_string.__getslice__    unicode_string.istitle
unicode_string.__gt__          unicode_string.isupper
unicode_string.__hash__         unicode_string.join
unicode_string.__init__        unicode_string.ljust
unicode_string.__le__          unicode_string.lower
unicode_string.__len__         unicode_string.lstrip
unicode_string.__lt__          unicode_string.partition
unicode_string.__mod__         unicode_string.replace
unicode_string.__mul__         unicode_string.rfind
unicode_string.__ne__          unicode_string.rindex
unicode_string.__new__         unicode_string.rjust
unicode_string.__reduce__      unicode_string.rpartition
```

unicode_string.__reduce_ex__	unicode_string.rsplit
unicode_string.__repr__	unicode_string.rstrip
unicode_string.__rmod__	unicode_string.split
unicode_string.__rmul__	unicode_string.splitlines
unicode_string.__setattr__	unicode_string.startswith
unicode_string.__str__	unicode_string.strip
unicode_string.capitalize	unicode_string.swapcase
unicode_string.center	unicode_string.title
unicode_string.count	unicode_string.translate
unicode_string.decode	unicode_string.upper
unicode_string.encode	unicode_string.zfill
unicode_string.endswith	

你或许现在不需要Unicode。但是如果你希望能够一直使用Python编程，熟悉Unicode是非常必要的。

## re

既然Python是一个连“电池都包括在内”的语言，你或许会希望Python也应包括一个正则表达式库。这一点不会让你失望的。本节讲述的重点是如何在Python中使用正则表达式，而不是正则表达式的in和out语法。因此，如果对正则表达式不熟悉，我们建议你阅读由Jeffrey E. F. Friedl编著的《Mastering Regular Expressions》（O'Reilly出版，你也可以访问<http://safari.oreilly.com/0596528124>）。本节假定你已经掌握了正则表达式，但是如果还没有掌握，不妨在手边准备一本Friedl的这本著作，这将是非常有帮助的。

如果对Perl比较熟悉，你很可能习惯于通过“=~”来使用正则表达式。Python所包括的正则表达式来自于库，而不是语言自身的语法特征。因此，为了使用正则表达式，必须首先载入正则表达式模块re。下面是一个基本的示例，展示了正则表达式的创建和使用。参见例3-13。

### 例3-13：基本正则表达式的使用

```
In [1]: import re

In [2]: re_string = "{{(.*)}}"

In [3]: some_string = "this is a string with {{words}} embedded in\
...: {{curly brackets}} to show an {{example}} of {{regular expressions}}"

In [4]: for match in re.findall(re_string, some_string):
...:     print "MATCH->", match
...:
MATCH-> words
MATCH-> curly brackets
MATCH-> example
MATCH-> regular expressions
```

我们做的第一件事情是加载re模块。或许正如你猜想的那样，re代表了“正则表达式”。接下来创建一个字符串re\_string，这将是我们在示例中进行查找操作所使用的

模式。该模式匹配两个连续的左大括号（{{），然后是任意文本（也可以为空），最后是两个连续的右大括号（}}）。然后，我们创建了一个字符串some\_string，该字符串包括由大括号括起来的一组单词以及没有被大括号括起来的单词。最后，我们循环使用re模块中的`.findall()`函数对匹配结果进行处理，`.findall()`将根据模式re\_string搜索some\_string字符串。正如你所看到的，输出结果中包括words、curly brackets、example以及regular expressions，这正是双大括号中的所包含的内容。

在Python中有两种使用正则表达式的方式。第一种是直接使用re模块中的函数，正如上例中所演示的那样。第二种是创建一个已编译的正则表达式对象，然后使用对象中的方法。

什么是已编译的正则表达式呢？它是一个简单的对象，通过传递一个模式给`re.compile()`来创建；它包括一些正则表达式方法，也通过传递模式给`re.compile()`来创建。在使用编译与非编译的示例中存在两个主要的差别。首先，没有继续引用正则表达式模式"{{(.\*?)}"}"，而是创建了一个编译的正则表达式对象，并且是使用模式来创建。第二，没有在re模块上调用`.findall()`，而是在编译后的正则表达式对象上调用`.findall()`。

关于re模块内容的更多信息，包括可用函数等，参见《Python Library Reference》的Module Contents相关内容，<http://docs.python.org/lib/node46.html>。更多的关于编译后正则表达式对象的内容，可参见《Python Library Reference》的Regular Expression Objects相关内容，网络链接地址是：<http://docs.python.org/lib/re-objects.html>。

例3-14演示了双大括号示例，并展示了如何使用一个编译后的正则表达式对象。

#### 例3-14：简单正则表达式，已编译模式

```
In [1]: import re
In [2]: re_obj = re.compile("{{(.*?)}}")
In [3]: some_string = "this is a string with {{words}} embedded in\
...: {{curly brackets}} to show an {{example}} of {{regular expressions}}"
In [4]: for match in re_obj.findall(some_string):
...:     print "MATCH->", match
...
MATCH-> words
MATCH-> curly brackets
MATCH-> example
MATCH-> regular expressions
```

在Python中，你可以根据个人喜好选择使用正则表达式的方法。然而，选用不同的方法会对执行的性能产生影响如果你在某些循环中多次重复某一操作，例如在一个循环中将正则表达式应用于一个有数百万行文本的文件的每一行，性能问题便会变得明显起来。

在以下的示例中，我们运行了一个简单的包括正则表达式的脚本，脚本中使用了编译和非编译的正则表达式，处理的文件包括50万行文本。我们运行Unix下的timeit工具对脚本执行情况进行检测，使你能够看到性能的差异。参见例3-15。

### 例3-15：re非编译代码性能检测

```
#!/usr/bin/env python

import re

def run_re():
    pattern = 'pDq'

    infile = open('large_re_file.txt', 'r')
    match_count = 0
    lines = 0
    for line in infile:
        match = re.search(pattern, line)
        if match:
            match_count += 1
        lines += 1
    return (lines, match_count)

if __name__ == "__main__":
    lines, match_count = run_re()
    print 'LINES:::', lines
    print 'MATCHES:::', match_count
```

timeit工具执行一段代码数次，然后报告最佳运行所花费的时间。下面是在IPytnon中运行Python的timeit工具来执行这段代码结果：

```
In [1]: import re_loop_nocompile
In [2]: timeit -n 5 re_loop_nocompile.run_re()
5 loops, best of 3: 1.93 s per loop
```

该示例执行run\_re()函数5次，并且报告最佳运行花费了1.93秒/次。timeit在一段时间内重复运行一段相同的代码的原因，是为了减少其他在同一运行时间运行的进程对测试结果的影响。

以下是使用Unix的time工具对相同代码的测试结果：

```
jmjones@dink:~/code$ time python re_loop_nocompile.py
LINES:: 500000 MATCHES:: 242
real 0m2.113s
user 0m1.888s
sys 0m0.163s
```

例3-16是相同的正则表达式示例，只是这里使用re.compile()来创建一个编译的模式对象。

### 例3-16: re编译代码性能检测

```
#!/usr/bin/env python

import re

def run_re():
    pattern = 'pDq'
    re_obj = re.compile(pattern)

    infile = open('large_re_file.txt', 'r')
    match_count = 0
    lines = 0
    for line in infile:
        match = re_obj.search(line)
        if match:
            match_count += 1
        lines += 1
    return (lines, match_count)

if __name__ == "__main__":
    lines, match_count = run_re()
    print 'LINES:::', lines
    print 'MATCHES:::', match_count
```

在IPython中使用Python的timeit工具运行该脚本，所产生的结果如下：

```
In [3]: import re_loop_compile

In [4]: timeit -n 5 re_loop_compile.run_re()
5 loops, best of 3: 860 ms per loop
```

使用Unix的time工具运行相同的脚本，所产生的结果如下：

```
jmjones@dink:~/code$ time python
re_loop_compile.py LINES:: 500000 MATCHES:: 242
real 0m0.996s
user 0m0.836s
sys 0m0.154s
```

很明显，编译版本更为优越。正如由Unix的time和Python的timeit工具所测量的结果所表明的，它只花费了一半的运行时间。因此，我们强烈建议养成创建编译后的正则表达式的习惯。

正如在这一章前面部分所讨论的，原始字符串可以用于表示不对转义序列进行解析的字符串。例3-17显示了在正则表达式中使用的原始字符串。

### 例3-17：原始字符串与正则表达式

```
In [1]: import re

In [2]: raw_pattern = r'\b[a-z]+\b'
```

```
In [3]: non_raw_pattern = '\b[a-z]+\b'  
In [4]: some_string = 'a few little words'  
In [5]: re.findall(raw_pattern, some_string)  
Out[5]: ['a', 'few', 'little', 'words']  
In [6]: re.findall(non_raw_pattern, some_string)  
Out[6]: []
```

正则表达式模式“\b”匹配词边界。因此在原始（raw）及普通字符串中，我们寻找到了单个的小写单词。值得注意的是，`raw_pattern`匹配了在`some_string`中合适的单词边界，而`non_raw_pattern`根本没有任何匹配。`raw_pattern`将“\b”识别为两个字符，而不是解析为转义字符中的退格字符。`non_raw_pattern`则将“\b”解析为转义字符中的退格字符。正则表达式函数`findall()`可以使用原始字符串模式来查找单词。但是，当`findall()`使用非原始字符串模式进行搜索时，不会找到任何退格字符。

对于使用`non_raw_pattern`匹配字符串的问题，正如下面对单词“little”所做的操作一样，我们在其前后放置退格字符：

```
► In [7]: some_other_string = 'a few \blittle\b words'  
In [8]: re.findall(non_raw_pattern, some_other_string)  
Out[8]: ['\x08little\x08']
```

值得注意的是`findall()`函数在单词“little”之前和之后匹配了十六进制符号“\x08”。该十六进制符号对应于退格字符，即使用转义字符"\b"插入的字符。

因此，正如你所看到的，如果希望使用这种反斜杠加指定字符的方法（例如"\b"可以作为单词的边界，“\d”表示数字，“\w”表示希腊数字字符），原始字符串是非常有用的。如果想查看完整的由反斜杠定义的转义字符列表，参见《Python Library Reference》(<http://docs.python.org/lib/re-syntax.html>) 中的正则表达式语法一节。

从例3-14到例3-17，无论是正则表达式，还是我们应用的不同方法都相当简单。有时，对正则表达式强大功能的这种有限使用就能满足我们的需要了。其他时候，你需要更多地利用包含在正则表达式库中的强大功能。

四个主要的也是最经常使用的正则表达式方法（或者称为函数）有`findall()`、`finditer()`、`match()`和`search()`。你或许也经常使用`split()`或是`sub()`，但很可能还是比不上使用其他那几个函数那样频繁。`findall()`可以找到搜索字符串中指定模式的所有匹配。`findall()`匹配模式返回的数据结构类型将取决于模式是否定义了一个组。

---

**注意：**关于正则表达式的一个快捷提示：分组允许你在一个正则表达式中定义需要提取的文本。要获得更多信息，请参见Friedl编著的《Mastering Regular Expressions》中“通用元字符与域”的相关内容，或在线访问以下网址：<http://safari.oreilly.com/0596528124/regex3-CHP-3-SECT-5?imagepage=137>。

---

如果没有在正则表达式模式中定义组，但是却找到了匹配，`.findall()`将返回一个字符串列表。例如：

```
In [1]: import re  
In [2]: re_obj = re.compile(r'\bt.*?e\b')  
In [3]: re_obj.findall("time tame tune tint tire")  
Out[3]: ['time', 'tame', 'tune', 'tint tire']
```

模式没有定义任何组，因此`.findall()`返回一个字符串列表。有趣的是返回列表的最后一个元素包含两个单词：tint和tire。正则表达式希望匹配以“t”开始的单词和以“e”结尾的单词。但是命令“.\*?”匹配所有字符包括空白。`.findall()`搜索所有可能的匹配。它找到一个以“t”开始的单词（tint），然后继续查找字符串，直到找到一个以“e”结尾的单词（tire）。因此，所匹配的“tint tire”是正确的。如果希望去除空白，你可以使用“r'\bt\w\*e\b'”：

```
In [4]: re_obj = re.compile(r'\bt\w*e\b')  
In [5]: re_obj.findall("time tame tune tint tire")  
Out[5]: ['time', 'tame', 'tune', 'tire']
```

第二个可能被返回的数据结构类型是一个多元列表。如果定义了一个组，并且其中有一个匹配，那么`.findall()`返回一个多元列表。例如，例3-18是一个使用这种模式和字符串的简单示例。

### 例3-18：`.findall()`简单分组

```
In [1]: import re  
In [2]: re_obj = re.compile(r"""(A\W+\b(big|small)\b\W+\b  
...: (brown|purple)\b\W+\b(cow|dog)\b\W+\b(ran|jumped)\b\W+\b  
...: (to|down)\b\W+\b(the)\b\W+\b(street|moon).*?\.)""",  
...: re.VERBOSE)  
In [3]: re_obj.findall('A big brown dog ran down the street. \  
...: A small purple cow jumped to the moon.')  
Out[3]:  
[('A big brown dog ran down the street.',  
 'big',  
 'brown',  
 'dog',  
 'ran',  
 'to',  
 'down',  
 'the',  
 'small',  
 'purple',  
 'cow',  
 'jumped',  
 'street',  
 'moon')]  
Out[4]: 11
```

```
'down',
'the',
'street'),
('A small purple cow jumped to the moon.',
'small',
'purple',
'cow',
'jumped',
'to',
'the',
'moon')]
```

尽管非常简单，该示例还是展示了一些重要内容。首先值得注意的是，这个简单模式比较长，而且包括了许多非数字字符（如果你长时间盯着这些长字符串，眼睛都会充血）。这似乎是许多正则表达式的常用格式。其次，模式包括明确的组嵌套。外层组匹配以字母“A”开始到结尾的所有字符。在A和结束字符之间的所有字符组成了内层组，内层组匹配“big or small”、“brown or purple”等。接下来，`findall()`的返回值是一个多元列表。这个多元列表的每一个元素是我们在正则表达式中定义的组之一。整个语句是多元组的第一个元素，因此它是最大、最外层的组。每一个子组由多元元素序列组成。值得注意的是`re.compile()`的最后一个参数`re.VERBOSE`。它表示允许以冗余模式编写正则表达式字符串。这意味着可以非常简单地在整个行上分割正则表达式，而模式匹配不受分割的影响。在组之外的空白将被忽略掉。`verbose`允许在正则表达式每一行的结尾处插入注释，以记录每一个特殊的正则表达式都完成了什么，但是我们经常选择不去这样做。正则表达式的难点之一是模式的描述，你想要匹配的模式通常会变得巨大且很难阅读。`re.VERBOSE`函数允许写一个简单的正则表达式，这对于改善代码（包括正则表达式）的可维护性是一个非常不错的工具。

`Finditer()`是对`findall()`的略微修改，不同于`findall()`之处在于`findall()`所返回的是一个多元列表，而正如其名称所表明的，`finditer()`返回一个迭代。每一次迭代的元素都是一个正则表达式匹配的对象，这将在本章后面进行介绍。例3-19是一个使用`finditer()`而不是`findall()`的同样非常简单的示例。

#### 例3-19：`finditer()`示例

```
In [4]: re_iter = re_obj.finditer('A big brown dog ran down the street.\n...: A small purple cow jumped to the moon.')
In [5]: re_iter
Out[5]: <callable-iterator object at 0xa17ad0>
In [6]: for item in re_iter:
...:     print item
...:     print item.groups()
...:
<_sre.SRE_Match object at 0x9ff858>
```

```
('A big brown dog ran down the street.', 'big', 'brown', 'dog', 'ran',
'down', 'the', 'street')
<_sre.SRE_Match object at 0x9ff940>
('A small purple cow jumped to the moon.', 'small', 'purple', 'cow',
'jumped', 'to', 'the', 'moon')
```

如果之前曾经接触过迭代器，你或许会把它想象成一个根据需要创建的列表。但这样认为是错误的。原因之一是在迭代器中不能通过索引来引用某一特定元素，但对于一个列表，却可以这样做，如`some_list[3]`。这一限制的后果是你不具备将迭代器进行分割的能力，而这在列表中是可以的，如`some_list[2:6]`。如果不考虑这个限制，迭代器可以说是轻量且功能强大的，尤其是仅需要迭代一些序列时。因为不需要将整个序列加载到内存，而是根据需要进行获取。因此，一个迭代器与其对应的列表相比，需要更小的内存。这也意味着迭代器在访问一个序列中的元素时，在很短的时间内就可以启动。

使用`finditer()`而不是`findall()`的另一个原因是`finditer()`的每一项都是一个匹配对象，而不是对应于匹配文本的简单的字符串列表或是多元列表。

`match()`与`search()`提供了相似的功能。它们对字符串应用一个正则表达式，定义开始搜索和结束搜索的位置，并且都返回一个首次匹配指定模式的匹配对象。两者之间的差异是`match()`从指定位置开始进行匹配，而不会移动到字符串的任何随机位置。但`search()`从指定的位置开始搜索，在指定的位置结束搜索。参见例3-20。

#### 例3-20：比较`match()`与`search()`

```
In [1]: import re

In [2]: re_obj = re.compile('FOO')

In [3]: search_string = ' FOO'

In [4]: re_obj.search(search_string)

Out[4]: <_sre.SRE_Match object at 0xa22f38>

In [5]: re_obj.match(search_string)

In [6]:
```

尽管`search_string`包括`match()`搜索的模式，但不能找到匹配。因为存在匹配的`search_string`子字符串无法在`search_string`起始位置开始查询。`search()`调用会创建一个匹配对象。

`search()`和`match()`调用接受开始和结束参数，这两个参数定义Python针对某一模式开始搜索的起始位置与结束位置。参见例3-21。

#### 例3-21：`search()`和`match()`的开始和结束

```
In [6]: re_obj.search(search_string, pos=1)
```

```
Out[6]: <_sre.SRE_Match object at 0xabe030>  
In [7]: re_obj.match(search_string, pos=1)  
Out[7]: <_sre.SRE_Match object at 0xabe098>  
In [8]: re_obj.search(search_string, pos=1, endpos=3)  
In [9]: re_obj.match(search_string, pos=1, endpos=3)  
In [10]:
```

参数pos是一个索引，该索引定义Python在字符串中寻找某一模式的位置。为search()定义起始参数pos不会改变任何事情，但是为match()定义pos参数会让其匹配在不使用pos时无法匹配模式。设置结束参数endpos为3将导致search()和match()都无法匹配模式，因为模式在第三个字符位置之后出现。

.findall()和finditer()需要回答这样的问题：“我的模式匹配什么？”而search()和match()面临的主要问题是“我的模式匹配吗？”。search()和match()还需要回答这个问题：“我的模式第一次匹配的内容是什么？”。但是最经常的，也是你真正想知道的问题是“我的模式匹配吗？”。例如，假设你正在写一段代码来读取logfiles，并且为了显示效果更好，以HTML格式显示且在每一行自动换行。你想要所有的“ERROR”行以红色显示，因此你可以循环检查文件中的每一行，看它是否匹配正则表达式，并且如果search()在搜索中出现命中“ERROR”的情况，你需要对该行进行格式化，用红色进行显示。

search()和match()是非常好的工具，因为它们不仅可以显示一个模式是否匹配一段文本，而且可以返回一个match()对象。match()对象包含当你搜索文本时各种各样的数据片段。start()、end()、span()、group()和groupdict()都是特别有趣的match()方法。

start()、end()和span()定义搜索字符串时模式匹配的开始和结束位置。start()返回一个整数，该整数代表字符串中模式开始进行匹配操作的位置。end()也返回一个整数，该整数代表模式匹配结束的位置。span()返回一个多元组，包括匹配的开始和结尾。

groups()返回一个匹配的多元组，每一元素都是模式所指定的组。这个元组与.findall()返回的元组相似。groupdict()返回一个命名组字典，组的名字通过正则表达式自身的“?P<group\_name>pattern”语法确定。

总之，为了有效地使用正则表达式，养成使用编译后的正则表达式对象的习惯是非常重要的。在希望查看模式匹配的文本是什么时，可以使用.findall()和finditer()。记住，finditer()比.findall()更具灵活性，因为它返回一个匹配对象的迭代。如果希望更详细地了解正则表达式库，参见由Alex Martelli编著的《Python in a Nutshell》

(O'Reilly) 第9章。如果希望查看一些正则表达式的实际应用，参见Greg Wilson编著的《DataCrunching》(The Pragmatic Bookshelf)。

## Apache配置文件详解

在学习了Python的正则表达式之后，让我们进一步了解Apache的配置文件：

```
▶ NameVirtualHost 127.0.0.1:80
<VirtualHost localhost:80>
    DocumentRoot /var/www/
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    ErrorLog /var/log/apache2/error.log
    LogLevel warn
    CustomLog /var/log/apache2/access.log combined
    ServerSignature On
</VirtualHost>
<VirtualHost local2:80>
    DocumentRoot /var/www2/
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    ErrorLog /var/log/apache2/error2.log
    LogLevel warn
    CustomLog /var/log/apache2/access2.log combined
    ServerSignature On
</VirtualHost>
```

该文件是将安装在Ubuntu上的Apache2的配置文件略作修改得到的。我们创建了虚拟用户进行配置。修改/etc/hosts文件，加入如下所示的行：

```
▶ 127.0.0.1      local2
```

这允许在浏览器中输入local2，并且将其解析为127.0.0.1，这是一个本地主机地址。那么这么做的有什么作用呢？如果访问http://local2，浏览器会将主机名通过HTTP请求进行传递。下面是一个HTTP对local2的请求：

```
▶ GET / HTTP/1.1
Host: local2
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.8.1.13)
Gecko/20080325 Ubuntu/7.10 (gutsy) Firefox/2.0.0.13
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

```
If-Modified-Since: Tue, 15 Apr 2008 17:25:24 GMT  
If-None-Match: "ac5ea-53-44aecf804900"  
Cache-Control: max-age=0
```

请注意以“Host:”开始的行。当Apache获得一个请求，它将其传递给匹配local2的虚拟主机。

因此，我们需要写一个脚本，解析Apache配置文件（类似于刚刚所演示的），找到VirtualHost部分，替换DocumentRoot。脚本如下所示：

```
#!/usr/bin/env python

from cStringIO import StringIO
import re

vhost_start = re.compile(r'<VirtualHost\s+(.*?)>')
vhost_end = re.compile(r'</VirtualHost>')
docroot_re = re.compile(r'(DocumentRoot\s+)(\S+)')

def replace_docroot(conf_string, vhost, new_docroot):
    '''yield new lines of an httpd.conf file where docroot lines matching
    the specified vhost are replaced with the new_docroot
    ...
    conf_file = StringIO(conf_string)
    in_vhost = False
    curr_vhost = None
    for line in conf_file:
        vhost_start_match = vhost_start.search(line)
        if vhost_start_match:
            curr_vhost = vhost_start_match.groups()[0]
            in_vhost = True
        if in_vhost and (curr_vhost == vhost):
            docroot_match = docroot_re.search(line)
            if docroot_match:
                sub_line = docroot_re.sub(r'\1%s' % new_docroot, line)
                line = sub_line
        vhost_end_match = vhost_end.search(line)
        if vhost_end_match:
            in_vhost = False
            yield line
    if __name__ == '__main__':
        import sys
        conf_file = sys.argv[1]
        vhost = sys.argv[2]
        docroot = sys.argv[3]
        conf_string = open(conf_file).read()
        for line in replace_docroot(conf_string, vhost, docroot):
            print line,
```

这个脚本先建立三个编译后的正则表达式对象：一个匹配VirtualHost的开始，一个匹配VirtualHost的结尾，一个匹配DocumentRoot这一行。我们创建了一个函数来完成繁杂的工作。函数被命名为replace\_docroot()，其参数包括配置文件名，匹配的VirtualHost

名和指向virtualHost的目录名DocumentRoot。该函数建立一个状态机，检测我们是否处于一个virtualHost部分，并保持对virtualHost的追踪。当处于调用代码定义的virtualHost中时，函数寻找DocumentRoot指令出现的位置，并且将指令所指向的目录更改为为调用代码定义的目录。当replace\_docroot()迭代配置文件中的每一行时，将产生没有修改的输入行或修改过的DocumentRoot行。

我们为此函数创建了一个简单的命令行接口。使用optparse进行实现没什么可奇怪的，对给定的参数数目进行错误检查也同样没什么异样，这些都是功能性的操作。现在在之前展示的Apache配置文件上运行脚本，修改“VirtualHost local2:80”，使用“/tmp”作为VirtualHost。该命令行接口打印输出从函数replace\_docroot()得到的结果，而不是将其写入一个文件：

```
jmjones@dinkgutsy:code$ python apache_conf_docroot_replace.py
/etc/apache2/sites-available/psa
local2:80 /tmp
NameVirtualHost 127.0.0.1:80
<VirtualHost localhost:80>
    DocumentRoot /var/www/
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    ErrorLog /var/log/apache2/error.log
    LogLevel warn
    CustomLog /var/log/apache2/access.log combined
    ServerSignature On
</VirtualHost>
<VirtualHost local2:80>
    DocumentRoot /tmp
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    ErrorLog /var/log/apache2/error2.log
    LogLevel warn
    CustomLog /var/log/apache2/access2.log combined
    ServerSignature On
</VirtualHost>
```

与local2:80 virtualHost仅有的不相同的是DocumentRoot这一行。在重新定向脚本的输出到一个文件时，两者之间的不同之处在于：

```
jmjones@dinkgutsy:code$ diff apache_conf.diff /etc/apache2/sites-available/psa
20c20
<     DocumentRoot /tmp
---
>     DocumentRoot /var/www2/
```

通过修改Apache配置文件来改变DocumentRoot是非常简单的工作，但是如果不得不经

常改变文档的根，或者有多个虚拟主机需要你切换使用，那么写一个与刚才写的类似的脚本将是非常值得的。但是，这是一个相当简单的脚本。对脚本进行修改以注释掉VirtualHost部分、修改LogLevel指令或是改变VirtualHost存放日志的位置，这些都非常简单。

## 处理文件

学习处理文件的关键是学会如何处理文本数据。经常地，需要处理的文件包含在诸如日志文件、配置文件或应用数据文件这样的文本文件中。当需要对正在分析的数据做进一步深入处理时，通常要创建一个指定类型的报告文件，或是将数据放入一个文本文件以便日后进一步处理。幸运的是，Python包括一个容易使用的称为file的内建类型，该类型可以协助完成所有这些事情。

### 创建文件

这看起来似乎不太合理，但是为了读入一个现有文件，不得不创建一个新的文件对象。但是不要把创建一个文件对象和创建一个文件搞混淆了。对文件进行写入操作需要创建一个新对象，并且需要在磁盘上创建一个新文件，因此与读入文件时创建一个文件对象相比，这似乎更合情合理。创建一个文件对象的目的是让你可以与磁盘上的这个文件进行交互。

要创建一个文件对象，可以使用内建函数open()。下面是一个打开文件进行读取操作的代码示例：

```
In [1]: infile = open("foo.txt", "r")
In [2]: print infile.read()
Some Random
Lines ,
of
Text.
```

由于open是内建函数，不需要使用import进行载入。open()具有三个参数：文件名，文件打开模式，以及缓冲区大小。只有第一个参数（文件名）是必须的。最普通的模式值为“r”（读模式，该值为默认值），“w”（写模式）和“a”（附加模式）。一个补充模式“b”可以被加到其他模式中，该模式表示二进制模式。第三个参数是缓存大小，表示缓存文件的操作方式。

在前一个示例中，我们指定以读模式使用open()打开文件“filefoo.txt”，并且使用变量infile引用新的可读文件对象。一旦创建了infile，就可以自由调用read()方法，读取整个文件的内容。

创建一个用于写入的文件与创建一个用于读取的文件方法类似。写入文件使用“w”标志，而不是“r”标志。

```
In [1]: outputfile = open("foo_out.txt", "w")
In [2]: outputfile.write("This is\nSome\nRandom\nOutput Text\n")
In [3]: outputfile.close()
```

在这个示例中，我们使用`open()`函数以写入方式打开`foo_out.txt`文件，并且使用变量`outputfile`引用新创建的可写文件对象。一旦我们创建了`outputfile`，我们可以使用`write()`将文本写入到文件中，并使用`close()`关闭文件。

由于这是创建文件的非常简单的方式，你或许想有一个容错性更好的创建文件的方式。使用“try/finally”代码块将文件进行封闭是一个非常有用的方法，尤其是在使用`write()`调用时。下面的示例展示了一个在“try/finally”块中被封闭的可写文件：

```
In [1]: try:
...:     f = open('writeable.txt', 'w')
...:     f.write('quick line here\n')
...: finally:
...:     f.close()
```

使用这种写文件的方法，在异常发生时会引起`close()`方法被调用。事实上，即使没有异常发生，`close()`方法也会被包括进去。在`try`块执行完毕之后，不管异常是否发生，`finally`块都会被执行。

在Python2.5中一个新的关键词是`with`语句，它允许使用上下文管理器。一个上下文管理器是一个简单的对象，具有`_enter_()`和`_exit_()`方法。当一个对象在表达式中被创建时，上下文管理器的`_enter_()`方法被调用。当`with`块结束后，即使发生异常，上下文管理器的`_exit_()`方法也会被调用。`File`对象定义了`_enter_()`和`_exit_()`方法。对于`_exit_()`，`File`对象`close()`方法被调用。下面的示例中演示了`with`语句：

```
In [1]: from __future__ import with_statement
In [2]: with open('writeable.txt', 'w') as f:
...:     f.write('this is a writeable file\n')
...:
...:
```

尽管我们没有调用文件对象`f`的`close()`方法，但上下文管理器在退出`with`块时将其关闭。

```
In [3]: f
Out[3]: <closed file 'writeable.txt', mode 'w' at 0x1382770>
```

```
In [4]: f.write("this won't work")
-----
ValueError                                Traceback (most recent call last)
/Users/jmjones/<ipython console> in <module>()
ValueError: I/O operation on closed file
```

正如我们所期望的，文件对象被关闭。这是一种处理可能的异常，并确保file对象被关闭的非常有效的实用方法。出于简化和清晰的目的，我们不会在所有示例中都这么做。

关于file对象可用方法的完整列表，参见《Python Library Reference》中文件对象一节的相关内容 (<http://docs.python.org/lib/bltin-file-objects.html>) 。

## 读取文件

一旦有了用r标志可以打开的可读文件对象，就会得到有三个常用的file方法来取得文件中包含的数据：read()、readline()和readlines()。毫无疑问read()方法可以从一个打开的文件对象读取数据，返回读取的字节数，并返回一个由这些字节组成的字符串对象。read()有一个可选参数，该参数定义读取的字节数。如果没有指定字节数，read()会尽可能读到文件的结尾。如果定义的字节数多于文件的实际字节数，read()会读到文件的结尾，并且返回已经读取的字符数。

假定有下面这样的文件：

```
jmjones@dink:~/some_random_directory$ cat foo.txt Some Random
Lines
of
Text.
```

read()对文件进行处理，如下所示：

```
In [1]: f = open("foo.txt", "r")
In [2]: f.read()
Out[2]: 'Some Random\n Lines\nof \n Text.\n'
```

注意换行被表示成\n，这是创建一个新行的标准方法。

如果仅需要文件的前5个字节，可以这样进行处理：

```
In [1]: f = open("foo.txt", "r")
In [2]: f.read(5)
Out[2]: 'Some '
```

接下来介绍的从一个文件获取文本的方法readline()。readline()方法的目的是次读取文件的一行文本。readline()有一个可选参数：size。size定义readline()在返回一

个字符串对象之前读取的最大字节数，而不论是否达到了行的结尾。因此，在下面的示例中，程序会从文件`foo.txt`读入文本的第一行，然后从第2行读入前7个字节，后面跟着的是第二行的其余文本：

```
In [1]: f = open("foo.txt", "r")
In [2]: f.readline()
Out[2]: 'Some Random\n'
In [3]: f.readline(7)
Out[3]: '    Lin'
In [4]: f.readline()
Out[4]: 'es\n'
```

最后一个要介绍的从文件读取文本的方法是`readlines()`。`readlines()`不是排版错误，也不是之前示例中一个因剪切/粘贴留下的错误。`readlines()`读入文件中的所有行。是的，这就是事实。`readlines()`有一个`sizehint`选项，用于指定读入字符的大约总数。在接下来的示例中，我们创建了一个文件`biglines.txt`，该文件包含了10 000行，每一行包括80个字符。我们打开文件，希望读取文件的前N行（这总共大约1 024个字节），之后读入文件中的其他行：

```
In [1]: f = open("biglines.txt", "r")
In [2]: lines = f.readlines(1024)
In [3]: len(lines)
Out[3]: 102
In [4]: len("".join(lines))
Out[4]: 8262
In [5]: lines = f.readlines()
In [6]: len(lines)
Out[6]: 9898
In [7]: len("".join(lines))
Out[7]: 801738
```

In [3]显示我们读取了102行，In [4]显示这些行总共8262字节。如果实际读取的字节数是8262，那么大约读取的字节数1024是怎么回事？原来内部缓存大约会占8KB。关键问题是`sizehint`并不是总是按着我们认为的方式工作，因此需要时刻注意。

## 写文件

有时需要对文件进行进一步的处理，而不仅是从文件中读取数据。这就需要创建自己的文件并将数据写到里面。为了将数据写入文件，需要掌握两个常用的`file`方法。第一个

方法之前已经演示过，是**write()**。**write()**有一个参数：写入文件的字符串。下面将数据写入文件的示例中使用了**write()**方法：

```
In [1]: f = open("some_writable_file.txt", "w")
In [2]: f.write("Test\nFile\n")
In [3]: f.close()
In [4]: g = open("some_writable_file.txt", "r")
In [5]: g.read()
Out[5]: 'Test\nFile\n'
```

在In [1]，我们使用“w”标志（即可写入方式）打开了一个文件。In [2]向文件内写入了两行数据。In [4]为了避免与之前使用的f混淆，使用变量g作为文件对象。从In [5]可以看到写入文件的内容与我们使用**read()**读取出来的内容是相同的。

接下来介绍的常用数据写入方法是**writelines()**。**writelines()**必须有一个参数：**writelines()**要写入打开文件的序列。该序列可以是任何迭代对象类型，如列表，元组，组合列表（也是列表）或者是发生器。下面是一个发生器表达式**writelines()**的例子，使用**writelines**将数据写入文件：

```
In [1]: f = open("writelines_outfile.txt", "w")
In [2]: f.writelines("%s\n" % i for i in range(10))
In [3]: f.close()
In [4]: g = open("writelines_outfile.txt", "r")
In [5]: g.read()
Out[5]: '0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
```

以下是一个使用发生器函数向文件中写入数据的示例（这在功能上等同于前一示例，但是使用了更多的代码）：

```
In [1]: def myRange(r):
...:     i = 0
...:     while i < r:
...:         yield "%s\n" % i
...:         i += 1
...:
In [2]: f = open("writelines_generator_function_outfile", "w")
In [3]: f.writelines(myRange(10))
In [4]: f.close()
```

```
In [5]: g = open("writelines_generator_function_outfile", "r")
In [6]: g.read()
Out[6]: '0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
```

值得注意的是`writelines()`无法写入一个新行（`\n`），你需要在写入的序列中使用“`\n`”。还有一点需要注意的是使用`writelines()`并非仅能向文件中写入基于行的信息，或许一个更准确的名字应该叫做`writeiter()`。在前面的示例中，我们在写入的文本中加入了换行，但是其实没有必要这么做。

## 额外资源

如果需要了解`file`对象更多的信息，请查阅David Ascher 和Mark Lutz (O'Reilly)编著的《Learning Python》第7章（也可以在线访问<http://safari.oreilly.com/0596002815/lpython2-chp-7-sect-2>）或者查阅《Python Library Reference》的文件对象一章（也可在线访问<http://docs.python.org/lib/bltin-file-objects.html>）。

要了解更多的`generator`表达式的相关内容可以参考《Python Reference Manual》的“generator expression”部分（<http://docs.python.org/ref/genexpr.html>）。为了获得更多有关`yield`语句的相关内容，可以查阅《Python Reference Manual》的`yield`语句一节（可在线访问<http://docs.python.org/ref/yield.html>）。

## 标准输入和输出

在进程的标准输入中读取文本，或是写入到进程的标准输出，这是绝大多数系统管理员熟悉的工作。标准输入只是简单地将数据送入程序，使程序在运行时可以读取。标准输出是程序的输出，由程序在运行时执行写入操作。使用标准输入和标准输出的好处是允许命令与其他工具连接使用。

Python标准库包括一个内建的称为`sys`的模块，该模块提供对标准输入输出的支持。标准库提供对标准输入和输出的访问时，尽管它们没有直接连接到磁盘上的文件，却将它们当作类文件对象对待。既然它们是类文件对象，就可以使用在文件中使用的方法。可以将其视为磁盘上的文件，并且使用合适的方法。

标准输入需要通过加载`sys`模块和引用`stdin`属性（`sys.stdin`）进行访问。`sys.stdin`是一个可读的文件对象。让我们看一下如果先打开一个磁盘上的名为`foo.txt`的文件来创建一个文件对象，然后对比`sys.stdin`与这个打开的文件对象会发生什么。

```
In [1]: import sys
In [2]: f = open("foo.txt", "r")
```

```
In [3]: sys.stdin
Out[3]: <open file '<stdin>', mode 'r' at 0x14020>

In [4]: f
Out[4]: <open file 'foo.txt', mode 'r' at 0x12179b0>

In [5]: type(sys.stdin) == type(f)
Out[5]: True
```

Python解释器将它们视为相同的类型，因此它们使用相同的方法。虽然技术上他们是相同的类型并且使用相同的方法，但类文件对象的一些方法却有不同的行为。例如，`sys.stdin.seek()`和`sys.stdin.tell()`是可用的，但是当你调用时，他们会抛出一个异常(`IOError`)。这里主要强调的是对于类文件对象，你可以像使用基于磁盘的文件一样进行使用。

在Python（或IPython）提示符下使用`sys.stdin`是毫无意义的。加载`sys`或执行`sys.stdin.read()`被永远禁止。为了展示`sys.stdin`如何工作的，我们创建了一个脚本，该脚本从`sys.stdin()`读取数据，然后打印输出带有行号的每一行。参见例3-22。

### 例3-22：枚举`sys.stdin.readline`

```
#!/usr/bin/env python

import sys

counter = 1
while True:
    line = sys.stdin.readline()
    if not line:
        break
    print "%s: %s" % (counter, line)
    counter += 1
```

在这个示例中，我们创建变量`counter`来对行数进行追踪记录。脚本中有一个`while`循环，从标准输入设备读入行。对于每一行，打印行号和行数。在程序循环执行时，该脚本处理所有输入的行，即使其是空行。当然空行也不是彻底为空，因为包括换行符“\n”。当脚本执行到文件结尾时从循环中跳出。

下面是`who`与之前的脚本通过管道连接后输出的结果：

```
jmjones@dink:~/psabook/code$ who | ./sys_stdin_readline.py
1: jmjones console Jul 9 11:01
2: jmjones ttyp1 Jul 9 19:58
3: jmjones ttyp2 Jul 10 05:10
4: jmjones ttyp3 Jul 11 11:51
5: jmjones ttyp4 Jul 13 06:48
```

```
6: jmjones ttys5 Jul 11 21:49
```

```
7: jmjones ttys6 Jul 15 04:38
```

有趣的是，可以通过使用`enumerate`函数将先前的示例写得非常简单短小。参见例3-23。

### 例3-23：`sys.stdin.readline`示例

```
→ #!/usr/bin/env python
```

```
import sys

for i, line in enumerate(sys.stdin):
    print "%s: %s" % (i, line)
```

在加载`sys`模块并且使用`stdin`属性时，可以使用标准输入。通过加载`sys`模块并引用`stdout`属性，可以使用标准输出。`sys.stdin`是一个可读的文件对象，`sys.stdout`是一个可写的文件对象。`sys.stdin`与可读文件对象具有相同的类型，`sys.stdout`与可写文件对象具有相同的类型。

```
→ In [1]: import sys

In [2]: f = open('foo.txt', 'w')

In [3]: sys.stdout
Out[3]: <open file '<stdout>', mode 'w' at 0x14068>

In [4]: f
Out[4]: <open file 'foo.txt', mode 'w' at 0x1217968>

In [5]: type(sys.stdout) == type(f)
Out[5]: True
```

作为一个相关的方面，既然可读文件与可写文件共享相同的类型，最后一点也是理所应当的。

```
→ In [1]: readable_file = open('foo.txt', 'r')

In [2]: writable_file = open('foo_writable.txt', 'w')

In [3]: readable_file
Out[3]: <open file 'foo.txt', mode 'r' at 0x1243530>

In [4]: writable_file
Out[4]: <open file 'foo_writable.txt', mode 'w' at 0x1217968>

In [5]: type(readable_file) == type(writable_file)
Out[5]: True
```

要了解`sys.stdout`类型，首先要弄明的一件重要事情是，它可以采用与处理可写文件相同的方法进行处理，就像`sys.stdin`可以被当作可读文件进行处理一样。

## StringIO

如果写了一个可以处理文件对象的函数，但需要处理的数据是文本字符串而不是文件，遇到这种情况你会不会有些不知所措？针对这个问题一个简单容易的解决方案是加载 `StringIO`：

```
In [1]: from StringIO import StringIO

In [2]: file_like_string = StringIO("This is a\nmultiline string.\nreadline() should see\nmultiple lines of\ninput")

In [3]: file_like_string.readline()
Out[3]: 'This is a\n'

In [4]: file_like_string.readline()
Out[4]: 'multiline string.\n'

In [5]: file_like_string.readline()
Out[5]: 'readline() should see\n'

In [6]: file_like_string.readline()
Out[6]: 'multiple lines of\n'

In [7]: file_like_string.readline()
Out[7]: 'input'
```

在这个示例中，我们创建了一个StringIO对象，并将字符串“`This is a\nmultiline string.\n\nreadline() should see \nmultiple lines of\\ninput`”传递到构造器。我们能够通过StringIO对象调用`readline()`方法。这里，虽然`readline()`是我们调用的唯一方法时，但这绝不意味着这是仅有的可以使用的`file`方法。

```
In [8]: dir(file_like_string)
Out[8]:
['__doc__',
 '__init__',
 '__iter__',
 '__module__',
 'buf',
 'buflist',
 'close',
 'closed',
 'flush',
 'getvalue',
 'isatty',
 'len',
 'next',
 'pos',
 'read',
 'readline',
 'readlines',
 'seek',
 'softspace',
 'tell']
```

```
'truncate',
'write',
'writelines']
```

需要明确的是，虽然两者之间存在差异，但接口允许在文件与字符串之间进行一个简单的转换。下面是file的方法和属性与StringIO对象的方法与属性的对比。

```
In [9]: f = open("foo.txt", "r")
In [10]: from sets import Set
In [11]: sio_set = Set(dir(file_like_string))
In [12]: file_set = Set(dir(f))
In [13]: sio_set.difference(file_set)
Out[13]: Set(['__module__', 'buflist', 'pos', 'len', 'getvalue', 'buf'])
In [14]: file_set.difference(sio_set)
Out[14]: Set(['fileno', '__setattr__', '__reduce_ex__', '__new__', 'encoding',
 '__getattribute__', '__str__', '__reduce__', '__class__', 'name',
 '__delattr__', 'mode', '__repr__', 'xreadlines', '__hash__', 'readinto',
 'newlines'])
```

正如你所看到的，如果需要将一个字符串作为文件来处理，StringIO会很有帮助。

## urllib

如果你有兴趣读入的文件碰巧是在互联网上该怎么办？或者是你希望复用一段代码，而该代码需要一个文件对象？内建的文件类型不知道互联网，但是urllib模块可以提供帮助。

如果想从一个web服务器读取文件，urllib.urlopen()提供了一个简单的解决方法。下面是一个简单的示例：

```
In [1]: import urllib
In [2]: url_file = urllib.urlopen("http://docs.python.org/lib/module-urllib.html")
In [3]: urllib_docs = url_file.read()
In [4]: url_file.close()
In [5]: len(urllib_docs)
Out[5]: 28486
In [6]: urllib_docs[:80]
Out[6]: '<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">\\n<html>\\n<head>\\n<li>
In [7]: urllib_docs[-80:]
Out[7]: 'nt...</a></i> for information on suggesting changes.\\n</address>\\n</body>\\n</html>\\n'
```

首先加载urllib。接下来使用urllib创建一个类文件对象，并命名为url\_file。然后，将url\_file的内容读入到称为urllib\_docs的字符串中。为表明获取的数据实际上来自互联网，我们对获得的文档分割前后80个字符。注意，urllib文件对象支持read()和close()方法，也支持readline()、readlines()、fileno()和geturl()。

如果需要了解更多强大的功能，例如使用代理服务器，你可以在<http://docs.python.org/lib/module-urllib.html>找到更多的关于urllib的相关信息。如果你还需要进一步了解更多内容，例如摘要认证和cookies，可以查阅urllib2：<http://docs.python.org/lib/module-urllib2.html>。

## 日志解析

如果不涉及日志解析，那么从一名系统管理员的角度来讲，对文件处理的介绍是不完整的，所以接下来要对日志解析进行介绍。此时，你已经能够打开一个日志文件，读取其中每一行，并以能够以便捷的方式读取数据。在开始编写这个示例之前，我们不得不问自己“我们希望如何读取日志文件？”答案相当简单：读取一个Apache访问日志，查看每一个独立客户端连接获得的字节数。

根据<http://httpd.apache.org/docs/1.3/logs.html>，这种“整合”的日志格式如下所示：

```
→ 127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0"
  200 2326 "http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I;
  Nav)"
```

这同Apache日志数据相匹配。从日志文件中的每一行可以获得的两个感兴趣的信息：客户端的IP地址和转输的字节数。IP地址是日志文件中的第一个字段，在本例中是127.0.0.1。传输的字节数是IP地址字段之后的两个字段，在引用的前面。在本例中传输了2326字节。关于如何获得该字段，参见例3-24。

例3-24：Apache日志文件解析-以空格字符分隔

```
→ #!/usr/bin/env python
"""
USAGE:

apache_log_parser_split.py some_log_file

This script takes one command line argument: the name of a log file
to parse. It then parses the log file and generates a report which
associates remote hosts with number of bytes transferred to them.

"""

import sys

def dictify_logline(line):
    '''return a dictionary of the pertinent pieces of an apache combined log file
```

```

Currently, the only fields we are interested in are remote host and bytes sent,
but we are putting status in there just for good measure.
...
split_line = line.split()
return {'remote_host': split_line[0],
        'status': split_line[8],
        'bytes_sent': split_line[9],
    }

def generate_log_report(logfile):
    '''return a dictionary of format remote_host=>[list of bytes sent]

This function takes a file object, iterates through all the lines in the file,
and generates a report of the number of bytes transferred to each remote host
for each hit on the webserver.
...
report_dict = {}
for line in logfile:
    line_dict = dictify_logline(line)
    print line_dict
    try:
        bytes_sent = int(line_dict['bytes_sent'])
    except ValueError:
        ##totally disregard anything we don't understand
        continue
    report_dict.setdefault(line_dict['remote_host'], []).append(bytes_sent)
return report_dict
if __name__ == "__main__":
    if not len(sys.argv) > 1:
        print __doc__
        sys.exit(1)
    infile_name = sys.argv[1]
    try:
        infile = open(infile_name, 'r')
    except IOError:
        print "You must specify a valid file to parse"
        print __doc__
        sys.exit(1)
    log_report = generate_log_report(infile)
    print log_report
    infile.close()

```

该示例非常简单。`__main__`部分仅执行了少量处理。首先，对命令行参数进行最少检测，以确保至少传递了一个参数。如果用户没有在命令行传递参数，脚本将打印一个用法信息并退出。关于如何更好地使用命令行及参数，可以参阅第13章，那里有详细的介绍。接下来，`__main__`试图打开指定的日志文件。如果不能成功打开，将显示用法信息并退出。然后将日志文件传递给`generate_log_report()`函数，并打印结果。

`generate_log_report()`创建一个字典，起到报告的作用。它会迭代日志文件中所有的行并将每一行传递给`dictify_logline()`，而该方法返回一个包含所需信息的字典。接下来，它检查`bytes_sent`的值是否是一个整数。如果是，则继续。如果不是，处理下一

行。之后，利用dictfy\_logline()返回的数据升级报告字典。最后，将报告字典返回到\_\_main\_\_。

dictify\_logline()简单地以空格分割较长的行，从结果列表中提取某些项，并返回一个由分割行数据组成的字典。

那么，它起作用了么？基本上起到了作用。请查看单元测试例3-25。

### 例3-25：Apache日志文件的单元测试——以空格字符分隔

```
#!/usr/bin/env python

import unittest
import apache_log_parser_split

class TestApacheLogParser(unittest.TestCase):

    def setUp(self):
        pass

    def testCombinedExample(self):
        # test the combined example from apache.org
        combined_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
            '"GET /apache_pb.gif HTTP/1.0" 200 2326 "http://www.example.com/start.html" '\
            '"Mozilla/4.08 [en] (Win98; I ;Nav)"'
        self.assertEqual(apache_log_parser_split.dictify_logline(combined_log_entry),
                        {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    def testCommonExample(self):
        # test the common example from apache.org
        common_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
            '"GET /apache_pb.gif HTTP/1.0" 200 2326'
        self.assertEqual(apache_log_parser_split.dictify_logline(common_log_entry),
                        {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    def testExtraWhitespace(self):
        # test for extra whitespace between fields
        common_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
            '"GET /apache_pb.gif HTTP/1.0" 200 2326'
        self.assertEqual(apache_log_parser_split.dictify_logline(common_log_entry),
                        {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    def testMalformed(self):
        # test for extra whitespace between fields
        common_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] '\
            '"GET /some/url/with white space.html HTTP/1.0" 200 2326'
        self.assertEqual(apache_log_parser_split.dictify_logline(common_log_entry),
                        {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

if __name__ == '__main__':
    unittest.main()
```

代码对于复合及通用的日志格式都是可以处理的，但是对于请求字段的略微修改则可能导致单员测试的失败。下面是一个测试的结果示例：

```
jmjones@dinkgutsy:code$ python test_apache_log_parser_split.py
...
=====
FAIL: testMalformed (_main_.TestApacheLogParser)
-----
Traceback (most recent call last):
  File "test_apache_log_parser_split.py", line 38, in testMalformed
    {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})
AssertionError: {'status': 'space.html', 'bytes_sent': 'HTTP/1.0',
'remote_host': '127.0.0.1'} != {'status': '200', 'bytes_sent': '2326',
'remote_host': '127.0.0.1'}

-----
Ran 4 tests in 0.001s

FAILED (failures=1)
```

因为数据域中的冒号可以转化为空格，日志文件中的所有域都被向右侧移动一位。严格根据日志格式的说明，提取远端主机名和字节数是相当安全的。这些字节数就是基于以空格为分隔符的字段。例3-26是使用正则表达式的相同示例。

### 例3-26：Apache日志文件解析——regex

```
#!/usr/bin/env python
"""

USAGE:

apache_log_parser_regex.py some_log_file

This script takes one command line argument: the name of a log file
to parse. It then parses the log file and generates a report which
associates remote hosts with number of bytes transferred to them.
"""

import sys
import re
log_line_re = re.compile(r'''(?P<remote_host>\S+) #IP ADDRESS
                           \s+ #whitespace
                           \S+ #remote logname
                           \s+ #whitespace
                           \S+ #remote user
                           \s+ #whitespace
                           \[[^\[\]]+\] #time
                           \s+ #whitespace
                           "[^"]+" #first line of request
                           \s+ #whitespace
                           (?P<status>\d+)
                           \s+ #whitespace
                           (?P<bytes_sent>-|\d+)
                           \s* #whitespace
                           ''', re.VERBOSE)

def dictify_logline(line):
    '''return a dictionary of the pertinent pieces of an apache combined log file
```

```

Currently, the only fields we are interested in are remote host and bytes sent,
but we are putting status in there just for good measure.
...
m = log_line_re.match(line)
if m:
    groupdict = m.groupdict()
    if groupdict['bytes_sent'] == '-':
        groupdict['bytes_sent'] = '0'
    return groupdict
else:
    return {'remote_host': None,
            'status': None,
            'bytes_sent': "0",
    }

def generate_log_report(logfile):
    '''return a dictionary of format remote_host=>[list of bytes sent]

This function takes a file object, iterates through all the lines in the file,
and generates a report of the number of bytes transferred to each remote host
for each hit on the webserver.
...
report_dict = {}
for line in logfile:
    line_dict = dictify_logline(line)
    print line_dict
    try:
        bytes_sent = int(line_dict['bytes_sent'])
    except ValueError:
        ##totally disregard anything we don't understand
        continue
    report_dict.setdefault(line_dict['remote_host'], []).append(bytes_sent)
return report_dict

if __name__ == "__main__":
    if not len(sys.argv) > 1:
        print __doc__
        sys.exit(1)
    infile_name = sys.argv[1]
    try:
        infile = open(infile_name, 'r')
    except IOError:
        print "You must specify a valid file to parse"
        print __doc__
        sys.exit(1)
    log_report = generate_log_report(infile)
    print log_report
    infile.close()

```

从regex示例到以空格为分隔符的示例，我们仅修改了函数dictify\_logline()。这表明在regex示例中，我们放弃了函数的准确返回类型。没有对日志中的每一行以空格进行分割，而是使用了编译后的正则表达式对象log\_line\_re来匹配日志行。如果匹配成

功，返回一个略有修改的groupdict()方法。该方法中当域中包含“-”时bytes\_sent被设置为0，（因为“-”表示空）。在没有任何匹配的情况下，返回一个具有相同关键字的字典，但是其值为空和0。

那么，正则表达式比字符串分割更有优势么？事实上，确实如此。下面的示例是对Apache解析脚本regex的最新版本的一次单元测试。

```
#!/usr/bin/env python

import unittest
import apache_log_parser_regex

class TestApacheLogParser(unittest.TestCase):

    def setUp(self):
        pass

    def testCombinedExample(self):
        # test the combined example from apache.org
        combined_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] \"\
        "GET /apache_pb.gif HTTP/1.0" 200 2326 \"\
        "http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I ;Nav)"'
        self.assertEqual(apache_log_parser_regex.dictify_logline(combined_log_entry),
                        {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    def testCommonExample(self):
        # test the common example from apache.org
        common_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] \"\
        "GET /apache_pb.gif HTTP/1.0" 200 2326'
        self.assertEqual(apache_log_parser_regex.dictify_logline(common_log_entry),
                        {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    def testMalformedEntry(self):
        # test a malformed modification derived from the example at apache.org
        #malformed_log_entry = '127.0.0.1 - frank [10/Oct/2000 13:55:36 -0700] \"\
        #'"GET /apache_pb.gif HTTP/1.0" 200 2326 '\
        #'"http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I ;Nav)"'
        #
        malformed_log_entry = '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] \"\
        "GET /some/url/with white space.html HTTP/1.0" 200 2326'
        self.assertEqual(apache_log_parser_regex.dictify_logline(malformed_log_entry),
                        {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})

    if __name__ == '__main__':
        unittest.main()
```

下面是单元测试的结果：

```
jmjones@dinkgutsy:code$ python test_apache_log_parser_regex.py
...
-----
Ran 3 tests in 0.001s
OK
```

# ElementTree

如果需要解析的文本是XML，那么你需要一种不同于处理面向行的日志文件的处理方式。你可能不想一行接一行地读入文件、查找模式，也不想太多地依赖正则表达式。XML使用一种树结构，因此你并不希望对其进行按行读取。而使用正则表达式来建立一个树状数据结构对于处理任何略微大一点的文件来说，都是一件十分令人头痛的事。

这样的话，还能使用什么呢？有两种典型方法可以处理XML。一种是“simple API for XML，”或者称为SAX。Python标准库中包括了SAX解析器。SAX是一种典型的极为快速的工具，在解析XML时，不会自动占用大量内存。但是这是基于回调机制的，因此在某些数据中，当它命中XML文档的某些部分时（例如开始和结束标签），它会调用某些方法并进行传递。这意味着必须为数据指定句柄，以维持自己的状态，而这是非常困难的。这两件事情使得“simple”在“simple API for XML”中看起来有些牵强。另一个操作XML的方法是使用Document Object Model，或者称为DOM。Python标准库也包括一个DOM XML库。与SAX相比，DOM是典型的比较慢，消耗更多内存的方法，因为DOM会将整个XML树读入到内存中，并为树中的第一个节点建立一个对象。使用DOM的好处是你不需要对状态进行追踪，因为每一个节点都知道谁是它的父节点，谁是它的子节点。但是DOM API使用起来多少有些麻烦。

第三种选择是使用ElementTree（元素树）。ElementTree是XML解析库，已经在Python2.5之后被包括在标准库中。ElementTree感觉就像一个轻量级的DOM，具有方便使用、十分友好的API。除了代码可复用之外，它运行速度快，消耗内存较少。这里我们重点推荐使用ElementTree。如果需要使用XML解析器，不妨先试一试ElementTree。

使用ElementTree开始解析XML文件，只须简单地加载库和使用parse()对文件进行处理：

```
In [1]: from xml.etree import ElementTree as ET  
In [2]: tcusers = ET.parse('/etc/tomcat5.5/tomcat-users.xml')  
In [3]: tcusers  
Out[3]: <xml.etree.ElementTree.ElementTree instance at 0xabb4d0></xml>
```

为了可以在使用库时省去不必要的键盘输入，我们使用名字ET来加载ElementTree模块，这样在使用该库时就可以使用ET，而不用输入全称。接下来，我们告诉ElementTree解析用户的XML文件，该文件来自一个已成功安装的Tomcat servlet引擎。我们称ElementTree对象为tcusers。tcusers的类型是xml.etree.ElementTree.ElementTree。

在删除了授权许可和用法提示之后，可以看到解析后的Tomcat用户文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<tomcat-users>
```

```
<user name="tomcat" password="tomcat" roles="tomcat" />
<user name="role1" password="tomcat" roles="role1" />
<user name="both" password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

ElementTree在解析Tomcat XML文件时创建了一个树对象，我们使用`tcusers`表示，这样就可以获得XML文件中的任何节点。在树对象中，两个最有意义的方法是`find()`和`findall()`。`find()`根据传递给它的查询内容查找匹配查询的第一个节点，并返回一个基于该节点的`element`对象。`findall()`会对匹配查询的所有节点进行查找，并返回一个`element`对象列表，该列表由找到的所有匹配的节点组成。

`find()`和`findall()`寻找的模式的类型都是XPath表达式的有限子集。ElementTree的有效搜索格式包括标签名\*（匹配所有子元素）。。（匹配当前元素）和//（匹配搜索起始点的所有子孙节点）。斜杠（/）可以用来分隔匹配格式。利用Tomcat 用户文件，我们可以使用`find()`提取第一个用户节点的标签名：

```
In [4]: first_user = tcusers.find('/user')
In [5]: first_user
Out[5]: <Element user at abdd88>
```

给`find()`指定搜索格式为“/user”。最前面的斜杠定义了绝对路径，即从根节点开始搜索。文本“user”定义了要查找的标签。因此，`find()`返回标签是“user”的第一个节点。可以看到，引用的对象`first_user`是`element`类型。

一些更有趣的`element`方法和属性包括`Attrib`、`find()`、`findall()`、`get()`、`tag`和`text`。`attrib`是一个元素属性字典。`find()`和`findall()`与ElementTree对象采用相同的处理方式。`get()`是一个字典方法，可以获取指定的属性，如果没有定义属性，返回为空。`attrib`和`get()`都可以为当前的XML标签使用相同的属性字典。`Tag`是包含当前元素标签名的属性。`text`是当前元素作为文本节点时所具有的文本的属性。

以下是一个XML元素ElementTree，是为`first_user`元素对象创建的：

```
<user name="tomcat" password="tomcat" roles="tomcat" />
```

现在调用方法并且引用`tcusers`对象的属性：

```
In [6]: first_user.attrib
Out[6]: {'name': 'tomcat', 'password': 'tomcat', 'roles': 'tomcat'}
In [7]: first_user.get('name')
Out[7]: 'tomcat'
In [8]: first_user.get('foo')
```

```
In [9]: first_user.tag  
Out[9]: 'user'  
  
In [10]: first_user.text
```

至此，你已经看到了一些ElementTree如何使用的基本示例。接下来看一个稍微深入且更通用的示例。我们解析Tomcat用户文件并且搜索name属性匹配我们指定内容（在本例中为'tomcat'）的用户节点（参见例3-27）。

### 例3-27：ElementTree解析Tomcat用户文件

```
#!/usr/bin/env python  
  
from xml.etree import ElementTree as ET  
  
if __name__ == '__main__':  
    infile = '/etc/tomcat5.5/tomcat-users.xml'  
    tomcat_users = ET.parse(infile)  
    for user in [e for e in tomcat_users.findall('/user') if  
        e.get('name') == 'tomcat']:  
        print user.attrib
```

示例中，我们只是使用了一个复合列表来匹配name属性。运行这个示例将返回下面的结果：

```
jmjones@dinkgutsy:code$ python elementtree_tomcat_users.py  
{'password': 'tomcat', 'name': 'tomcat', 'roles': 'tomcat'}
```

最后是一个ElementTree示例，该示例用来从一个写得不是很好的XML中提取信息。Mac OS X有一个称为system\_profiler的工具，可以显示系统的大量信息。XML是system\_profiler支持的两种输出格式之一，但是对XML的支持似乎来得晚一些。希望被提取出来的信息内容是操作系统的版本号，包括在类似下面这样的XML文件中：

```
<dict>  
  <key>_dataType</key>  
  <string>SPSoftwareDataType</string>  
  <key>_detailLevel</key>  
  <integer>-2</integer>  
  <key>_items</key>  
  <array>  
    <dict>  
      <key>_name</key>  
      <string>os_overview</string>  
      <key>kernel_version</key>  
      <string>Darwin 8.11.1</string>  
      <key>os_version</key>  
      <string>Mac OS X 10.4.11 (8S2167)</string>  
    </dict>  
  </array>
```

为什么我们认为这个XML格式写得不规范呢？因为在任何一个XML标签中都没有属性。标签类型是主要的数据类型。一些元素，如可替换的key、string标签，位于相同的父节点下（参见例3-28）。

#### 例3-28：Mac OS X system\_profiler输出解析

```
#!/usr/bin/env python
import sys

from xml.etree import ElementTree as ET
e = ET.parse('system_profiler.xml')
if __name__ == '__main__':
    for d in e.findall('/array/dict'):
        if d.find('string').text == 'SPSoftwareDataType':
            sp_data = d.find('array').find('dict')
            break
    else:
        print "SPSoftwareDataType NOT FOUND"
        sys.exit(1)

record = []
for child in sp_data.getchildren():
    record.append(child.text)
    if child.tag == 'string':
        print "%-15s -> %s" % tuple(record)
        record = []
```

基本上，脚本搜索dict标签，该标签有一个字符串子元素，它的文本值为“SPSoftwareDataType”。脚本搜索的信息在该节点下。在这个示例中，我们之前没有介绍过的唯一内容是getchildren()方法。该方法可以简单地返回一个指定元素的子节点列表。另外，该示例条理非常清晰，XML也写得更好一些。下面是该脚本运行在Mac OS X Tiger笔记本上产生的输出结果：

```
dink:~/code jmjones$ python elementtree_system_profile.py
_name          -> os_overview
kernel_version -> Darwin 8.11.1
os_version      -> Mac OS X 10.4.11 (8S2167)
```

ElementTree是Python标准库的有力补充。到目前为止，我们已经多次使用，并且对使用它所带来的好处非常满意。可以尝试一下Python标准库中的SAX和DOM库，但是我们认为在你尝试之后仍会回来选择使用ElementTree。

## 本章小结

本章讲述了在Python中操作文本的一些基本技术。我们使用了来自标准库的内建string类型、正则表达式、标准输入输出、StringIO以及urllib模块。然后对其中的一些技术

进行联合，应用到两个解析Apache日志文件的示例中。最后介绍了ElementTree库的一些经典用法，并通过两个示例演示了实际应用效果。

一些UNIX使用者认为当复杂的文本处理已经超出了使用grep或awk所能应对的程度时，这种情况下，他们只考虑选用Perl作为更高级的替代工具。Perl是一个功能极为强大的语言，特别是在文本处理方面，然而我们认为Python具有与Perl一样优秀的性能。事实上，如果看到Python简单的语法，以及可以轻松实现由面向过程的编码方式到面向对象的编码方式的转变，你会认为Python比Perl更为优秀，甚至在文本处理方面同样如此。因此，我们希望你下次从事文本处理工作时，首先选择Python。

# 文档与报告

你可能会发现，工作中最单调乏味的事情就是根据用户的需要，对各种各样的信息进行归档。这可能是为了那些希望阅读文档的用户的直接利益，也可能是为了用户的间接利益，你或是你的继任者都或许会在将来对程序实施改进时用到这些文档。因此，无论哪种情况，创建文档都是工作中一个十分重要的内容。但是，如果发现这不是自己喜欢做的工作，就会让人十分烦恼。现在，Python可以帮助我们完成与文档相关的一些工作。尽管Python无法写文档，但是它可以帮助你搜集、格式化这些文档，并将信息发送到需要它们的人手中。

在本章中，我们将集中介绍如何对写的程序进行信息采集、格式化和发布。你感兴趣的共享信息会被保存到某些地方，或许是日志文件的某个位置，或许是头脑中，或许是执行某个shell命令的结果中，也有可能是在某处的数据库中。首先，要对信息进行采集。接下来，为了能够有效地共享信息，需要按照某种方式对数据进行格式化，使其更有意义。数据的格式可以是PDF、PNG、JPG、HTML或纯文本。最后，需要将这些信息送到感兴趣的人手中。感兴趣的人将会通过收取电子邮件，访问一个网站，或者直接通过共享驱动器来查找文件，但哪种方式最方便呢？

## 自动信息收集

信息共享的第一步就是收集信息。本书已在另外两章对数据采集进行了介绍，分别是：文本（第3章）和SNMP（第7章）。文本一章中包括了一些示例，这些示例演示了从一个大文本文件中解析和提取各种数据的方法。第3章中一个典型的示例，就是对Apache web服务器日志中的每一行进行解析，包括解析客户端IP地址、传输字节数以及HTTP状态代码。而第7章，包含了系统信息查询的示例，包括查询RAM容量以及网络接口的速度等。

比较而言，信息收集所涉及的内容，比定位和提取某些数据更为深入一些。大多数情况下，信息收集都包含了从某种格式中提取信息，例如Apache logfile，并以一个中间格式

进行保存以备将来使用。例如，如果想创建一个图表，显示一个月内每一个唯一IP地址从指定的Apache web服务器下载信息的字节数。这一过程中，信息收集阶段可能包括每晚解析Apache日志，提取必要的信息（在本例中，就是IP地址信息和每一个请求所发送的字节数），并将这些数据保存起来，这样以后就可以打开使用。这些需要保存的数据可以存储在关系数据库、对象数据库、pickle文件、CSV文件和纯文本文件中。

这一节的剩余部分试图将文本处理和数据持久性中的多个概念进行整合，并特别阐明这种整合是如何构建在第3章的数据提取以及第12章的数据存储之上。我们将使用与文本处理中相同的库，也将使用shelve模块来保存每个来自HTTP客户端的HTTP请求。在第12章中会对shelve模块进行介绍。

下面是一个简单的模块示例，使用了在前一章中创建的Apache日志文件解析模块，以及shelve模块：

```
#!/usr/bin/env python

import shelve
import apache_log_parser_regex

logfile = open('access.log', 'r')
shelve_file = shelve.open('access.s')

for line in logfile:
    d_line = apache_log_parser_regex.dictify_logline(line)
    shelve_file[d_line['remote_host']] = \
        shelve_file.setdefault(d_line['remote_host'], 0) + \
        int(d_line['bytes_sent'])

logfile.close()
shelve_file.close()
```

该示例首先加载了shelve和apache\_log\_parser\_regex。shelve是一个来自Python标准库的模块。apache\_log\_parser\_regex是我们在第3章中编写的模块。之后，打开apache日志文件，access.log和一个框架文件access.s。对日志文件中的每一行进行迭代处理，使用Apache日志文件解析模块为每一行创建一个字典。该字典包括HTTP请求的状态代码，客户端IP地址，以及传输到客户端的字节数。将特定请求的字节数加到总字节数中，总字节数已经在shelve对象中为每一个客户端IP地址都进行了计算。如果在shelve对象中没有该客户端IP地址的记录，则总字节数自动归0。在对日志文件中的所有行都进行了迭代之后，关闭日志文件和shelve对象。本章后面介绍信息格式化的部分将继续使用这个示例。

## 收取邮件

你或许从未考虑过将收取邮件作为信息收集的一种方式，而事实上，收取邮件确实可以

实现信息收集。想象一下你有一些服务器，这些服务器很难彼此直接连接在一起，但是每一台服务器都支持邮件功能。如果有一个监测这些服务器上web应用的脚本，该脚本可以每隔几分钟就写入或读取日志，那么，就可以利用email作为信息传递的机制。不管写入或读取日志成功与否，都发送一个email，email中记录了成功或失败的信息。可以收集这些email信息来生成报告，或者在出现服务器故障时随时进行替换。

IMAP和POP3是两个最常见的收取邮件协议。在Python“连电池都包括在内”的风格下，Python标准库提供了对这两个协议的支持。

POP3也许保存在指定的服务器上更常见些，使用poplib访问POP3的email非常容易。例4-1显示了如何使用poplib来收取所有的email并将它们写入一系列磁盘文件。

#### 例4-1：通过POP3收取email

```
#!/usr/bin/env python

import poplib

username = 'someuser'
password = 'S3Cr37'

mail_server = 'mail.somedomain.com'

p = poplib.POP3(mail_server)
p.user(username)
p.pass_(password)
for msg_id in p.list()[1]:
    print msg_id
    outf = open('%s.eml' % msg_id, 'w')
    outf.write('\n'.join(p.retr(msg_id)[1]))
    outf.close()
p.quit()
```

正如你所看到的，我们首先定义了username，password和mail\_server，之后连接到邮件服务器，并给出了已定义的用户名和密码。假设用户名和密码正确，那么就获得了在该账户下查看email的权限。迭代处理邮件文件列表，依次读取邮件并将其写入磁盘。该脚本没有完成在收取邮件之后对邮件的删除操作。删除邮件只需要retr()之后调用delete()。

IMAP几乎与POP3一样简单，但是在Python标准库文档中没能充分地介绍。例4-2显示了IMAP代码，该代码能够完成POP3示例中相同的功能。

#### 例4-2：通过IMAP收取email

```
#!/usr/bin/env python

import imaplib

username = 'some_user'
password = '70P53Cr37'
```

```
mail_server = 'mail_server'

i = imaplib.IMAP4_SSL(mail_server)
print i.login(username, password)
print i.select('INBOX')
for msg_id in i.search(None, 'ALL')[1][0].split():
    print msg_id
    outf = open('%s.eml' % msg_id, 'w')
    outf.write(i.fetch(msg_id, '(RFC822)')[1][0][1])
    outf.close()
i.logout()
```

正如在POP3示例中所做的操作，我们在脚本开始处定义了username, password和mail\_server，之后通过SSL连接到IMAP服务器，接下来登录并设置email的目录为INBOX，然后对整个目录进行迭代搜索。search()方法在Python标准库文件中只有少量的描述。字符集和搜索准则是Search()的两个必要参数。什么是一个有效的字符集？需要什么格式？我们怀疑阅读IMAP RFC是否会有帮助，但幸运的是，有大量关于IMAP示例的文档可供参考。对于每一次循环迭代，我们将邮件的内容写入到磁盘。这里会出现一个警告信息：该操作要求目录下的邮件标识为“可读”。这对你来说或许根本不是什么问题，如果删除这一信息也没什么大不了，但是还是注意一下这些信息为好。

## 手工信息收集

接下来，让我们看一下更为复杂的手工收集信息的方法。手工收集信息意味着对我们依靠眼睛和手中的按键对内容进行收集。例如服务器的列表，其中包含每台服务器所对应的IP地址和功能；联系人的列表，其中包含email地址，电话号码，IM屏幕名；或是你的团队成员计划休假的日期等。实际上，已经有一些工具能够管理这类信息。尽管不能实现完全管理，但也可以管理大多数这类信息。例如，Excel或OpenOffice的电子表格（Spreadsheet）就能够用于管理服务器列表。Outlook或Address Book.app就能够用于管理联系人。无论是Excel/OpenOffice电子表格还是Outlook，都可以对人员休假日期进行管理。如果是使用纯文本编辑数据，并具有可配置的、支持HTML的输出（或XHTML），这或许就是解决这一问题的方案。

虽然有很多选择，但是我们将在这里推荐一种可替换的、专门的纯文本格式，即reStructuredText（也称作reST）。下面是reStructuredText网站对其进行的描述：

reStructuredText是一个易于读取，所见即所得的纯文本标记语法和解析系统。对于行内程序归档非常有用（例如Python中的docstrings），可以快速地创建简单的web页面和独立的文档。reStructuredText为扩展特定应用领域而设计。reStructuredText解析器是一个Docutils组件。reStructuredText是对StructuredText和Setext这样的轻量级标记系统的修订和再解析。

## 名人简介：RESTLESS

### Aaron Hillegass



Aaron Hillegass，曾在NeXT和Apple任职，是Mac系统下的应用开发专家。他是*Cocoa Programming for Mac OS X (Big Nerd Ranch)*一书的作者。在Big Nerd Ranch教授Cocoa编程。

请从该书的代码资源库<http://www.oreilly.com/9780596515829>下载ReSTless的全部代码。下面是如何在Cocoa应用中调用Python脚本的示例：

```
#import "MyDocument.h"

@implementation MyDocument

- (id)init
{
    if (![super init])
        return nil;
}

// What you see for a new document
textStorage = [[NSTextStorage alloc] init];
return self;
}

- (NSString *)windowNibName
{
    return @"MyDocument";
}

- (void)prepareEditView
{
    // The layout manager monitors the text storage and
    // layout the text in the text view
    NSLayoutManager *lm = [editView layoutManager];

    // Detach the old text storage
    [[editView textStorage] removeLayoutManager:lm];

    // Attach the new text storage
    [textStorage addLayoutManager:lm];
}

- (void>windowControllerDidLoadNib:(NSWindowController *) aController
{
    [super windowControllerDidLoadNib:aController];

    // Show the text storage in the text view
    [self prepareEditView];
}
```

```

#pragma mark Saving and Loading

// Saves (the URL is always a file:)
- (BOOL)writeToURL:(NSURL *)absoluteURL
    ofType:(NSString *)typeName
    error:(NSError **)outError;
{
    return [[textStorage string] writeToURL:absoluteURL
        atomically:NO
        encoding:NSUTF8StringEncoding
        error:outError];
}

// Reading (the URL is always a file:)
- (BOOL)readFromURL:(NSURL *)absoluteURL
    ofType:(NSString *)typeName
    error:(NSError **)outError
{
    NSString *string = [NSString stringWithContentsOfURL:absoluteURL
        encoding:NSUTF8StringEncoding
        error:outError];
    // Read failed?
    if (!string) {
        return NO;
    }
    [textStorage release];
    textStorage = [[NSTextStorage alloc] initWithString:string
        attributes:nil];
    // Is this a revert?
    if (editView) {
        [self prepareEditView];
    }
    return YES;
}

#pragma mark Generating and Saving HTML

- (NSData *)dataForHTML
{
    // Create a task to run rst2html.py
    NSTask *task = [[NSTask alloc] init];

    // Guess the location of the executable
    NSString *path = @"/usr/local/bin/rst2html.py";

    // Is that file missing? Try inside the python framework
    if (![[NSFileManager defaultManager] fileExistsAtPath:path]) {
        path = @"/Library/Frameworks/Python.framework/Versions/Current/bin/rst2html.py";
    }
    [task setLaunchPath:path];

    // Connect a pipe where the ReST will go in
    NSPipe *inPipe = [[NSPipe alloc] init];
    [task setStandardInput:inPipe];
    [inPipe release];
}

```

```
// Connect a pipe where the HMTL will come out
NSPipe *outPipe = [[NSPipe alloc] init];
[task setStandardOutput:outPipe];
[outPipe release];

// Start the process
[task launch];

// Get the data from the text view
NSData *inData = [[textStorage string] dataUsingEncoding:NSUTF8StringEncoding];

// Put the data in the pipe and close it
[[inPipe fileHandleForWriting] writeData:inData];
[[inPipe fileHandleForWriting] closeFile];

// Read the data out of the pipe
NSData *outData = [[outPipe fileHandleForReading] readDataToEndOfFile];

// All done with the task
[task release];

return outData;
}

- (IBAction)renderRest:(id)sender
{
    // Start the spinning so the user feels like waiting
    [progressIndicator startAnimation:nil];

    // Get the html as an NSData
    NSData *htmlData = [self dataForHTML];

    // Put the html in the main WebFrame
    WebFrame *wf = [webView mainFrame];
    [wf loadData:htmlData
        MIMEType:@"text/html"
        textEncodingName:@"utf-8"
        baseURL:nil];

    // Stop the spinning so the user feels done
    [progressIndicator stopAnimation:nil];
}

// Triggered by menu item
- (IBAction)startSavePanelForHTML:(id)sender
{
    // Where does it save by default?
    NSString *restPath = [self fileName];
    NSString *directory = [restPath stringByDeletingLastPathComponent];
    NSString *filename = [[[restPath lastPathComponent]
        stringByDeletingPathExtension]
        stringByAppendingPathExtension:@"html"];

    // Start the save panel
    NSSavePanel *sp = [NSSavePanel savePanel];
```

```

[sp setRequiredFileType:@"html"];
[sp setCanSelectHiddenExtension:YES];
[sp beginSheetForDirectory:directory
    file:filename
    modalForWindow:[editView window]
    modalDelegate:self
    didEndSelector:@selector(htmlSavePanel:endedWithCode:context:)
    contextInfo:NULL];
}

// Called when the save panel is dismissed
- (void)htmlSavePanel:(NSSavePanel *)sp
    endedWithCode:(int)returnCode
    context:(void *)context
{
    // Did the user hit Cancel?
    if (returnCode != NSOKButton) {
        return;
    }

    // Get the chosen filename
    NSString *savePath = [sp filename];

    // Get the HTML data
    NSData *htmlData = [self dataForHTML];

    // Write it to the file
    NSError *writeError;
    BOOL success = [htmlData writeToFile:savePath
        options:NSAtomicWrite
        error:&writeError];

    // Did the write fail?
    if (!success) {

        // Show the user why
        NSAlert *alert = [NSAlert alertWithError:writeError];
        [alert beginSheetModalForWindow:[editView window]
            modalDelegate:nil
            didEndSelector:NULL
            contextInfo:NULL];
        return;
    }
}

#pragma mark Printing Support

- (NSPrintOperation *)printOperationWithSettings:(NSDictionary *)printSettings
    error:(NSError **)outError
{
    // Get the information from Page Setup
    NSPrintInfo *printInfo = [self printInfo];

    // Get the view that displays the whole HTML document
    NSView *docView = [[[webView mainFrame] frameView] documentView];
}

```

```
// Create a print operation
return [NSPrintOperation printOperationWithView:docView
                                         printInfo:printInfo];
}

@end
```

ReST是Python文档优先使用的格式。如果创建了一个包含了代码的Python包，并打算上传到PyPI，最好用reStructuredText作为文档的格式。由于归档的需要，许多独立的Python项目也使用ReST作为主要的文档格式。

那么，我们为什么要用ReST作为文档格式呢？首先，这种格式并不复杂。其次，标记几乎可以立即掌握。当你看到文档的结构，会很快理解作者的意图。下面是一个非常简单的ReST文件示例：

```
=====
Heading
=====
SubHeading
-----
This is just a simple
little subsection. Now,
we'll show a bulleted list:

- item one
- item two
- item three
```

你或许已经理解了一些基本结构，不需要再去阅读构成一个有效的reStructuredText文件需要些什么之类的内容了。但是你还不能写ReST文件，只是已经可以一行接一行地进行阅读了。

第三，从ReST转换到HTML非常简单。这也是我们将在本节集中介绍的第三个方面。这里不会给出一个reStructuredText的培训指导。如果想要快速地浏览标记语法，可以访问<http://docutils.sourceforge.net/docs/user/rst/quickref.html>。

使用刚刚演示的ReST文档，下面的示例展示了将ReST转换为HTML的步骤：

```
In [2]: import docutils.core
In [3]: rest = '''=====
...: Heading
...: =====
...: SubHeading
...: -----
...: This is just a simple
```

```
....: little subsection. Now,  
....: we'll show a bulleted list:  
....:  
....: - item one  
....: - item two  
....: - item three  
....:  
In [4]: html = docutils.core.publish_string(source=rest, writer_name='html')  
  
In [5]: print html[html.find('<body>') + 6:html.find('</body>')]  
  
<div class="document" id="heading">  
<h1 class="title">Heading</h1>  
<h2 class="subtitle" id="subheading">SubHeading</h2>  
<p>This is just a simple  
little subsection. Now,  
we'll show a bulleted list:</p>  
<ul class="simple">  
<li>item one</li>  
<li>item two</li>  
<li>item three</li>  
</ul>  
</div>
```

整个过程十分简单。首先加载`docutils.core`。然后定义了一个包含`reStructuredText`的字符串，再通过`docutils.core.publish_string()`运行字符串，并将它格式化为HTML。最后，我们做了一个字符串分割，提取在`<body>`和`</body>`标记之间的文本。我们分割`div`区域是因为`docutils`（进行HTML转换用到的库）在产生的HTML页面中嵌入样式表（stylesheet），以使转换的HTML页面看起来不是太平淡。

现在，举一个与系统管理员更为相关的例子。每一个好的系统管理需要对服务器以及服务器上的任务进行追踪。下面是一个示例，演示了创建一个纯文本（plain-text）服务列表，并将其转换为HTML。

```
In [6]: server_list = '''===== ===== =====  
...: Server Name      IP Address   Function  
...: ===== ===== =====  
...: card            192.168.1.2  mail server  
...: vinge           192.168.1.4  web server  
...: asimov          192.168.1.8  database server  
...: stephenson       192.168.1.16 file server  
...: gibson          192.168.1.32 print server  
...: ===== ===== ====='''  
  
In [7]: print server_list  
===== ===== =====  
Server Name      IP Address   Function  
===== ===== =====  
card            192.168.1.2  mail server  
vinge           192.168.1.4  web server  
asimov          192.168.1.8  database server
```

```
stephenson      192.168.1.16  file server
gibson         192.168.1.32  print server
===== ===== =====

In [8]: html = docutils.core.publish_string(source=server_list,
writer_name='html')

In [9]: print html[html.find('<body>') + 6:html.find('</body>')]

<div class="document">
<table border="1" class="docutils">
<colgroup>
<col width="33%" />
<col width="29%" />
<col width="38%" />
</colgroup>
<thead valign="bottom">
<tr><th class="head">Server Name</th>
<th class="head">IP Address</th>
<th class="head">Function</th>
</tr>
</thead>
<tbody valign="top">
<tr><td>card</td>
<td>192.168.1.2</td>
<td>mail server</td>
</tr>
<tr><td>vinge</td>
<td>192.168.1.4</td>
<td>web server</td>
</tr>
<tr><td>asimov</td>
<td>192.168.1.8</td>
<td>database server</td>
</tr>
<tr><td>stephenson</td>
<td>192.168.1.16</td>
<td>file server</td>
</tr>
<tr><td>gibson</td>
<td>192.168.1.32</td>
<td>print server</td>
</tr>
</tbody>
</table>
</div>
```

另外一个非常好的纯文本标记模式是Textile。根据其网站的说明，Textile将纯文本替换为\*simple\*标记，产生有效的XHTML。在一些web应用中，例如内容管理系统、博客以及在线论坛中Texttile被广泛使用。那么，如果Textile是一种标记语言，为什么我们在这本关于Python的书中介绍它呢？原因是Python库允许处理Textile标记，并将其转换为XHTML。可以编写命令行工具来调用Python库，并转换Textile文件，然后重定向输

出到XHTML文件中。或者可以在一些脚本中调用Textile转换模块，并编程处理返回的 XHTML。无论怎样做，Textile标记和Textile处理模块都可以根据归档的需求为你带来巨大的帮助。

可以使用easy\_install textile安装Textile Python模块。也可以使用系统中的打包系统（如果已经安装了）来完成安装。对于Ubuntu，包名是python-textile，可以使用apt-get install python-textile命令安装。一旦被安装，就可以通过简单地加载创建一个Textiler对象，并且在该对象上调用一个简单的方法来开始使用Textile。下面示例代码演示了如何将一个Textile的符号列表转换为XHTML：

```
► In [1]: import textile  
  
In [2]: t = textile.Textiler(''*' item one  
...: * item two  
...: * item three''')  
  
In [3]: print t.process()  
<ul>  
<li>item one</li>  
<li>item two</li>  
<li>item three</li>  
</ul>
```

我们在这里不会对Textile进行教学。在web上有大量这方面的资源。例如<http://hobix.com/textile/>提供了许多与使用Textile相关的不错的参考资料。由于不会过多地对Textile的ins和outs进行介绍，我们会演示Textile的一个手工收集数据的示例。示例中收集的信息之前也说过，是一个包括IP地址和功能的服务器列表：

```
► In [1]: import textile  
  
In [2]: server_list = '''|_. Server Name|_. IP Address|_. Function|  
....|card|192.168.1.2|mail server|  
....|vinge|192.168.1.4|web server|  
....|asimov|192.168.1.8|database server|  
....|stephenson|192.168.1.16|file server|  
....|gibson|192.168.1.32|print server|'''  
  
In [3]: print server_list  
|_. Server Name|_. IP Address|_. Function|  
|card|192.168.1.2|mail server|  
|vinge|192.168.1.4|web server|  
|asimov|192.168.1.8|database server|  
|stephenson|192.168.1.16|file server|  
|gibson|192.168.1.32|print server|  
  
In [4]: t = textile.Textiler(server_list)  
  
In [5]: print t.process()  
<table>  
<tr>  
<th>Server Name</th>
```

```
<th>IP Address</th>
<th>Function</th>
</tr>
<tr>
<td>card</td>
<td>192.168.1.2</td>
<td>mail server</td>
</tr>
<tr>
<td>vinge</td>
<td>192.168.1.4</td>
<td>web server</td>
</tr>
<tr>
<td>asimov</td>
<td>192.168.1.8</td>
<td>database server</td>
</tr>
<tr>
<td>stephenson</td>
<td>192.168.1.16</td>
<td>file server</td>
</tr>
<tr>
<td>gibson</td>
<td>192.168.1.32</td>
<td>print server</td>
</tr>
</table>
```

可以看到，使用ReST和Textile都可以有效地整合将纯文本数据转换为Python脚本的功能。如果确实有一些数据需要转换为HTML，例如服务器列表，联系人列表，然后在这些数据之上完成一些操作（如将HTML依据接收者列表通过email进行发送，或是将HTML通过FTP传输到某处的web服务器），那么docutils或者Textile库都是非常有用的工具。

## 信息格式化

将信息交到用户手中之前，需要对信息进行格式化，将其转换为一种更为容易读取和识别的格式。这些格式应具有容易被用户理解的特点，如果还同时具有很强的吸引力，那就更好了。从技术上说，ReST和Textile包括对共享数据的收集和格式化两个步骤，但是，接下来的示例主要集中在对已经采集的数据进行转换方面，即如何将这些数据转换为一种更具表达力的格式。

## Graphical Images

接下来的两个示例将继续前面的内容，即解析Apache日志文件的客户端IP地址和传输

的字节数。前一章中的示例产生了一个shelve文件，其中包括了一些我们希望与其他用户共享的信息。现在，我们根据shelve文件创建一个图表对象，实现更方便地阅读这些数据：

```
#!/usr/bin/env python

import gdchart
import shelve

shelve_file = shelve.open('access.s')
items_list = [(i[1], i[0]) for i in shelve_file.items()]
items_list.sort()
bytes_sent = [i[0] for i in items_list]
#ip_addresses = [i[1] for i in items_list]
ip_addresses = ['XXX.XXX.XXX.XXX' for i in items_list]

chart = gdchart.Bar()
chart.width = 400
chart.height = 400
chart.bg_color = 'white'
chart.plot_color = 'black'
chart.xtitle = "IP Address"
chart.ytitle = "Bytes Sent"
chart.title = "Usage By IP Address"
chart.setData(bytes_sent)
chart.setLabels(ip_addresses)
chart.draw("bytes_ip_bar.png")

shelve_file.close()
```

在这个示例中，首先加载了两个模块，`gdchart`和`shelve`。之后打开了在前一个示例中创建的`shelve`文件。`shelve`对象类似内建的字典对象，我们能够调用其中的`Items()`方法。`Items()`返回一个元组列表，这个元组中的第一个元素就是字典关键字，第二个元素是关键字的值。`Items()`方法能够按着使数据更有意义的方式来对数据进行排序。使用一个列表对之前的元组进行反向排序。现在元组的内容由`(ip_address, bytes_sent)`变成了`(bytes_sent, ip_addresses)`。然后对这个列表进行排序，由于`bytes_sent`是第一个元素，`list.sort()`方法会按该字段进行排序。再次使用复合列表填充`bytes_sent`和`ip_addresses`字段。你或许已经注意到了，我们插入`XXX.XXX.XXX.XXX`，`XXX`代替具体IP地址，因为这是从一个实际的服务器上获取的日志文件。

在获得用于生成图表的数据之后，使用`gdchart`来制作一个数据的图形表示。首先创建一个图表对象`gdchart.Bar`，为其设置一些属性，并生成一个PNG文件。之后，以像素为单位定义图表的大小。使用冒号来设置背景、前景并创建标题。为图表设置数据和标签，这两项都可以从Apache日志文件解析模块中获得。最后，使用`draw()`来绘制图表，并输出到文件，然后关闭`shelve`对象。图4-1显示了生成的图表。

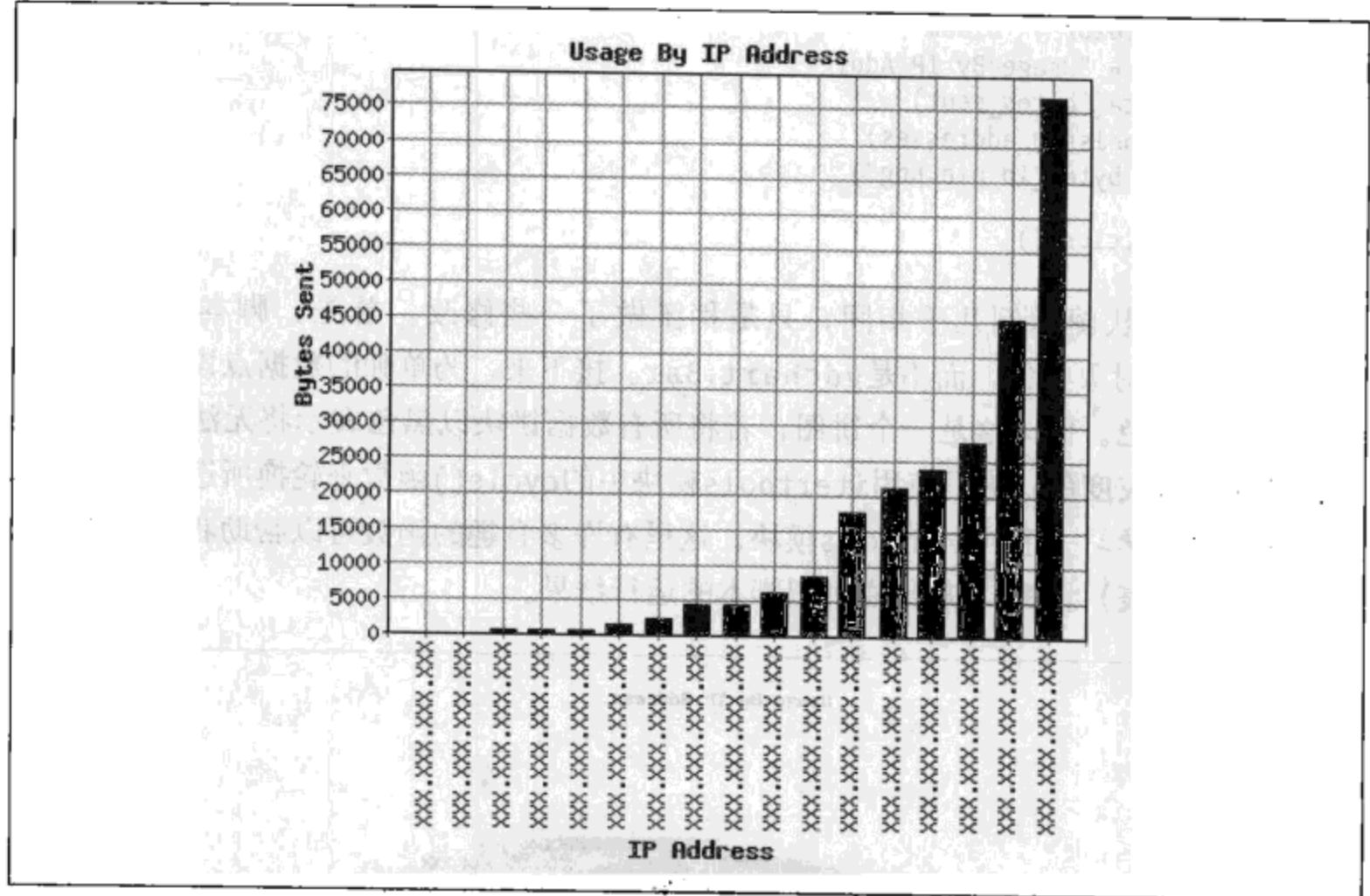


图4-1：每个IP地址请求字节数的柱状图

下面是另一个可视的格式化shelve数据脚本示例，但是这次没有使用柱状图，而是使用了饼图：

```
#!/usr/bin/env python

import gdchart
import shelve
import itertools

shelve_file = shelve.open('access.s')
items_list = [(i[1], i[0]) for i in shelve_file.items() if i[1] > 0]
items_list.sort()
bytes_sent = [i[0] for i in items_list]
#ip_addresses = [i[1] for i in items_list]
ip_addresses = ['XXX.XXX.XXX.XXX' for i in items_list]

chart = gdchart.Pie()
chart.width = 800
chart.height = 800
chart.bg_color = 'white'
color_cycle = itertools.cycle([0xDDDDDD, 0x111111, 0x777777])
color_list = []
for i in bytes_sent:
    color_list.append(color_cycle.next())
chart.color = color_list
```

```
chart.plot_color = 'black'
chart.title = "Usage By IP Address"
chart.setData(*bytes_sent)
chart.setLabels(ip_addresses)
chart.draw("bytes_ip_pie.png")

shelve_file.close()
```

这个脚本与柱状图示例几乎相同，只是稍微做了一些修改。首先，脚本创建了一个`gdchart.Pie`对象实例，而不是`gdchart.Bar`。接下来，为单独的数据点设置颜色，而不是都使用黑色。因为这是一个饼图，若将所有数据饼块以黑色表示将无法阅读，因此轮流使用三种灰度色。这里使用`itertools`模块中的`cycle()`函数来轮换所选择的三种颜色。我们建议学习一下`itertools`模块。这里有许多有趣的函数可以帮助我们处理迭代对象（例如列表）。图4-2是生成饼图脚本的运行结果。

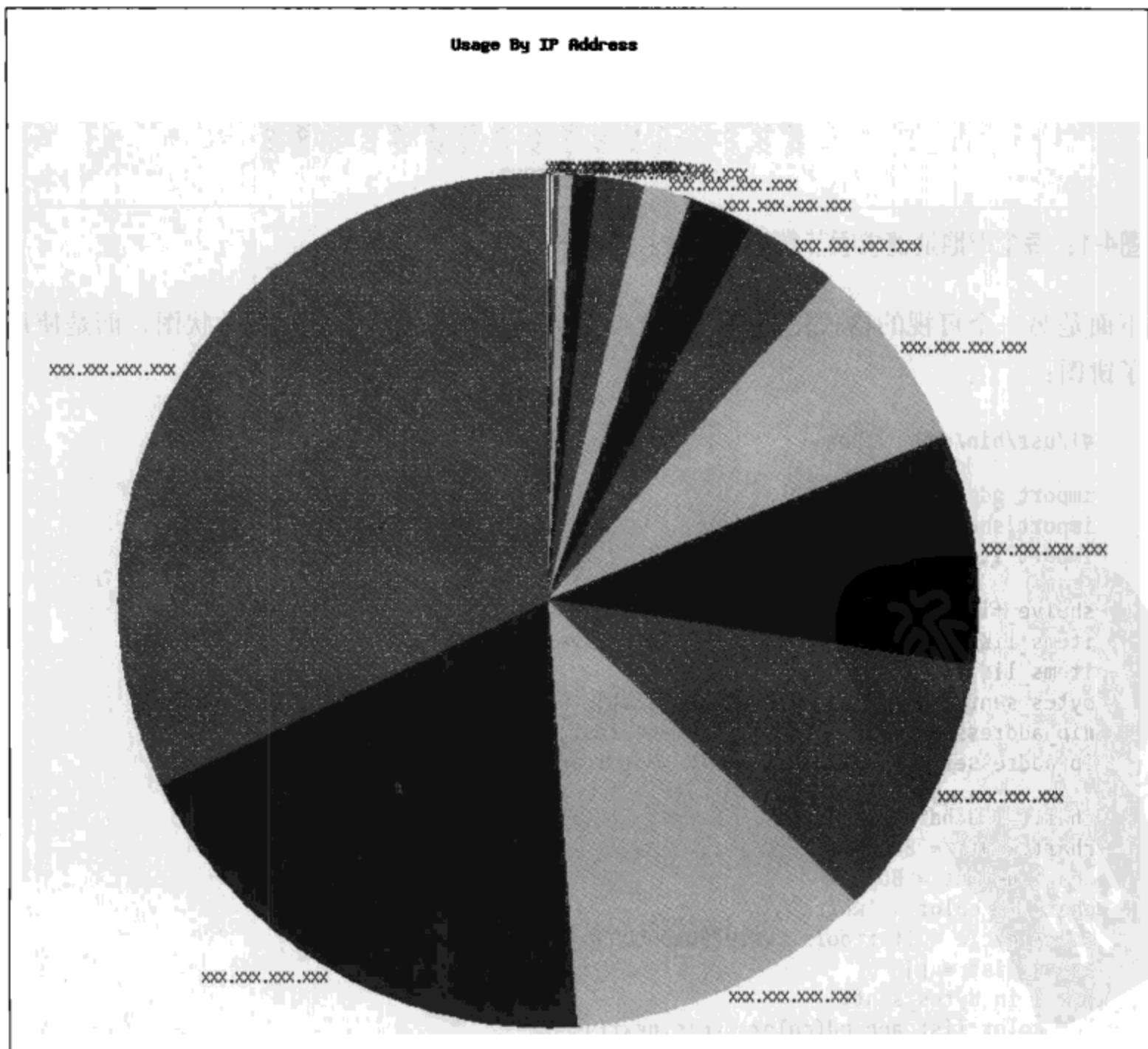


图4-2：每个IP地址请求字节数的饼图

在饼图中，唯一的问题是IP地址有可能会与具体的字节数值叠在一起显示。柱状图和饼图使得保存在shelve文件中的数据十分易于阅读，两种图也都很容易创建，同时，向其中添加信息也极其简单。

## PDF

另外一种对数据文件中的信息进行格式化的方法，是将信息保存在PDF文件中。现在PDF已经成为主流，我们总是希望所有的文档都可以转化为PDF格式。作为系统管理员，知道如何创建易读的PDF文档可以使自己的生活变得轻松。在阅读本节之后，你将能够创建各种PDF报告，例如网络应用情况、用户账号等。我们还将描述如何使用Python在多部件（multipart）MIME email中自动嵌入PDF。

在PDF库中，最重量级的库是ReportLab。ReportLab有自由版本和商业版本两种。你可以在ReportLab PDF库中查看一些示例，地址为<http://www.reportlab.com/docs/userguide.pdf>。除了阅读本章之外，我们强烈建议你阅读ReportLab的官方文档。为了在Ubuntu上安装ReportLab，可以简单地使用`apt-get install python-reportlab`命令。如果没有使用Ubuntu，可以搜索适合于你所使用操作系统的安装包。无论如何，总有一个源码发布版本可供使用。

例4-3演示了如何使用ReportLab创建一个“Hello World”PDF文件。

### 例4-3：“Hello World”PDF

```
#!/usr/bin/env python
from reportlab.pdfgen import canvas

def hello():
    c = canvas.Canvas("helloworld.pdf")
    c.drawString(100,100,"Hello World")
    c.showPage()
    c.save()
hello()
```

在创建“Hello World”PDF文件的过程中，有一些需要注意的地方。首先，我们创建了一个`canvas`对象。接下来使用`drawString()`方法，该方法等同于处理文本文件时使用的`file_obj.write()`方法。最后，由`showPage()`方法停止绘制，使用`save()`方法创建实际的PDF文件。如果运行这段代码，会得到一个空白的PDF文件，在文件底部有“Hello World”字样。

如果已经下载了ReportLab源码版本，就可以使用其中包含的测试作为示例驱动文档。也就是说，运行测试时，会生成一系列PDF文件，你可以将这些PDF与测试代码进行比较，来查看通过ReportLab库如何成功获得各种各样的可视化效果。

现在已经演示了如何使用ReportLab来创建PDF，接下来，让我们看看如何使用ReportLab来创建一个自定义磁盘使用情况报告。这样的报告是非常有用的。参见例4-4。

#### 例4-4：PDF磁盘报告

```
#!/usr/bin/env python
import subprocess
import datetime
from reportlab.pdfgen import canvas
from reportlab.lib.units import inch

def disk_report():
    p = subprocess.Popen("df -h", shell=True,
                        stdout=subprocess.PIPE)
    return p.stdout.readlines()

def create_pdf(input,output="disk_report.pdf"):
    now = datetime.datetime.today()
    date = now.strftime("%h %d %Y %H:%M:%S")
    c = canvas.Canvas(output)
    textobject = c.beginText()
    textobject.setTextOrigin(inch, 11*inch)
    textobject.textLines('''
Disk Capacity Report: %
''' % date)
    for line in input:
        textobject.textLine(line.strip())
    c.drawText(textobject)
    c.showPage()
    c.save()
report = disk_report()
create_pdf(report)
```

这段代码会产生一个报告来显示当前磁盘的使用情况，包括时间戳和“Disk Capacity Report”（磁盘容量报告）字样。完成这样一个功能仅用了如此少的几行代码，肯定会给你留下深刻印象。接下来看一下本示例中的一些亮点。首先，`disk_report()`函数只是简单地获得`df -h`命令的输出，并将其作为列表返回。接下来，在`create_pdf()`函数中创建了一个格式化的时间戳。而本示例中最重要的部分是`textobject`函数。

`textobject`函数用于创建一个放在PDF中的对象。我们先通过调用`beginText()`创建了一个`textobject`对象。然后，定义了数据打包的方法。这里使用的PDF大约为 $8.5 \times 11$ 英寸的文档，因此为了能从页面的最顶端进行打包，我们告诉`text`对象在11英寸处设置文本边界。之后，通过将字符串写入文本对象来创建标题，最后对`df`命令的执行结果的每一行进行迭代处理。值得注意的是这里使用`line.strip()`来删除换行字符。如果没有这样做，在换行符处会看到黑色方块。

通过添加颜色和图片，可以创建更为复杂的PDF文件。至于如何实现，可以通过阅读

ReportLab PDF库中非常不错的用户指南得到答案。通过这些示例可以看到其中最重要的部分是文本对象，文本对象可以收集和保留数据并最终输出数据。

## 信息发布

在获得并格式化数据之后，需要将其分发给对其感兴趣的人。这一节将主要讲述如何被文档通过email发送给其接收者。如果需要将文档发送到web服务器供用户查看，可以使用FTP。下一章将讨论Python标准FTP模块的使用。

### 发送email

处理email是系统管理的一个重要内容。不仅要管理email服务器，还需要通过email来产生警告和报警信息。Python标准库对于发送email提供了强有力的支持，只是我们之前提及的比较少。由于所有的系统管理员都对自动发送email感兴趣，本节将向你展示如何使用Python来完成各种各样的email任务。

#### 发送基本信息

在Python中有两个不同的包允许你发送电子邮件。一个是低级别的包smtplib，是针对RFC中关于SMTP协议的各种描述的接口。它可以发送email。另一个包是email，帮助解析和产生email。例4-5中使用smtplib建立一个包含邮件信息的字符串，然后使用email包将邮件发送至邮件服务器。

#### 例4-5：使用SMTP发送消息

```
#!/usr/bin/env python

import smtplib
mail_server = 'localhost'
mail_server_port = 25
from_addr = 'sender@example.com'
to_addr = 'receiver@example.com'

from_header = 'From: %s\r\n' % from_addr
to_header = 'To: %s\r\n\r\n' % to_addr
subject_header = 'Subject: nothing interesting'

body = 'This is a not-very-interesting email.'

email_message = '%s\n%s\n%s\n%s' % (from_header, to_header, subject_header, body)

s = smtplib.SMTP(mail_server, mail_server_port)
s.sendmail(from_addr, to_addr, email_message)
s.quit()
```

首先，我们定义了email服务器的主机和端口号，也定义了“to”和“from”地址。然后，通过连接邮件头与邮件体建立了email邮件。最后，连接到SMTP服务器，并从

`from_addr`发送到`to_addr`。应该注意到的是，为了与RFC规定相兼容，这里使用“\r\n”专门对From:和To:进行了格式化。

在第10章的“调度Python进程”部分有一个创建cron作业的代码示例，该示例使用Python实现了邮件的自动发送。现在，让我们从这一简单的基本示例出发，转到使用Python中的电子邮件可以完成的更有趣的一些事情上。

## 使用SMTP认证

上一个示例非常简单，对于使用Python发送email来说简单得有些微不足道。但不幸的是，有一些SMTP服务器会强迫你使用认证，因此上面的示例在许多情况下无法完成相应工作。例4-6是一个包含了SMTP认证的示例。

例4-6：SMTP认证

```
#!/usr/bin/env python
import smtplib
mail_server = 'smtp.example.com'
mail_server_port = 465

from_addr = 'foo@example.com'
to_addr = 'bar@exmaple.com'

from_header = 'From: %s\r\n' % from_addr
to_header = 'To: %s\r\n\r\n' % to_addr
subject_header = 'Subject: Testing SMTP Authentication'

body = 'This mail tests SMTP Authentication'

email_message = '%s\n%s\n%s\n\n%s' % (from_header, to_header, subject_header, body)

s = smtplib.SMTP(mail_server, mail_server_port)
s.set_debuglevel(1)
s.starttls()
s.login("fatalbert", "mysecretpassword")
s.sendmail(from_addr, to_addr, email_message)
s.quit()
```

其中，主要的差别在于定义了用户名和密码，启动了`debuglevel`，然后通过使用`starttls()`方法启动了SSL。使用认证时启动debugging是一个非常好的做法。如果我们查看一下失败的调试会话，会看到如下内容：

```
$ python2.5 mail.py
send: 'ehlo example.com\r\n'
reply: '250-example.com Hello example.com [127.0.0.1], pleased to meet you\r\n'
reply: '250-ENHANCEDSTATUSCODES\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-8BITMIME\r\n'
reply: '250-SIZE\r\n'
reply: '250-DSN\r\n'
reply: '250-ETRN\r\n'
```

```
reply: '250-DELIVERBY\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: example.com example.com [127.0.0.1], pleased to meet you
ENHANCEDSTATUSCODES
PIPELINING
8BITMIME
SIZE
DSN
ETRN
DELIVERBY
HELP
send: 'STARTTLS\r\n'
reply: '454 4.3.3 TLS not available after start\r\n'
reply: retcode (454); Msg: 4.3.3 TLS not available after start
```

在这个示例中，我们试图用来初始化SSL的服务器没有提供对SSL的支持，因此将我们直接拒绝。类似这种问题的处理方法十分简单，在试图使用级联服务器系统来发送电子邮件时，通过命令localhost attempt to send mail，许多其他潜在的问题通过编写包含错误处理代码的脚本可以简单地解决。

## 使用Python发送附件

仅发送包含文本的邮件已经跟不上潮流了。使用Python我们可以使用MIME标准来发送信息，它允许对邮件的附件进行编码。在本章前面部分，我们介绍了如何创建PDF报表。由于系统管理是非常需要耐心的，这里跳过了令人乏味的对MIME起源的介绍，直接来到发送带附件的邮件部分。参见例4-7。

### 例4-7：发送带PDF附件的email

```
import email
from email.MIMEText import MIMEText
from email.MIMEMultipart import MIMEMultipart
from email.MIMEBase import MIMEBase
from email import encoders
import smtplib
import mimetypes

from_addr = 'noah.gift@gmail.com'
to_addr = 'jjinux@gmail.com'
subject_header = 'Subject: Sending PDF Attachemt'
attachment = 'disk_usage.pdf'
body = '''
This message sends a PDF attachment created with Report
Lab.
'''

m = MIMEMultipart()
m["To"] = to_addr
m["From"] = from_addr
m["Subject"] = subject_header
```

```
ctype, encoding = mimetypes.guess_type(attachment)
print ctype, encoding
maintype, subtype = ctype.split('/', 1)
print maintype, subtype

m.attach(MIMEText(body))
fp = open(attachment, 'rb')
msg = MIMEBase(maintype, subtype)
msg.set_payload(fp.read())
fp.close()
encoders.encode_base64(msg)
msg.add_header("Content-Disposition", "attachment", filename=attachment)
m.attach(msg)

s = smtplib.SMTP("localhost")
s.set_debuglevel(1)
s.sendmail(from_addr, to_addr, m.as_string())
s.quit()
```

在这里，我们使用了一些小技巧，对之前创建的磁盘报告PDF文件进行编码，然后通过电子邮件发送出去。

## Trac

Trac是一个wiki和问题追踪系统，其典型应用是软件开发。但是只要你想使用wiki或是订票系统，就可以使用Trac。Trac在Python下编写。你可以在<http://trac.edgewall.org/>找到Trac的文档及安装包。这里不打算涉及Trac的更多细节，因为这超出了本书的范围。但是它对解决订票系统中的一般性故障，是非常好的工具。Trac的另一个非常有趣的地方是它可以通过插件来扩展。

之所以在此提到Trac，是因为它非常适合我们所讨论的所有三个类别：信息收集，格式化和发布。wiki允许用户通过浏览器创建web页面。他们添加的信息以HTML格式显示，这有利于其他用户通过浏览器查看。这是在这一章中多次讨论过的。

类似地，订票系统允许用户根据工作的需要提出请求，或是报告他们遇到的问题。你可以通过web界面报告输入的票数，甚至产生一个CSV报告。再说一次，Trac涵盖了这一章讨论的所有内容。

我们建议你对Trac是否可以满足自己的需要做进一步的尝试。或许你需要一些具有更多特征和性能的工具，或许你需要一些更简单的工具，不管哪种情况这确实是值得做更多尝试的事情。

## 本章小结

在本章中，我们介绍了自动和手工进行数据收集的方法。也介绍了如何将数据整合为各种不同的更适合发布的格式，如HTML、PDF和PNG。最后，我们查看了将信息传送到对其感兴趣的用户手中的方法。正如我们在本章开始时所说的，处理文档或许不是你所有工作中最吸引眼球的一项，甚至在记录文件时，你可能还没有意识到是在对文档进行处理。但是简单明了的文档是系统管理的一个重要内容。我们希望本章中的一些技巧可以使枯燥的文档处理工作变得更有生趣一些。



## 第5章

# 网络

说到网络，通常会涉及对多台计算机进行连接，保证它们之间可以相互通信。但是我们更感兴趣的不是计算机之间的相互通信，而是进程之间的相互通信。对于我们将要介绍的技术而言，进程是在同一台计算机上还是在不同的计算机上无关紧要。

本章集中介绍如何使用标准的socket库（也可能是建立在socket基础之上的其他库）编写Python程序，连接其他进程，并与其他进程进行交互。

## 网络客户端

在客户端连接到服务器之前，服务器一直处于等待状态，因此，连接是由客户端发起的。Python标准库中包括了许多已经实现的网络客户端代码。这一章，我们将讨论一些更为通用的、经常被使用的客户端。

### socket

socket模块为操作系统的socket连接提供了一个Python接口。这表示使用Python，可以完成任何使用socket或对socket进行处理的操作。如果之前没有做过任何网络编程工作，本章将会给出一个网络编程的简要说明。这会给你一个大体印象，即使用Python的networking库都可以完成哪些操作。

socket模块提供了工厂函数socket()。socket()函数会返回一个socket对象。为了定义socket的类型，需要传递给socket()一些参数。如果不带参数调用socket()工厂函数，其返回的socket对象默认使用TCP/IP协议：

```
In [1]: import socket  
In [2]: s = socket.socket()  
In [3]: s.connect(('192.168.1.15', 80))
```

```
In [4]: s.send("GET / HTTP/1.0\n\n")
Out[4]: 16

In [5]: s.recv(200)
Out[5]: 'HTTP/1.1 200 OK\r\n\
Date: Mon, 03 Sep 2007 18:25:45 GMT\r\n\
Server: Apache/2.0.55 (Ubuntu) DAV/2 PHP/5.1.6\r\n\
Content-Length: 691\r\n\
Connection: close\r\n\
Content-Type: text/html; charset=UTF-8\r\n\
\r\n\
<!DOCTYPE HTML P'
In [6]: s.close()
```

该示例通过`socket()`工厂函数创建一个名为`s`的`socket`对象。它连接到本地默认的web服务器，指定端口号为HTTP默认的端口号80。接下来，它将文本字符串"GET / HTTP/1.0\n\n"（一个简单的HTTP请求）发送到服务器。发送完毕之后，它会收到服务器响应的前200个字节，这200字节包含状态信息和HTTP头信息。最后，关闭连接。

本例中演示的`socket`方法可能是你最常使用的方法。`connect()`用于在`socket`对象与远程对象（即“不是当前的`socket`对象”）之间建立通信连接。`send()`用于从`socket`对象向远端发送数据。`recv()`用于接收远端发送的任何数据。`close()`则用于关闭两个`socket`之间的连接。这一简单的示例表明了创建`socket`对象，并通过该对象发送和接收数据是非常容易的。

现在，让我们看一个更有用的示例。假定有一个运行某些网络应用程序的服务器，例如一个web服务器。现在希望查看该服务器，以确认在经历了一整天运行之后，仍可以创建一个连接到该web服务器上的`socket`连接。这几乎是一个最小规模的监测程序，但是却可以证明这个web服务器仍在工作，并且它仍旧在侦听某些端口。参见例5-1。

### 例5-1：TCP 端口检查

```
#!/usr/bin/env python

import socket
import re
import sys

def check_server(address, port):
    #create a TCP socket
    s = socket.socket()
    print "Attempting to connect to %s on port %s" % (address, port)
    try:
        s.connect((address, port))
        print "Connected to %s on port %s" % (address, port)
        return True
    except socket.error, e:
        print "Connection to %s on port %s failed: %s" % (address, port, e)
        return False
```

```

if __name__ == '__main__':
    from optparse import OptionParser
    parser = OptionParser()

    parser.add_option("-a", "--address", dest="address", default='localhost',
                      help="ADDRESS for server", metavar="ADDRESS")

    parser.add_option("-p", "--port", dest="port", type="int", default=80,
                      help="PORT for server", metavar="PORT")

(options, args) = parser.parse_args()
print 'options: %s, args: %s' % (options, args)
check = check_server(options.address, options.port)
print 'check_server returned %s' % check
sys.exit(not check)

```

所有的工作都由`check_server()`函数完成的。`check_server()`先创建了一个`socket`对象，之后试图连接到指定的地址和端口号。如果成功，则返回值为真。如果失败，`socket.connect()`调用会抛出一个可操作的异常，函数返回值为假。代码中的`main`部分调用`check_server()`。`main`部分解析用户的参数，并将用户请求的参数转化为适当的格式传递给`check_server()`。整个脚本代码在执行过程中都能够输出状态信息。输出的最后一项数据是`check_server()`的返回值。最后，脚本将与`check_server()`返回值相反的值返回给`shell`。返回与`check_server()`返回值相反的值使得该脚本成为一个极为有用的工具。成功时返回0值到`shell`，失败时返回非0值（正数）到`shell`，这是一种非常典型的用法。以下是一个成功连接到web服务器的代码示例：

→ jmjones@dinkgutsy:code\$ python port\_checker\_tcp.py -a 192.168.1.15 -p 80
options: {'port': 80, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
check\_server returned True

最后一行输出`check_server returned True`，表示成功建立连接。

下面是一个失败的连接示例：

→ jmjones@dinkgutsy:code\$ python port\_checker\_tcp.py -a 192.168.1.15 -p 81
options: {'port': 81, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 81
Connection to 192.168.1.15 on port 81 failed: (111, 'Connection refused')
check\_server returned False

日志的最后一行是`check_server returned False`，这表明连接失败。倒数第二行的输出为`Connection to 192.168.1.15 on port 81 failed`，是因为“`Connection refused`”（连接被拒绝）。大胆地猜测一下，可能与这台服务器中没有在81端口守护的进程有关。

现在已经创建了三个示例来展示如何在`shell`脚本中使用`socket`工具。首先，给出运行

脚本的shell命令，如果脚本成功执行，则输出SUCCESS。这里用&&操作符来代替if-then语句：

```
→ $ python port_checker_tcp.py -a 192.168.1.15 -p 80 && echo "SUCCESS"
options: {'port': 80, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
check_server returned True
SUCCESS
```

可以看到，该脚本成功执行，因此，在执行并输出状态结果后，shell输出SUCCESS。

```
→ $ python port_checker_tcp.py -a 192.168.1.15 -p 81 && echo "FAILURE"
options: {'port': 81, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 81
Connection to 192.168.1.15 on port 81 failed: (111, 'Connection refused')
check_server returned False
```

该脚本执行失败，但不会输出FAILURE：

```
→ $ python port_checker_tcp.py -a 192.168.1.15 -p 81 && echo "FAILURE"
options: {'port': 81, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 81
Connection to 192.168.1.15 on port 81 failed: (111, 'Connection refused')
check_server returned False
FAILURE
```

这个脚本执行失败，但是我们将&&变为||，这意味着如果脚本返回结果为假，将打印输出FAILURE。可以看到，它确实是这样执行的。

事实上，web服务器允许连接80端口，并不意味着存在可供连接使用的HTTP服务器。可以通过一个测试帮助我们准确地判断web服务器的状态，该测试可以检测是否产生HTTP头以及一些特定的URL状态代码。例5-2就实现了这样一个测试。代码如下所示：

#### 例5-2：基于Socket的web服务器检测

```
→ #!/usr/bin/env python

import socket
import re
import sys

def check_webserver(address, port, resource):
    #build up HTTP request string
    if not resource.startswith('/'):
        resource = '/' + resource
    request_string = "GET %s HTTP/1.1\r\nHost: %s\r\n\r\n" % (resource, address)
    print 'HTTP request:'
    print '%%%s%%%s' % request_string
```

```

#create a TCP socket
s = socket.socket()
print "Attempting to connect to %s on port %s" % (address, port)
try:
    s.connect((address, port))
    print "Connected to %s on port %s" % (address, port)
    s.send(request_string)
    #we should only need the first 100 bytes or so
    rsp = s.recv(100)
    print 'Received 100 bytes of HTTP response'
    print '|||%s|||' % rsp
except socket.error, e:
    print "Connection to %s on port %s failed: %s" % (address, port, e)
    return False
finally:
    #be a good citizen and close your connection
    print "Closing the connection"
    s.close()
lines = rsp.splitlines()
print 'First line of HTTP response: %s' % lines[0]
try:
    version, status, message = re.split(r'\s+', lines[0], 2)
    print 'Version: %s, Status: %s, Message: %s' % (version, status, message)
except ValueError:
    print 'Failed to split status line'
    return False
if status in ['200', '301']:
    print 'Success - status was %s' % status
    return True
else:
    print 'Status was %s' % status
    return False

if __name__ == '__main__':
    from optparse import OptionParser
    parser = OptionParser()
    parser.add_option("-a", "--address", dest="address", default='localhost',
                      help="ADDRESS for webserver", metavar="ADDRESS")

    parser.add_option("-p", "--port", dest="port", type="int", default=80,
                      help="PORT for webserver", metavar="PORT")

    parser.add_option("-r", "--resource", dest="resource", default='index.html',
                      help="RESOURCE to check", metavar="RESOURCE")

(options, args) = parser.parse_args()
print 'options: %s, args: %s' % (options, args)
check = check_webserver(options.address, options.port, options.resource)
print 'check_webserver returned %s' % check
sys.exit(not check)

```

与之前使用`check_server()`完成所有工作的示例类似，本示例中使用`check_webserver()`来完成所有工作。首先，`check_webserver()`建立HTTP请求字符串。如果不了解HTTP，可以把HTTP协议理解为一种已定义的HTTP客户端与服务器进行通信的方法。`check_`

`webserver()`建立的HTTP请求几乎是最简单的HTTP请求。接下来，`check_webserver()`创建一个socket对象，连接到服务器，并向服务器发送HTTP请求。之后，它读取从服务器返回的响应并关闭连接。出现socket错误时，`check_webserver()`返回False，表示检测失败。它会取出从服务器读取的信息，并从中提出状态代码。如果状态码是表示OK的200，或是表示永久移动的301，`check_webserver()`都会返回True，否则，返回False。脚本中的main部分解析用户输入，并调用`check_webserver()`。在从`check_webserver()`取得结果后，它向shell返回与`check_webserver()`返回值相反的值，这个与之前使用普通socket的检测代码相似。我们希望能够从shell脚本调用该方法，并且查看是否成功。下面是执行代码的示例：

```
$ python web_server_checker_tcp.py -a 192.168.1.15 -p 80 -r apache2-default
options: {'resource': 'apache2-default', 'port': 80, 'address': '192.168.1.15'}, args: []
HTTP request:
|||GET /apache2-default HTTP/1.1
Host: 192.168.1.15

||
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
Received 100 bytes of HTTP response
|||HTTP/1.1 301 Moved Permanently
Date: Wed, 16 Apr 2008 23:31:24 GMT
Server: Apache/2.0.55 (Ubuntu) ||
Closing the connection
First line of HTTP response: HTTP/1.1 301 Moved Permanently
Version: HTTP/1.1, Status: 301, Message: Moved Permanently
Success - status was 301
check_webserver returned True
```

最后四行输出表示在这个Web服务器上，HTTP/apache2-default的默认状态码为301，这说明运行是成功的。

下面是另一次运行。这一次，我们特意制定一个并不存在的资源，以查看HTTP调用失败时的显示结果：

```
$ python web_server_checker_tcp.py -a 192.168.1.15 -p 80 -r foo
options: {'resource': 'foo', 'port': 80, 'address': '192.168.1.15'}, args: []
HTTP request:
|||GET /foo HTTP/1.1
Host: 192.168.1.15

||
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
Received 100 bytes of HTTP response
|||HTTP/1.1 404 Not Found
Date: Wed, 16 Apr 2008 23:58:55 GMT
Server: Apache/2.0.55 (Ubuntu) DAV/2 PH|||
```

```
Closing the connection
First line of HTTP response: HTTP/1.1 404 Not Found
Version: HTTP/1.1, Status: 404, Message: Not Found
Status was 404
check_webserver returned False
```

之前示例代码中，最后四行显示代码被成功执行，而这个示例代码的最后四行却表明它没有成功执行。由于在web服务器上不存在/`foo`，检测程序返回`False`。

本节主要介绍如何构建底层网络服务器连接，并实现基本的检测功能。通过一系列示例，展示了客户端与服务器进行通讯时有可能出现的不同场景。如果你有机会编写网络组件，应该使用高于`socket`模块的其他库。实际上，在真正编写网络组件的时候，并不需要在类似`socket`这样的底层库上花费太多时间。

## httpplib

先前的示例演示了如何直接使用`socket`模块创建一个HTTP请求。接下来的示例将演示如何使用`httpplib`模块。我们首先考虑的是，什么时候应该使用`httpplib`模块而不是`socket`模块呢？或者说，什么时候应该使用更高层的库而不是较低层的库呢？真正的经验是根据具体情况而定。有时候，我们需要使用较低层的库。比如当我们需要完成一些在可用的库中找不到的任务时，或是需要对库中的任务进行细粒度的控制时，或是需要有更出色的性能时。但是在本示例中，我们没有任何理由不使用`httpplib`这样的高级库而使用`socket`这样的低级库。例5-3实现了与之前示例相同的功能，只是这里使用`httpplib`模块完成。

### 例5-3：基于`httpplib`的web服务器检测

```
#!/usr/bin/env python

import httpplib
import sys

def check_webserver(address, port, resource):
    #create connection
    if not resource.startswith('/'):
        resource = '/' + resource
    try:
        conn = httpplib.HTTPConnection(address, port)
        print 'HTTP connection created successfully'
        #make request
        req = conn.request('GET', resource)
        print 'request for %s successful' % resource
        #get response
        response = conn.getresponse()
        print 'response status: %s' % response.status
    except Exception, e:
        print 'An error occurred: %s' % e
```

```

except sock.error, e:
    print 'HTTP connection failed: %s' % e
    return False
finally:
    conn.close()
    print 'HTTP connection closed successfully'
if response.status in [200, 301]:
    return True
else:
    return False

if __name__ == '__main__':
    from optparse import OptionParser
    parser = OptionParser()
    parser.add_option("-a", "--address", dest="address", default='localhost',
                      help="ADDRESS for webserver", metavar="ADDRESS")
    parser.add_option("-p", "--port", dest="port", type="int", default=80,
                      help="PORT for webserver", metavar="PORT")
    parser.add_option("-r", "--resource", dest="resource", default='index.html',
                      help="RESOURCE to check", metavar="RESOURCE")
    (options, args) = parser.parse_args()
    print 'options: %s, args: %s' % (options, args)
    check = check_webserver(options.address, options.port, options.resource)
    print 'check_webserver returned %s' % check
    sys.exit(not check)

```

本示例与socket示例非常相似。两者最大的差异是你没必要手动创建HTTP请求，也不必手动解析HTTP响应。`httplib`连接对象具有`request()`方法，该方法能够建立和发送HTTP请求。`connection`对象也有一个`getresponse()`方法，该方法可以创建一个响应对象。可以通过引用响应对象的状态（status）属性访问HTTP的状态。虽然这并没有少写代码，但无须我们手动创建、发送和接收HTTP请求和响应，减少了不必要的麻烦。而且，这段代码看起来也让人觉得更整洁一些。

下面的示例中，我们使用了与之前示例成功执行时所使用的相同的命令行参数在web服务器上寻找/，并且找到了：

```

$ python web_server_checker_httplib.py -a 192.168.1.15 -r /
options: {'resource': '/', 'port': 80, 'address': '192.168.1.15'}, args: []
HTTP connection created successfully
request for / successful
response status: 200
HTTP connection closed successfully
check_webserver returned True

```

下面的示例中，我们使用了与之前示例执行失败时所使用的相同的命令行参数寻找/foo目录，但是没有找到。

```

$ python web_server_checker_httplib.py -a 192.168.1.15 -r /foo
options: {'resource': '/foo', 'port': 80, 'address': '192.168.1.15'}, args: []
HTTP connection created successfully

```

```
request for /foo successful
response status: 404
HTTP connection closed successfully
check_webserver returned False
```

正如之前所说，如果有机会使用高级库，一定要使用高级库。使用`httpplib`而不是单独使用`socket`模块会使代码更简洁、清晰。而代码越简洁，其中的bug就越少。

## ftplib

除了`socket`和`httpplib`模块之外，Python标准库还包含了一个名为`ftplib`的FTP客户端模块。`ftplib`是一个全功能（full-featured）的FTP客户端库，它可让你以编程方式执行任何通常会使用FTP客户端应用程序来执行的任务。例如，可以登录FTP服务器，列出指定目录中的文件，下载文件、上传文件、更改目录、退出，所有这一切都可以通过Python脚本来完成。你甚至可以使用许多在Python中可用的GUI框架，建立属于自己的GUI FTP应用程序。

在这里，我们没有对该库进行全面的介绍，仅展示了例5-4，然后对该示例进行解析。

### 例5-4：使用`ftplib`实现FTP URL retriever

```
#!/usr/bin/env python

from ftplib import FTP
import ftplib
import sys
from optparse import OptionParser

parser = OptionParser()

parser.add_option("-a", "--remote_host_address", dest="remote_host_address",
    help="REMOTE FTP HOST.",
    metavar="REMOTE FTP HOST")

parser.add_option("-r", "--remote_file", dest="remote_file",
    help="REMOTE FILE NAME to download.",
    metavar="REMOTE FILE NAME")

parser.add_option("-l", "--local_file", dest="local_file",
    help="LOCAL FILE NAME to save remote file to", metavar="LOCAL FILE NAME")

parser.add_option("-u", "--username", dest="username",
    help="USERNAME for ftp server", metavar="USERNAME")

parser.add_option("-p", "--password", dest="password",
    help="PASSWORD for ftp server", metavar="PASSWORD")

(options, args) = parser.parse_args()

if not (options.remote_file and
        options.local_file and
        options.remote_host_address):
```

```
parser.error('REMOTE HOST, LOCAL FILE NAME, ' \
    'and REMOTE FILE NAME are mandatory')

if options.username and not options.password:
    parser.error('PASSWORD is mandatory if USERNAME is present')

ftp = FTP(options.remote_host_address)
if options.username:
    try:
        ftp.login(options.username, options.password)
    except ftplib.error_perm, e:
        print "Login failed: %s" % e
        sys.exit(1)
else:
    try:
        ftp.login()
    except ftplib.error_perm, e:
        print "Anonymous login failed: %s" % e
        sys.exit(1)
try:
    local_file = open(options.local_file, 'wb')
    ftp.retrbinary('RETR %s' % options.remote_file, local_file.write)
finally:
    local_file.close()
    ftp.close()
```

代码的起始部分（跳过所有的命令行解析）创建了一个FTP对象，该对象通过将FTP服务器的地址传递给FTP构造器（constructor）来实现。另一种可选择的方法是，创建FTP对象，但不向构造器传递参数，创建之后再调用connect()方法，而connect()方法需要指定FTP服务器地址。之后，登录到FTP服务器。如果提供了用户名和密码，则以该用户名和密码登录。如果没有提供，则使用匿名登录。接下来，创建了一个文件对象保存FTP服务器上的文件数据。之后，调用FTP对象的retrbinary()方法。retrbinary()方法，正如其名称所表达的含义，表示从FTP服务器上获得一个二进制文件。该方法需要两个参数：FTP的retrieve命令和一个回调（callback）函数。你或许注意到，这里的回调函数是在前一步骤中创建的文件对象的write方法。值得注意的是，在本示例中没有调用write()方法。我们将write方法传入retrbinary()方法，这样retrbinary()就可以调用write()。retrbinary()会连同传递给它的从FTP服务器上获得的数据块调用传递给它的任何回调函数。回调函数可以对数据进行任何处理。这里，该回调函数仅对它从FTP服务器收到的字节数进行记录。传递一个file对象的write方法能够将脚本从FTP服务器上获得的文件内容写入到file对象中。最后，关闭文件对象和FTP连接。这一处理过程存在着一些疏漏：我们在FTP服务器上获取文件的代码中建立了一个try块，在关闭本地文件和FTP连接的代码中使用了finally块。这样，如果有错误发生，就会在脚本结束之前试图清除文件。附录提供了一个对回调函数的简明介绍，仅供参考。

## urllib

urllib位于标准库模块的更高层。看到urllib时，我们很容易会想到HTTP库，而忘记了FTP资源也是可以通过URL来识别的。因此，或许你从没有想过使用urllib来获取FTP资源，但它的确具备这一功能。例5-5与之前的ftplib示例实现的功能相同，只是这里使用了urllib。

### 例5-5：使用urllib实现FTP URL retriever

```
▶▶▶ #!/usr/bin/env python
"""
url retriever

Usage:

url_retrieve_urllib.py URL FILENAME

URL:
If the URL is an FTP URL the format should be:
ftp://[username[:password]@]hostname/filename
If you want to use absolute paths to the file to download,
you should make the URL look something like this:
ftp://user:password@host/%2Fpath/to/myfile.txt
Notice the '%2F' at the beginning of the path to the file.

FILENAME:
absolute or relative path to the filename to save downloaded file as
"""

import urllib
import sys

if '-h' in sys.argv or '--help' in sys.argv:
    print __doc__
    sys.exit(1)

if not len(sys.argv) == 3:
    print 'URL and FILENAME are mandatory'
    print __doc__
    sys.exit(1)
url = sys.argv[1]
filename = sys.argv[2]
urllib.urlretrieve(url, filename)
```

这一段脚本简短而且亲切，它展示了urllib的强大之处。事实上，其间有很多是使用文档而并非代码，甚至注释都比代码多。我们使用该脚本完成了一个非常简单的参数解析过程。由于两个选项是必须的，因此我们需要指定具体的参数。在该示例中，仅有的有效代码行如下所示：

```
▶▶▶ urllib.urlretrieve(url, filename)
```

在获得`sys.argv`选项之后，代码下载指定的URL，并保存为指定的本地文件名。使用HTTP URL和FTP URL都可以，甚至在URL中包含用户名和密码也可以。

需要强调的一点是，你有可能认为应该有比使用其他语言完成这些功能更容易一些的方法，的确如此。在Python标准库中应该会有一些更高级的库，可以完成经常要做的工作。而在本示例中，`urllib`已经能够准确地执行我们想做的事，因此，无须使用更多的库。有时候，Python标准库可能不能满足需要，但可以找到其他的Python资源，如`http://pypi.python.org/pypi`提供的Python包索引（PyPI）。

## urllib2

另外一个高级库是`urllib2`。`urllib2`包含了很多与`urllib`相似的功能，是对`urllib`的扩展。例如，`urllib2`能够更好地支持认证和cookie。因此，当你发现`urllib`无法完成一些任务时，应该查看一下`urllib2`，看看它是否能够满足需要。

# 远程过程调用

网络编程的典型目的就是实现进程间的通信（IPC）。通常情况下，简单的IPC使用HTTP或是socket就足够用了。但是，有些时候，需要在不同进程或者是不同计算机之间执行代码，这时，IPC能够让你有在同一进程中执行代码的感觉。事实上，如果可以在Python程序中远程执行一些代码，你或许会希望远程调用可以返回Python对象，这样你就可以更容易地进行处理，而不是返回一大堆文本后再进行手工解析。值得高兴的是，有许多能够执行RPC（远程过程调用）功能的工具可供使用。

## XML-RPC

XML-RPC在两个进程之间交换指定格式的XML文档，以实现远程过程调用。但是这里不需要考虑XML的问题，很可能你根本就不需要知道两个进程间所交换的文档的具体格式。要使用XML-RPC，只需要知道在Python标准库中已经有了客户端和服务器端的实现。此外，XML-RPC在大多数编程语言中都可以使用，而且使用起来非常简单。例5-6是一个简单的XML-RPC服务器。

### 例5-6：简单 XML-RPC 服务器

```
#!/usr/bin/env python

import SimpleXMLRPCServer
import os

def ls(directory):
    try:
        return os.listdir(directory)
```

```

except OSError:
    return []

def ls_boom(directory):
    return os.listdir(directory)

def cb(obj):
    print "OBJECT:::", obj
    print "OBJECT.__class__:::", obj.__class__
    return obj.cb()

if __name__ == '__main__':
    s = SimpleXMLRPCServer.SimpleXMLRPCServer(('127.0.0.1', 8765))
    s.register_function(ls)
    s.register_function(ls_boom)
    s.register_function(cb)
    s.serve_forever()

```

该代码创建了一个新的SimpleXMLRPCServer对象，并将其绑定到回环地址127.0.0.1的端口8765上，这使得该对象只能访问指定机器上的进程。然后对已定义的函数ls()、ls\_boom()和cb()进行注册。稍后我们会对cb()函数进行介绍。ls()函数会列出使用os.listdir()传递给它的目录中的所有内容，并且以列表的方式返回结果。ls()会屏蔽任何OSError异常。ls\_boom()可以将异常返回到XML-RPC客户端。接下来，程序进入serve\_forever()循环，该循环等待可以处理的连接。下面是上述代码在IPython shell中使用的示例。



```

In [1]: import xmlrpclib
In [2]: x = xmlrpclib.ServerProxy('http://localhost:8765')
In [3]: x.ls('.')
Out[3]:
['.svn',
 'web_server_checker_httplib.py',
 ...
 'subprocess_arp.py',
 'web_server_checker_tcp.py']

In [4]: x.ls_boom('.')
Out[4]:
['.svn',
 'web_server_checker_httplib.py',
 ...
 'subprocess_arp.py',
 'web_server_checker_tcp.py']

In [5]: x.ls('/foo')
Out[5]: []
In [6]: x.ls_boom('/foo')
-----
<class 'xmlrpclib.Fault'>                                Traceback (most recent call last)
...

```

```
<<big nasty traceback>>
.
.
.
...
786     if self._type == "fault":
--> 787         raise Fault(**self._stack[0])
788     return tuple(self._stack)
789

<class 'xmlrpclib.Fault': <Fault 1: '<type 'exceptions.OSError'>
:[Errno 2] No such file or directory: '/foo'">
```

首先，通过传递XML-RPC服务器地址来创建一个`ServerProxy()`对象。然后，调用`.ls('..')`来查看服务器当前工作目录中有哪些文件。服务器在保存有示例代码的目录下运行，因此，你可以从该目录列表中看到这些文件。真正有趣的事是在客户端这一边，`x.ls('..')`返回一个Python列表。如果服务器是由Java、Perl、Ruby或是C#来实现的，也会有相同情况。实现服务器的语言可以执行列目录，创建列表，数组或是文件名集合；XML-RPC服务器代码可以用XML格式来表示创建的列表或数组，并通过连接客户端的线程将数据返回。这里也对`ls_boom()`进行了测试。由于`ls_boom()`缺少`ls()`的异常处理，可以看到异常从服务器传回了客户端。我们甚至在客户端也可以进行追踪。

XML-RPC带来了交互操作的可能性，非常有意义。但更有意义的是，可以写一个能够在多台机器上运行的代码，并根据需要远程执行。

但是，XML-RPC也不是完全没限制的，关键在于你是否把这些限制看作麻烦。例如，如果传入了一个自定义的Python对象，XML-RPC库会将该对象转换为Python字典，再将其串行化为XML，然后进行传递。这样，你需要写一些从字典的XML版本中提取数据的代码。实际上也可以直接使用RPC服务器上的对象。因此，是否使用XML-RPC，同样需要根据需要进行选择。

## Pyro

Pyro是能够避免XML-RPC缺点的框架。Pyro代表Python Remote Objects（Python远程对象，首字母缩写）。Pyro能完成任何XML-RPC能够实现的功能，而不需将对象字典化。在传递数据时，它能够保持其原有类型。如果确定要使用Pyro，必需独立安装它。Python中不包含Pyro。值得注意的是，Pyro仅能够与Python一起使用，而XML-RPC既可以与Python一起使用，也可以与其他语言一起使用。例5-7实现了与XML-RPC示例相同的`ls()`功能。

### 例5-7：简单的Pyro服务器

```
#!/usr/bin/env python
```

```

import Pyro.core
import os
from xmlrpclib import PSACB

class PSAExample(Pyro.core.ObjBase):

    def ls(self, directory):
        try:
            return os.listdir(directory)
        except OSError:
            return []

    def ls_boom(self, directory):
        return os.listdir(directory)

    def cb(self, obj):
        print "OBJECT:", obj
        print "OBJECT.__class__:", obj.__class__
        return obj.cb()

if __name__ == '__main__':
    Pyro.core.initServer()
    daemon=Pyro.core.Daemon()
    uri=daemon.connect(PSAExample(),"psaexample")

    print "The daemon runs on port:",daemon.port
    print "The object's uri is:",uri
    daemon.requestLoop()

```

Pyro示例与XML-RPC示例相似。首先，创建了PSAExample类，该类具有ls()、ls\_boom()和cb()方法。然后通过Pyro的内部管道（internal plumbing）创建了daemon。接下来，将PSAExample与daemon关联。最后，通知daemon开始服务请求。

下面的示例演示如何在IPython提示符下访问Pyro服务器：

```

In [1]: import Pyro.core
/usr/lib/python2.5/site-packages/Pyro/core.py:11: DeprecationWarning:
The sre module is deprecated, please import re.
import sys, time, sre, os, weakref

In [2]: psa = Pyro.core.getProxyForURI("PYROLOC://localhost:7766/psaexample")
Pyro Client Initialized. Using Pyro V3.5

In [3]: psa.ls(".")
Out[3]:
['pyro_server.py',
...
['subprocess_arp.py',
 'web_server_checker_tcp.py']

In [4]: psa.ls_boom('.')
Out[4]:
['pyro_server.py',
...
['subprocess_arp.py',
 'web_server_checker_tcp.py']

```

```

In [5]: psa.ls("/foo")
Out[5]: []

In [6]: psa.ls_boom("/foo")
-----
<type 'exceptions.OSError'>                                Traceback (most recent call last)
/home/jmjones/local/Projects/psabook/oreilly/<ipython console> in <module>()
.
.
...
<<big nasty traceback>>
...
.
.
--> 115             raise self.excObj
116         def __str__(self):
117             s=self.excObj.__class__.__name__

<type 'exceptions.OSError'>: [Errno 2] No such file or directory: '/foo'

```

非常不错！该示例返回了与XML-RPC示例相同的输出结果。这正是我们所期望的结果。但是，当传递一个自定义对象时，会是什么情况呢？下面，我们将要定义一个新的类，创建一个属于该类的对象，然后传递给XML-RPC的cb()函数以及Pyro示例中的cb()方法。例5-8显示了我们将要执行的代码段。

#### 例5-8：XML-RPC 与Pyro的区别

```

>>> import Pyro.core
>>> import xmlrpclib

class PSACB:
    def __init__(self):
        self.some_attribute = 1

    def cb(self):
        return "PSA callback"

if __name__ == '__main__':
    cb = PSACB()

    print "PYRO SECTION"
    print "*" * 20
    psapyro = Pyro.core.getProxyForURI("PYROLOC://localhost:7766/psaexample")
    print "-->", psapyro.cb(cb)
    print "*" * 20

    print "XML-RPC SECTION"
    print "*" * 20
    psaxmlrpc = xmlrpclib.ServerProxy('http://localhost:8765')
    print "-->", psaxmlrpc.cb(cb)
    print "*" * 20

```

调用Pyro和XML-RPC的cb()函数都需要在传递给它们的对象上调用cb()。在这两个示例中，都返回PSA回调字符串。下面是运行该代码时返回的内容：

```
jmjones@dinkgutsy:code$ python xmlrpc_pyro_diff.py
/usr/lib/python2.5/site-packages/Pyro/core.py:11: DeprecationWarning:
The sre module is deprecated, please import re.
    import sys, time, sre, os, weakref
PYRO SECTION
*****
Pyro Client Initialized. Using Pyro V3.5
-->> PSA callback
*****
XML-RPC SECTION
*****
-->>
Traceback (most recent call last):
  File "xmlrpc_pyro_diff.py", line 23, in <module>
    print "-->>", psaxmlrpclib.cb(cb)
  File "/usr/lib/python2.5/xmlrpclib.py", line 1147, in __call__
    return self.__send(self.__name, args)
  File "/usr/lib/python2.5/xmlrpclib.py", line 1437, in __request
    verbose=self.__verbose
  File "/usr/lib/python2.5/xmlrpclib.py", line 1201, in request
    return self.__parse_response(h.getfile(), sock)
  File "/usr/lib/python2.5/xmlrpclib.py", line 1340, in __parse_response
    return u.close()
  File "/usr/lib/python2.5/xmlrpclib.py", line 787, in close
    raise Fault(**self.__stack[0])
xmlrpclib.Fault: <Fault 1: '<type 'exceptions.AttributeError'>:'dict' object
has no attribute 'cb'">
```

Pyro的实现方法能够成功执行，但是XML-RPC的实现方法执行失败并返回一个追踪（traceback）。追踪返回的最后一行表明字典对象没有cb属性。当我们展示XML-RPC服务器的输出结果时，这会更容易理解。记住，cb()函数包含一些print语句，能够输出运行情况的信息。下面是XML-RPC服务器的输出结果：

```
OBJECT:: {'some_attribute': 1}
OBJECT.__class__:: <type 'dict'>
localhost - - [17/Apr/2008 16:39:02] "POST /RPC2 HTTP/1.0" 200 -
```

当我们字典化XML-RPC客户端创建的对象时，some\_attribute被转换为一个字典关键字。当这个属性被保留时，cb()方法不保留。

下面是Pyro服务器的输出结果：

```
OBJECT: <xmlrpc_pyro_diff.PSACB instance at 0x9595a8>
OBJECT.__class__: xmlrpc_pyro_diff.PSACB
```

值得注意的是，对象所属类为PSACB。PSACB定义了对象如何被创建。在Pyro服务器端，

我们不得不包括客户端使用的相同的代码。Pyro服务器需要导入客户端代码，这一点非常重要。Pyro使用Python标准的pickle来序列化对象，因此，它与Pyro非常相似。

总之，如果想要一个简单的RPC解决方案，而不希望有外部依赖，并且能够容忍XML-RPC的一些限制，就可以选择XML-RPC。如果再考虑到它能够与其他语言方便地实现互操作，那么或许就会认为XML-RPC是一个不错的选择了。另一方面，如果嫌XML-RPC限制太多，也不介意安装外部库，而且不介意仅使用Python一种语言，那么对你来说Pyro或许才是更好的选择。

## SSH

SSH是一个极其强大并被广泛使用的协议。由于大多数协议的实现都会与协议有相同的名字，因此也可以将SSH视为一个工具。SSH允许你安全地连接到远程服务器，执行shell命令，传输文件，并在连接双方进行端口转发。

如果有一个命令行的SSH工具，为什么还要通过编写脚本来使用SSH协议呢？主要原因是这样做除了能够使用SSH协议的全部功能外，还能够使用Python的全部功能。

SSH2协议就是通过名为paramiko的Python库实现的。通过Python代码，可以连接到SSH服务器，并完成一些SSH任务。例5-9是一个连接到SSH服务器并执行简单命令的示例。

### 例5-9：连接到SSH服务器并远程执行命令

```
#!/usr/bin/env python

import paramiko

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
password = 'xxxYYYYxxx'

if __name__ == "__main__":
    paramiko.util.log_to_file('paramiko.log')
    s = paramiko.SSHClient()
    s.load_system_host_keys()
    s.connect(hostname, port, username, password)
    stdin, stdout, stderr = s.exec_command('ifconfig')
    print stdout.read()
    s.close()
```

从上面的代码可以看到，我们先加载了paramiko模块，定义了三个变量。接下来，创建了一个SSHClient对象。然后告诉它加载host keys。对于Linux系统，host keys来自文件known\_host。之后就连接到SSH服务器上了。接下来就没有什么特别复杂的步骤了，尤其在熟悉了SSH之后。

现在，我们准备好远程执行命令了。对exec\_command()的调用会执行传递给它的命令并返回三个与执行命令相关的文件句柄：标准输入、标准输出和标准错误。为了演示的需要，执行命令的机器IP地址与使用SSH调用连接的IP地址是相同的。下面是远程服务器上ifconfig命令的执行结果。

```
jmjones@dinkbuntu:~/code$ python paramiko_exec.py
eth0      Link encap:Ethernet HWaddr XX:XX:XX:XX:XX:XX
          inet addr:192.168.1.15 Bcast:192.168.1.255 Mask:255.255.255.0
          inet6 addr: xx00::000:x0xx:xx0x:0x00/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
            RX packets:9667336 errors:0 dropped:0 overruns:0 frame:0
            TX packets:11643909 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:1427939179 (1.3 GiB) TX bytes:2940899219 (2.7 GiB)

lo       Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING MTU:16436 Metric:1
            RX packets:123571 errors:0 dropped:0 overruns:0 frame:0
            TX packets:123571 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:94585734 (90.2 MiB) TX bytes:94585734 (90.2 MiB)
```

这看起来与在本地机器上执行ifconfig命令的结果非常相似，只是IP地址不同而已。

例5-10演示了如何使用paramiko的SFTP来实现远程主机与本地主机之前的文件传输。这个特殊示例演示的仅是使用get()方法从远程主机下载文件。如果想向远程主机上传文件，应使用put()方法。

#### 例5-10：从SSH服务器上检索文件

```
#!/usr/bin/env python

import paramiko
import os

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
password = 'xxxYYYxxx'
dir_path = '/home/jmjones/logs'

if __name__ == "__main__":
    t = paramiko.Transport((hostname, port))
    t.connect(username=username, password=password)
    sftp = paramiko.SFTPClient.from_transport(t)
    files = sftp.listdir(dir_path)
    for f in files:
        print 'Retrieving', f
        sftp.get(os.path.join(dir_path, f), f)
    t.close()
```

如果需要使用公钥/私钥而不是密码该怎么做呢？例5-11是对上述远程执行命令程序的修改，其中使用了RSA加密。

#### 例5-11：连接SSH服务器并远程执行命令-使用私钥

```
#!/usr/bin/env python

import paramiko

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
pkey_file = '/home/jmjones/.ssh/id_rsa'

if __name__ == "__main__":
    key = paramiko.RSAKey.from_private_key_file(pkey_file)
    s = paramiko.SSHClient()
    s.load_system_host_keys()
    s.connect(hostname, port, pkey=key)
    stdin, stdout, stderr = s.exec_command('ifconfig')
    print stdout.read()
    s.close()
```

例5-12是一个sftp脚本的修改版本，其中也使用了RSA加密。

#### 例5-12：从SSH服务器上检索文件

```
#!/usr/bin/env python

import paramiko
import os

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
dir_path = '/home/jmjones/logs'
pkey_file = '/home/jmjones/.ssh/id_rsa'

if __name__ == "__main__":
    key = paramiko.RSAKey.from_private_key_file(pkey_file)
    t = paramiko.Transport((hostname, port))
    t.connect(username=username, pkey=key)
    sftp = paramiko.SFTPClient.from_transport(t)
    files = sftp.listdir(dir_path)
    for f in files:
        print 'Retrieving', f
        sftp.get(os.path.join(dir_path, f), f)
    t.close()
```

## Twisted

Twisted是一个事件驱动的Python网络框架，可以处理大量与网络相关的任务类型。一个全面独立的解决方案需要以复杂性为代价。在使用Twisted若干次之后，你会觉得它非常

好用，但是刚开始理解它确定存在困难。进一步说，学习Twisted是一个如此庞大的工程，以至于要找到一个能解决某一特定问题的切入点都有些困难。

尽管如此，我们还是强烈建议你去熟悉它，并且考虑一下它是否适合你。如果可以很容易地将固有思维转换到Twisted方式，那么学习Twisted会是一笔十分有价值的投资。由Abe Fettig (O'Reilly)编著的《Twisted Network Programming Essentials》是一本入门级的非常不错的书，可以帮助我们解决学习过程中的许多问题。

Twisted是一个事件驱动的网络，这意味着代码的编写不是集中在实现初始连接、断开连接以及数据处理等较低级的细节问题上，而是集中在编写任务的处理程序。

通过Twisted可以获得什么优势呢？Twisted框架能够帮助你将要处理的问题分成若干个小问题。而Twisted的网络连接能够帮助你无须考虑连接时需要的功能。这两个优势都能够使代码更易于复用。此外，使用Twisted，不需要考虑对一些低层的连接和错误操作进行处理。你的工作将集中在编写实现对各类事件处理的代码。

例5-13是用Twisted实现的一个端口检测程序。这是一个很基本的事例，但是在我们学习该代码后，会理解Twisted事件驱动的本质。在开始看示例之前，应该了解一些基本概念。最基本的概念包括反应器（reactor），工厂（factory），协议和延迟（deferred）。reactor是Twisted应用的主事件循环。reactor操纵事件分发，网络通信和线程。factory负责创建新的协议实例。每一个factory实例可以产生一种类型的协议。协议定义了对指定连接如何进行操作。运行时，每一个连接都会创建一个协议实例。deferred是一种链接协同行为方式。

## Twisted

大多数写代码的人对一个程序或脚本的逻辑流有非常强烈的直觉：就像水在山间流淌，自上而下。这样的代码非常容易理解，编写和调试也十分容易。Twisted代码则完全不同。作为一种异步方式，或许有人会说它更像是在低重力环境下的水滴，而不是一条沿着山坡流淌的河流。Twisted引入了新的组件：事件反应器（reactor）和friend。使用Twisted创建和调试代码，必须放弃之前的程序逻辑，建立起不同的逻辑流。

### 例5-13：Twisted实现端口检测

```
#!/usr/bin/env python

from twisted.internet import reactor, protocol
import sys

class PortCheckerProtocol(protocol.Protocol):
    def __init__(self):
```

```

        print "Created a new protocol"
def connectionMade(self):
    print "Connection made"
    reactor.stop()

class PortCheckerClientFactory(protocol.ClientFactory):
    protocol = PortCheckerProtocol
    def clientConnectionFailed(self, connector, reason):
        print "Connection failed because", reason
        reactor.stop()

if __name__ == '__main__':
    host, port = sys.argv[1].split(':')
    factory = PortCheckerClientFactory()
    print "Testing %s" % sys.argv[1]
    reactor.connectTCP(host, int(port), factory)
    reactor.run()

```

值得注意的是我们定义了Twisted类的两个子类PortCheckerProtocol和PortCheckerClientFactory。我们通过将PortCheckerProtocol指定到PortCheckerClientFactory的protocol类属性，将PortCheckerClientFactory绑定到PortCheckerProtocol。如果factory试图创建一个连接，但是失败了，factory的clientConnectionFailed()方法将被调用。ClientConnectionFailed()是一个对所有Twisted工厂来说都十分常见的方法，也是我们为factory定义的唯一方法。通过定义与factory类相关的方法，可以重载类的默认行为。在一个客户端连接失败之后，我们希望输出相关信息并停止reactor。

PortcheckerProtocol是之前讨论过的一个协议。一旦建立了一个到服务器的连接，且该服务器端口正是我们所检测的，该类的一个实例就会被创建。这里仅定义了PortCheckerProtocol类的一个方法：connectionMadw()。这是一个对所有的Twisted协议类都通用的方法。通过自己定义该方法，我们重载了默认行为。一旦成功建立了一个连接，Twisted会调用该协议的connectionMade()方法。可以看到，它输出了一个简单的信息并且停止了reactor（一会儿我们会介绍reactor）。

在这个示例中，connectionMade()和clientConnectionFailed()展示了Twisted的事件驱动本质。创建连接即为一个事件。同样，创建连接失败也是一个事件。当这些事件发生时，Twisted调用适合的方法来处理事件，这被称为事件处理函数。

在这个示例的主要部分，我们先创建了一个PortCheckerClientFactory实例，然后告诉Twisted的reactor对指定的主机和端口进行连接。主机名和端口号是使用指定的工厂从命令行获得的。在告诉reactor连接某一服务器的某一端口之后，再告诉reactor运行起来。如果没有告诉reactor运行，什么也不会发生。

总结流程顺序，我们在给出一个指令之后启动reactor。在这个示例中，指令就是连接到一台服务器的指定端口，然后使用PortCheckerClientFactory来帮助分发事

件。如果连接到指定主机和端口失败，事件循环会调用PortCheckerClientFactory的clientConnectionFailed()方法。如果连接成功，工厂会创建一个协议实例，PortcheckerProtocol，然后在该实例上调用connectionMade()方法。不管连接成功或失败，相应事件的处理函数都会关闭reactor，并且程序会停止执行。

这是一个非常基本的示例，但是它显示了Twisted事件处理的本质。有一个Twisted编程的关键概念在本例中没有涉及，那就是deferred和callback。一个deferred表示执行请求动作的承诺。callback指定了一个定义成功执行动作的方法。deferred可以连续使用，并可以将从一次使用得来的结果传递到下一次。在Twisted中这一点往往比较难理解。（例5-14将详细介绍deferred）

例5-14是一个使用Perspective Broker的示例，Perspective Broker是Twisted独有的RPC机制。这个示例是对远程ls服务器的实现，在这一章的前面部分已经在XML-RPC和Pyro中实现过了。首先，我们看服务器端实现。

#### 例5-14：Twisted 实现Perspective Broker服务器

```
→ import os
from twisted.spread import pb
from twisted.internet import reactor

class PBDirLister(pb.Root):
    def remote_ls(self, directory):
        try:
            return os.listdir(directory)
        except OSError:
            return []

    def remote_ls_boom(self, directory):
        return os.listdir(directory)

if __name__ == '__main__':
    reactor.listenTCP(9876, pb.PBServerFactory(PBDirLister()))
    reactor.run()
```

这个示例定义了一个PBDirLister类。这是一个当客户端接到服务器时，作为远端对象的Perspective Broker (PB)类。这个示例仅在这个类中定义了两个方法：remote\_ls()和remote\_ls\_boom()。remote\_ls()方法会简单地返回一个指定目录的列表。remote\_ls\_boom()能够实现与remote\_ls()相同功能，但不会理会异常操作。在示例的主要部分，我们使用Perspective Broker绑定端口9876并返回reactor。

例5-15 不是顺序的，它调用了remote\_ls()。

#### 例5-15：Twisted实现 Perspective Broker客户端

```
→ #!/usr/bin/python
```

```
from twisted.spread import pb
from twisted.internet import reactor

def handle_err(reason):
    print "an error occurred", reason
    reactor.stop()

def call_ls(def_call_obj):
    return def_call_obj.callRemote('ls', '/home/jmjones/logs')

def print_ls(print_result):
    print print_result
    reactor.stop()

if __name__ == '__main__':
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 9876, factory)
    d = factory.getRootObject()
    d.addCallback(call_ls)
    d.addCallback(print_ls)
    d.addErrback(handle_err)
    reactor.run()
```

这个客户端示例定义了三个函数，`handle_err()`，`call_ls()`和`print_ls()`。`handle_err()`将处理这一过程中出现的任何错误。`call_ls()`会初始化调用远端的`ls`方法。`print_ls()`会输出`ls`调用的结果。也就是说代码中有一个初始化远端调用的函数和一个打印调用结果的函数，这似乎有点怪异。正是因为Twisted是一个异步的事件驱动网络框架，所以才会有这样的情况产生。这一框架鼓励我们在写代码的时候，将工作分解为许多小的部分。

这个示例的主要部分展示了`reactor`如何知道什么时候调用`callback`函数。首先，创建一个客户端Perspective Broker工厂，然后告诉`reactor`连接到`localhost:9876`，并使用PB客户端工厂来处理请求。接下来，通过调用`factory.getRootObject()`，我们获得了对远端对象的控制。这实际上是一个`deferred`，因此，可以通过调用`addCallback()`来进行管道连接。

我们添加的第一个回调函数是`call_ls()`。`call_ls()`在前一步创建的`deferred`对象上调用`callRemote()`方法。`callRemote()`也会返回一个`deferred`。在过程链中第二个回调函数是`print_ls()`。当`reactor`调用了`print_ls()`，`print_ls()`将远端调用的结果打印到前一步中创建的`remote_ls()`中。事实上，是`reactor`将远端调用的结果传递到`print_ls()`的。第三个在过程链中用到的回调函数是`handle_err()`，这是一个简单的错误处理函数，它让我们知道在这一过程中是否出现了错误。一旦有错误发生或是管道达到了`print_ls()`，`respective`方法将关闭`reactor`。

下面是客户端代码运行示例：

```
jmjones@dinkgutsy:code$ python twisted_perspective_broker_client.py  
['test.log']
```

输出结果为指定目录的所有文件的列表，这与我们期望的结果一致。

相对于这里列出的简单RPC示例，这一示例看起来有些复杂。服务器端看起来相当漂亮。创建客户端的工作看起来就是一些将callbacks, deferreds, reactors和工厂通过管道进行连接处理。但是这只是一个非常简单的示例。Twisted结构的真正闪光之处只有在处理非常复杂的任务时才能充分体现。

例5-16是对之前展示的Perspective Broker客户端代码进行略微修改的版本。没有在远端服务器上调用ls，而是调用了ls\_boom。下面会展示客户端与服务器如何处理异常。

#### 例5-16：Twisted 实现Perspective Broker 客户端—异常处理

```
#!/usr/bin/python

from twisted.spread import pb
from twisted.internet import reactor

def handle_err(reason):
    print "an error occurred", reason
    reactor.stop()

def call_ls(def_call_obj):
    return def_call_obj.callRemote('ls_boom', '/foo')

def print_ls(print_result):
    print print_result
    reactor.stop()

if __name__ == '__main__':
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 9876, factory)
    d = factory.getRootObject()
    d.addCallback(call_ls)
    d.addCallback(print_ls)
    d.addErrback(handle_err)
    reactor.run()
```

下面是代码运行后的结果：

```
jmjones@dinkgutsy:code$ python twisted_perspective_broker_client_boom.py an  
error occurred [Failure instance: Traceback from remote host -- Traceback  
unavailable  
]
```

在服务器端：

```
Peer will receive following PB traceback:  
Traceback (most recent call last):  
...
```

```
<more traceback>
...
    state = method(*args, **kw)
File "twisted_perspective_broker_server.py", line 13, in remote_ls_boom
    return os.listdir(directory)
exceptions.OSError: [Errno 2] No such file or directory: '/foo'
```

错误位于服务器端代码中，而不是客户端代码中。在客户端，我们仅知道一个错误发生了。如果Pyro或是XML-RPC这样运行将是一件糟糕的事情。然而，在Twisted客户端代码中，错误处理函数被调用了。由于这是一个与Pyro和XML-RPC（基于事件）不同的模型，我们期望能够有差别地处理错误，而Perspective Broker代码正好实现了这些功能。

这里对Twisted的介绍仅是冰山一角。在刚开始使用Twisted时会觉得有些困难，因为它是非常复杂的项目和任务，是与我们通常习惯的方法不同的方法。Twisted非常值得进一步花些时间学习，也值得你把它放入工具包。

## Scapy

如果喜欢网络编程，你会喜欢上Scapy的。Scapy是一个非常便捷的交互式包操作程序和库。Scapy可以发现网络、执行扫描、跟踪路由并进行探测。Scapy同样有非常好的文档和资料可供使用。如果希望深入学习，可以购买一本有关Scapy的书来进一步了解Scapy的细节。

在写这本书时，Scapy还只是一个文件。可以在<http://hg.secdev.org/scapy/raw-file/tip/scapy.py>下载最新的版本。一旦下载了Scapy，就可以将其作为一个工具独立运行或是作为库文件加载使用。这里从交互式工具的角度来开始介绍Scapy。请记住，在运行Scapy时需要具有root权限，因为这样才能够对网络接口进行控制。

一旦下载并安装了Scapy，会看到这样的信息：

```
► Welcome to Scapy (1.2.0.2)
>>>
```

你可以实现任何通常在Python解释器中所做的操作，同时，Scapy也有其特有的命令。我们要做的第一件事就是运行Scapy的ls()函数，它列出了所有可用的层次：

```
► >>> ls()
ARP      : ARP
ASN1_Packet : None
BOOTP   : BOOTP
CookedLinux : cooked linux
DHCP     : DHCP options
DNS      : DNS
DNSQR   : DNS Question Record
DNSRR   : DNS Resource Record
```

```
Dot11      : 802.11
Dot11ATIM  : 802.11 ATIM
Dot11AssoReq : 802.11 Association Request
Dot11AssoResp : 802.11 Association Response
Dot11Auth   : 802.11 Authentication
[snip]
```

由于输出比较长，这里只截取了一部分。现在，执行一个递归DNS查询，使用加利福尼亚理工学院的公共DNS服务器对www.oreilly.com进行查询：

```
>>> sr1(IP(dst="131.215.9.49")/UDP()/DNS(rd=1,qd=DNSQR(qname="www.oreilly.com")))
Begin emission:
Finished to send 1 packets.
...
Received 4 packets, got 1 answers, remaining 0 packets
IP version=4L ihl=5L tos=0x0 len=223 id=59364 flags=DF
frag=0L ttl=239 proto=udp checksum=0xb1e src=131.215.9.49 dst=10.0.1.3 options=''
|UDP sport=domain dport=domain len=203 checksum=0x843 |
DNS id=0 qr=1L opcode=QUERY aa=0L tc=0L rd=1L ra=1L z=0L
    rcode=ok qdcount=1 ancount=2 nscount=4 arcount=3 qd=
    DNSQR qname='www.oreilly.com.' qtype=A qclass=IN |>
        an=DNSRR rrname='www.oreilly.com.' type=A rclass=IN ttl=21600 rdata='208.201.239.36'
[snip]
```

接下来，执行一个traceroute：

```
>>> ans,unans=sr(IP(dst="oreilly.com",
>>> ttl=(4,25),id=RandShort())/TCP(flags=0x2))
Begin emission:
.....*Finished to send 22 packets.
*.....*****.***.*.*.*.*.
Received 54 packets, got 22 answers, remaining 0 packets
>>> for snd, rcv in ans:
...     print snd.ttl, rcv.src, isinstance(rcv.payload, TCP)
...
[snip]
20 208.201.239.37 True
21 208.201.239.37 True
22 208.201.239.37 True
23 208.201.239.37 True
24 208.201.239.37 True
25 208.201.239.37 True
```

Scapy甚至可以实现纯粹的包复制，类似于tcpdump：

```
>>> sniff(iface="en0", prn=lambda x: x.show())
###[ Ethernet ]###
dst= ff:ff:ff:ff:ff:ff
src= 00:16:cb:07:e4:58
type= IPv4
###[ IP ]###
version= 4L
ihl= 5L
```

```
tos= 0x0
len= 78
id= 27957
flags=
frag= 0L
ttl= 64
proto= udp
chksum= 0xf668
src= 10.0.1.3
dst= 10.0.1.255
options=
[snip]
```

如果安装了graphviz和imagemagic，还可以将网络路由的追踪过程可视化。下面的示例来自Scapy官方文档：

```
>>> res,unans = traceroute(["www.microsoft.com","www.cisco.com","www.yahoo.com",
    "www.wanadoo.fr","www.pacsec.com"],dport=[80,443],maxttl=20,retry=-2)
Begin emission:
*****
Finished to send 200 packets.
*****
Begin emission:
*****
Finished to send 110 packets.
*****
Begin emission:
Finished to send 5 packets.
Begin emission:
Finished to send 5 packets.

Received 195 packets, got 195 answers, remaining 5 packets
193.252.122.103:tcp443 193.252.122.103:tcp80 198.133.219.25:tcp443 198.133.219.25:tcp80
207.46.193.254:tcp443 207.46.193.254:tcp80 69.147.114.210:tcp443 69.147.114.210:tcp80
72.9.236.58:tcp443 72.9.236.58:tcp80
```

现在，可以利用这些结果创建一幅有趣的图画：

```
>>> res.graph()
>>> res.graph(type="ps",target="| lp")
>>> res.graph(target="> /tmp/graph.svg")
```

如果已经安装了graphviz和imagemagic，网络可视化将会给你留下深刻印象。

在使用Scapy的过程中，真正有趣的是创建自定义的命令行工具和脚本。在接下来的一节中，我们将进一步介绍Scapy的库。

## 使用Scapy创建脚本

现在试着使用Scapy，模仿bat创建一个arping工具。首先看一下特定平台（platform-specific）的arping工具。

```
#!/usr/bin/env python
import subprocess
import re
import sys

def arping(ipaddress="10.0.1.1"):
    """Arping function takes IP Address or Network, returns nested mac/ip list"""

    #Assuming use of arping on Red Hat Linux
    p = subprocess.Popen("/usr/sbin/arping -c 2 %s" % ipaddress, shell=True,
                         stdout=subprocess.PIPE)
    out = p.stdout.read()
    result = out.split()
    #pattern = re.compile(":")
    for item in result:
        if ':' in item:
            print item

if __name__ == '__main__':
    if len(sys.argv) > 1:
        for ip in sys.argv[1:]:
            print "arping", ip
            arping(ip)
    else:
        arping()
```

现在，让我们看看如何以平台中立（platform-neutral）的方式使用Scapy完成同样的工作。

```
#!/usr/bin/env python
from scapy import srp,Ether,ARP,conf
import sys

def arping(iprange="10.0.1.0/24"):
    """Arping function takes IP Address or Network, returns nested mac/ip list"""

    conf.verb=0
    ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=iprange),
                   timeout=2)

    collection = []
    for snd, rcv in ans:
        result = rcv.sprintf(r"%ARP.psrc% %Ether.src%").split()
        collection.append(result)
    return collection

if __name__ == '__main__':
    if len(sys.argv) > 1:
        for ip in sys.argv[1:]:
            print "arping", ip
            print arping(ip)
    else:
        print arping()
```

可以看到，输出中所包含的信息相当清晰，是子网中所有的Mac地址和IP地址：

```
→ # sudo python scapy_arp.py
[['10.0.1.1', '00:00:00:00:00:10'], ['10.0.1.7', '00:00:00:00:00:12'],
 ['10.0.1.30', '00:00:00:00:00:11'], ['10.0.1.200', '00:00:00:00:00:13']]
```

从这些示例中，你会对Scapy的简单和易用留下深刻印象。



## 第6章

# 数据

### 引言

IT组织需要系统管理员的原因之一，是需要有专门人员能够对数据、文件及目录的处理进行控制。什么样的系统管理员不需要对目录树中的文件进行处理，不需要对文本进行解析和替换呢？如果不曾编写过一个对目录树中的所有文件进行重新命名的脚本，你或许在将来的某些时候还是需要这样做。这些能力是一名系统管理员应该具备的能力，或者至少是一名真正好的系统管理员必须具备的能力。本章接下来的内容将集中在数据、文件和目录上。

系统管理员需要不断地将数据从一个地方迁移到另一个地方。数据的迁移对于系统管理员来说是极为普通的工作。在动画业中，需要不断地将数据从一个位置迁移到另一个位置，因为数字电影产品需要数以千兆的存储空间。而且基于某一时刻观看图像时所需的不同分辨率和画面质量，不同数据有着不同的磁盘I/O。如果数据需要被迁移到HD预览房间进行数字化检测，则新被解压的或略微压缩的HD图像文件需要被移走。文件之所以常常需要被移动，是因为动画的存储设备通常有两种。既有廉价的、大的、慢的、安全的存储设备，也有快速的、昂贵的存储设备，例如JBOD或者是高速磁盘阵列RAID0。在电影工业中，主要进行数据处理的系统管理员被称为“数据牧马人”。

数据牧马人需要不断地将数据从一个地方移动和整合到另一个地方。通常，使用最为频繁的命令是rsync，scp，cp和mv。将这些简单而功能强大的工具用在Python脚本中，能够完成一些不可思议的工作。

使用标准库，可以在shell中完成许多不可思议的工作。使用标准库的好处是可以在任何地方执行数据移动脚本，而不需要依赖特定的平台。

当然，我们也不应忘记备份。只需要编写少量Python代码，就可以实现大多数自定义的备份脚本和应用程序。需要注意的是，为备份代码编写额外的测试程序不仅明智，而

且必要。应该确保在需要执行自己编写的备份脚本时，已经对该脚本进行了单元及功能测试。

此外，我们经常需要在一次移动之前，之后或移动过程中对数据进行处理。当然，Python同样完全可以胜任这一工作。创建一个删除重复工具来查找并删除重复文件，是非常有意义的。接下来我们会展示如何实现这一功能。此外，下面还将例举一个有关系统管理员经常会遇到的数据流处理的示例。

## 使用 OS 模块与Data进行交互

如果曾为编写跨平台的shell脚本而费尽心机，你会非常感谢OS模块，它是一个便携的系统服务应用程序接口（API）。在Python2.5中，OS模块包含超过200个方法，并且这些方法中的大部分都能够进行数据处理。本节会介绍该模块中的一些方法。在处理数据时，系统管理员应该更多地关注这个模块。

无论何时，如果发现自己需要去了解一个模块，IPython通常是完成这一工作的恰当工具。因此这里从使用IPython来执行一系列非常普通的操作起步，开始我们的OS模块学习之旅。

### 例6-1：浏览普通OS模块数据的方法

```
In [1]: import os  
  
In [2]: os.getcwd()  
Out[2]: '/private/tmp'  
  
In [3]: os.mkdir("/tmp/os_mod_explore")  
  
In [4]: os.listdir("/tmp/os_mod_explore")  
Out[4]: []  
  
In [5]: os.mkdir("/tmp/os_mod_explore/test_dir1")  
  
In [6]: os.listdir("/tmp/os_mod_explore")  
Out[6]: ['test_dir1']  
  
In [7]: os.stat("/tmp/os_mod_explore")  
Out[7]: (16877, 6029306L, 234881026L, 3, 501, 0, 102L,  
1207014425, 1207014398, 1207014398)  
  
In [8]: os.rename("/tmp/os_mod_explore/test_dir1",  
"/tmp/os_mod_explore/test_dir1_renamed")  
  
In [9]: os.listdir("/tmp/os_mod_explore")  
Out[9]: ['test_dir1_renamed']  
  
In [10]: os.rmdir("/tmp/os_mod_explore/test_dir1_renamed")  
  
In [11]: os.rmdir("/tmp/os_mod_explore/")
```

可以看到，加载了OS模块之后，在line[2]获得了当前的工作目录。之后，在line[3]创建了一个目录。在line[4]使用了os.listdir列出新创建目录的内容。接下来，执行了os.stat()，这与Bash中的stat命令非常相似()，之后在line[8]对目录重新命名。在line[9]，对目录是否创建进行了验证，最后使用os.rmdir方法删除了创建的目录。

无论如何，我们都需要详细地看一下OS模块。OS模块中有许多方法是处理数据时可能用到的，包括更改权限，创建符号连接等。请参考当前使用Python版本的文档，或者使用IPython的tab自动完成功能来查看OS模块的可用方法。

## 拷贝、移动、重命名和删除数据

既然在引言中已经提到了数据迁移，你现在也对如何使用OS模块有了一定认识，那么，我们再来学习一个高层模块shutil，该模块能够处理大规模数据。与OS模块十分相似，shutil模块具有复制、移动、重命名和删除数据的方法。但是它可以在整个数据树上执行操作。

在IPython中尝试使用shutil模块是一个熟悉该工具的有效途径。在下面的示例中，我们将用到shutil.copytree，而shutil还有许多其他的方法，能够完成一些不同的功能。请参考Python标准库文档，查看不同的shutil copy方法之间的差别。

### 例6-2：使用shutil模块复制数据树

```
In [1]: import os  
  
In [2]: os.chdir("/tmp")  
In [3]: os.makedirs("test/test_subdir1/test_subdir2")  
  
In [4]: ls -lR  
total 0  
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test/  
  
./test:  
total 0  
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/  
  
./test/test_subdir1:  
total 0  
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/  
  
. ./test/test_subdir1/test_subdir2:  
In [5]: import shutil  
  
In [6]: shutil.copytree("test", "test-copy")  
  
In [19]: ls -lR  
total 0  
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test/  
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test-copy/
```

```
./test:  
total 0  
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/  
  
. ./test/test_subdir1:  
total 0  
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/  
  
. ./test/test_subdir1/test_subdir2:  
  
. ./test-copy:  
total 0  
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/  
  
. ./test-copy/test_subdir1:  
total 0  
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/  
  
. ./test-copy/test_subdir1/test_subdir2:
```

很明显，本示例非常简单，但极为有用。可以非常容易地将这类代码封装成更复杂且跨平台的数据移动脚本。我们能够想到的这类代码的直接用途，是在事件驱动下将数据从一个文件系统转移到另一个文件系统。在动画环境中，经常需要等最后一帧完成之后，才能将全部图像转换成序列进行编辑。

我们可以在cron任务中编写脚本查看目录中的帧数是否达到了“X”个。当cron任务看到已经达到指定的帧数目时，可以将该目录移动到另一个帧可以在其中被处理的目录，或者直接将目录移动到具有快速I/O的磁盘上，便于回放未经压缩的HD胶片。

shutil模块不仅能够复制文件，它还有一些移动和删除数据树的方法。例6-3显示了数据树的移动，例6-4显示了数据树的删除。

#### 例6-3：使用shutil移动数据树

```
In [20]: shutil.move("test-copy", "test-copy-moved")  
  
In [21]: ls -lR  
total 0  
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test/  
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test-copy-moved/  
  
. ./test:  
total 0  
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/  
  
. ./test/test_subdir1:  
total 0  
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/  
  
. ./test/test_subdir1/test_subdir2:  
  
. ./test-copy-moved:  
total 0  
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/
```

```
./test-copy-moved/test_subdir1:  
total 0  
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/  
  
./test-copy-moved/test_subdir1/test_subdir2:
```

#### 例6-4：使用shutil删除数据树

```
In [22]: shutil.rmtree("test-copy-moved")  
  
In [23]: shutil.rmtree("test-copy")  
In [24]: ll
```

移动数据树要比删除数据树更令人激动，因为在删除之后不会有任何显示。多个这类简单的示例可以与其他的操作合并到一个更复杂的脚本中。如果要写一个备份工具来向网络存储空间复制一个目录树，并创建一个时间戳记录，这类脚本会非常有用。幸运的是，本章中正好有一个在Python中完成这类工作的示例。

## 使用路径、目录和文件

不考虑路径、目录和文件是无法处理数据的。每一名系统管理员都需要至少会写一个工具，该工具可以浏览目录，完成条件搜索，做一些有返回结果的操作。下面将要讨论一些能够实现这些操作的有趣方法。

通常，Python的标准库有一些杀死工具，可以停止作业的执行。Python最出名的就是它的“连电池都包括在内”的特性。例6-5显示了如何创建一个额外的冗余（verbose）目录遍历脚本，具有返回明确的文件、目录和路径的功能。

#### 例6-5：冗余目录遍历脚本

```
import os  
path = "/tmp"  
  
def enumeratepaths(path=path):  
    """Returns the path to all the files in a directory recursively"""  
    path_collection = []  
    for dirpath, dirnames, filenames in os.walk(path):  
        for file in filenames:  
            fullpath = os.path.join(dirpath, file)  
            path_collection.append(fullpath)  
  
    return path_collection  
  
def enumeratefiles(path=path):  
    """Returns all the files in a directory as a list"""  
    file_collection = []  
    for dirpath, dirnames, filenames in os.walk(path):  
        for file in filenames:  
            file_collection.append(file)  
  
    return file_collection
```

```
def enumeratedir(path=path):
    """Returns all the directories in a directory as a list"""
    dir_collection = []
    for dirname, dirnames, filenames in os.walk(path):
        for dir in dirnames:
            dir_collection.append(dir)
    return dir_collection

if __name__ == "__main__":
    print "\nRecursive listing of all paths in a dir:"
    for path in enumeratepaths():
        print path
    print "\nRecursive listing of all files in dir:"
    for file in enumeratefiles():
        print file
    print "\nRecursive listing of all dirs in dir:"
    for dir in enumeratedir():
        print dir
```

在一台Mac笔记本中，该脚本的输出结果如下所示：

```
[ngift@Macintosh-7][H:12022][J:0]# python enumarate_file_dir_path.py

Recursive listing of all paths in a dir:
/tmp/.aksusb
/tmp/ARD_ABJMMRT
/tmp/com.hp.launchport
/tmp/error.txt
/tmp/liten.py
/tmp/LitenDeplicationReport.csv
/tmp/ngift.liten.log
/tmp/hspfdata_ngift/58920
/tmp/launch-h360kI/Render
/tmp/launch-qy1S9C/Listeners
/tmp/launch-RTJzTw/:0
/tmp/launchd-150.wDvODl/sock

Recursive listing of all files in dir:
.aksusb
ARD_ABJMMRT
com.hp.launchport
error.txt
liten.py
LitenDeplicationReport.csv
ngift.liten.log
58920
Render
Listeners
:0
sock

Recursive listing of all dirs in dir:
.X11-unix
hspfdata_ngift
launch-h360kI
```

```
launch-qy1S9C  
launch-RTJzTw  
launchd-150.wDvODl  
ssh-YcE2t6Pfn0
```

需要注意一点，`os.walk`返回的是一个generator对象，因此，如果在调用时向`os.walk`传递了值，你可以自行遍历目录树。

```
In [2]: import os  
  
In [3]: os.walk("/tmp")  
Out[3]: [generator object at 0x508e18]
```

这是在IPython中运行示例的情形。注意，使用generator使我们能够调用`path.next()`。这里不会介绍generator的细节，但需要知道`os.walk`返回的是一个generator对象。generator对于系统编程非常有用。请访问David Beazley的网站 (<http://www.dabeaz.com/generators/>)，在那里你可以找到需要了解的相关内容。

```
In [2]: import os  
  
In [3]: os.walk("/tmp")  
Out[3]: [generator object at 0x508e18]  
  
In [4]: path = os.walk("/tmp")  
  
In [5]: path.  
path.__class__          path.__init__      path.__repr__      path.gi_running  
path.__delattr__        path.__iter__       path.__setattr__   path.next  
path.__doc__            path.__new__        path.__str__       path.send  
path.__getattribute__  path.__reduce__     path.close        path.throw  
path.__hash__           path.__reduce_ex__ path.gi_frame  
  
In [5]: path.next()  
Out[5]:  
('/tmp',  
 ['.X11-unix',  
 'hsperfdata_ngift',  
 'launch-h36okI',  
 'launch-qy1S9C',  
 'launch-RTJzTw',  
 'launchd-150.wDvODl',  
 'ssh-YcE2t6Pfn0'],  
 ['.aksusb',  
 'ARD_ABJMMRT',  
 'com.hp.launchport',  
 'error.txt',  
 'liten.py',  
 'LitenDeplicationReport.csv',  
 'ngift.liten.log'])
```

再来多看一点有关generator的细节。让我们首先创建一个cleaner模块，该模块将通过API返回文件、目录及路径。

现在，我们遍历了一个非常基本的目录，接下来要考虑的是如何将其变成一个面向对象的模块，以便于加载和复用。也许需要花些工夫才能创建一个固定编码的模块，但是一旦完成创建，就可以将其作为可复用的通用模块，使用起来非常简单方便。参见例6-6。

#### 例6-6：创建可复用的目录浏览模块

```
→ import os

class diskwalk(object):
    """API for getting directory walking collections"""
    def __init__(self, path):
        self.path = path

    def enumeratePaths(self):
        """Returns the path to all the files in a directory as a list"""
        path_collection = []
        for dirpath, dirnames, filenames in os.walk(self.path):
            for file in filenames:
                fullpath = os.path.join(dirpath, file)
                path_collection.append(fullpath)

        return path_collection

    def enumerateFiles(self):
        """Returns all the files in a directory as a list"""
        file_collection = []
        for dirpath, dirnames, filenames in os.walk(self.path):
            for file in filenames:
                file_collection.append(file)

        return file_collection

    def enumerateDir(self):
        """Returns all the directories in a directory as a list"""
        dir_collection = []
        for dirpath, dirnames, filenames in os.walk(self.path):
            for dir in dirnames:
                dir_collection.append(dir)

        return dir_collection
```

可以看到，进行少量修改之后，能够创建一个非常好的接口，以方便将来进一步修改。这个新模块的优点之一是可以将其载入到其他脚本中。

## 数据比较

数据比较对系统管理员非常重要。你或许会经常问自己，“这两个目录中的文件到底有什么差别？系统中有多少个重复的文件存在？”在这一节中，你会找到这些问题或是类似问题的答案。

在处理大批量的重要数据时，对目录树和文件进行比较以找到其中的差异是非常重要的。如果正要编写大块数据的迁移脚本，这会变得十分关键。一种非常糟糕的情况就是编写了一个会损坏关键生产数据的大数据块迁移脚本。

本节首先介绍一些轻量级的文件和目录进行比较的方法，最后介绍如何对文件进行校验比较。Python标准库有许多模块支持比较操作，这里将介绍`filecmp`和`os.listdir`。

## 使用`filecmp`模块

`filecmp`模块包括执行快速有效的文件和目录比较函数。`filecmp`模块会在两个文件上执行`os.stat`，如果`os.stat`对两个文件的判断结果相同，则返回`True`，如果结果不同，则返回`False`。典型地，`os.stat`判断两个文件是否使用同一磁盘上相同的i节点，或者它们的大小是否相等，但实际上并没有比较文件的内容。

要全面理解`filecmp`如何工作，需要创建三个新文件。先将目录切换到`/tmp`目录下，创建一个名为`file0.txt`的文件，并在文件中写入数字“0”。接下来，创建名为`file1.txt`的文件，并在该文件中写入数字“1”。最后，创建名为`file00.txt`的文件，并写入数字“0”。我们将在下面的示例代码中使用这几个文件。

```
In [1]: import filecmp  
In [2]: filecmp.cmp("file0.txt", "file1.txt")  
Out[2]: False  
  
In [3]: filecmp.cmp("file0.txt", "file00.txt")  
Out[3]: True
```

可以看到，在对`file0.txt`和`file00.txt`进行比较时，`cmp`函数返回`True`。在对`file1.txt`和`file0.txt`进行比较时返回`False`。

`dirncmp`函数有一些属性，这些属性可以报告目录树之间的差异。这里不会对每一个属性都进行介绍，但是我们创建了一些非常有用的示例。例如，在目录`/tmp`中创建两个子目录，从先前的示例中复制文件到每一个目录中。在`dirB`中，我们创建了另一个名为`file11.txt`的文件，并且在其中放入数字“11”：

```
In [1]: import filecmp  
In [2]: pwd  
Out[2]: '/private/tmp'  
  
In [3]: filecmp.dircmp("dirA", "dirB").diff_files  
Out[3]: []  
  
In [4]: filecmp.dircmp("dirA", "dirB").same_files  
Out[4]: ['file1.txt', 'file00.txt', 'file0.txt']
```

```
In [5]: filecmp.dircmp("dirA", "dirB").report()
diff dirA dirB
Only in dirB : ['file11.txt']
Identical files : ['file0.txt', 'file00.txt', 'file1.txt']
```

你或许有点奇怪，尽管我们创建了*file11.txt*文件，且该文件中的信息是独一无二的，但是*diff\_files*没有匹配。原因是*diff\_files*仅对文件名相同的文件进行比较。

接下来查看一下*same\_files*的输出结果，注意，它仅对两个目录中的相同文件进行报告。最后，我们对上一示例生成了一个报告，包括了两个目录之间差异的细目。这里仅是对*filecmp*模块功能作了一个简单介绍，因此我们建议你查阅Python标准库文档从而获得全面的知识，以弥补本书中介绍的不足。

## 使用os.listdir

另外一个轻量级的目录比较方法是使用*os.listdir*。你可以将*os.listdir*视为*ls*命令，该命令返回找到的文件列表。因为Python支持许多可以对列表进行处理的有趣的方法，你可以通过将列表转换为一个集合后从一个集合中去掉另一个，使用*os.listdir*来查看目录自身的差异。下面是一个示例，显示了在IPython中是如何实现的：

```
In [1]: import os
In [2]: dirA = set(os.listdir("/tmp/dirA"))
In [3]: dirA
Out[3]: set(['file1.txt', 'file00.txt', 'file0.txt'])
In [4]: dirB = set(os.listdir("/tmp/dirB"))
In [5]: dirB
Out[5]: set(['file1.txt', 'file00.txt', 'file11.txt', 'file0.txt'])
In [6]: dirA - dirB
Out[6]: set([])
In [7]: dirB-dirA
Out[7]: set(['file11.txt'])
```

这个示例中，我们使用了一个简洁的技巧来将两个列表转换为集合，然后对集合进行减操作来查找差异。注意，line[7]之所以返回了*file11.txt*，是因为*dirB*是*dirA*的超集。而在line[6]中之所以结果为空，是因为*dirA*包含了*dirB*的全部内容。使用集合也可以方便地创建两个数据结构的合并，具体做法是：参照另一个目录减去一个目录的全路径，然后对差异进行复制。我们在下一章中将讨论数据的合并。

但是，这个方法也有很大的局限性。当两个文件同名时会导致错误。例如一个文件只有0字节，而另一个同名的文件却有200G。在下一节中，我们将介绍一个更好的查找两个目录之间差异并进行内容合并的方法。

# 合并数据

如果不想简单地对数据文件进行比较，而是想对两个目录树进行合并，该怎么办呢？当要把一个目录树的内容合并到另一个目录树，而不创建任何副本时，这是经常会出现的问题。

你可以直接复制一个目录中的文件到目标目录，然后对目录进行复制，但是在第一步不进行复制将会效率更高。一个合理而简单的解决方法是使用`filecmp`模块的`dircmp`函数来对两个目录进行比较，并使用之前介绍的`os.listdir`复制唯一的结果。还可以使用MD5校验，在接下来一节中我们将介绍MD5校验。

## MD5 校验和比较

对一个文件执行MD5检验和（checksum）并拿它与另一个文件比较就像是使用火箭筒向目标射击一样。虽然逐字节地比较是百分之百地精确，但也无法匹敌MD5检验和这个重量级武器。例6-7显示了该函数如何获得指定文件路径，并返回校验和。

### 例6-7：执行文件MD5校验和

```
→ import hashlib

def create_checksum(path):
    """
    Reads in file. Creates checksum of file line by line.
    Returns complete checksum total for file.
    """
    fp = open(path)
    checksum = hashlib.md5()
    while True:
        buffer = fp.read(8192)
        if not buffer:break
        checksum.update(buffer)
    fp.close()
    checksum = checksum.digest()
    return checksum
```

下面是一个迭代示例，该示例在IPython中使用该函数对两个文件进行比较：

```
→ In [2]: from checksum import createChecksum

In [3]: if createChecksum("image1") == createChecksum("image2"):
...:     print "True"
...:
...:
True

In [5]: if createChecksum("image1") == createChecksum("image_unique"):
print "True"
```

...:

...:

在示例中，文件的校验是通过手工对比完成的。但是我们也可以使用之前编写的代码，该代码返回路径列表，对目录树进行迭代比较并得到副本数。此外，还可以创建API，通过IPython交互测试我们的解决方案，如果证明确实可行，就可以创建另一个模块。例6-8展示了查找重复的代码。

#### 例6-8：执行目录MD5校验和以查找重复

```
In [1]: from checksum import createChecksum  
In [2]: from diskwalk_api import diskwalk  
In [3]: d = diskwalk('/tmp/duplicates_directory')  
In [4]: files = d.enumeratePaths()  
In [5]: len(files)  
Out[5]: 12  
In [6]: dup = []  
In [7]: record = {}  
In [8]: for file in files:  
    compound_key = (getsize(file),create_checksum(file))  
    if compound_key in record:  
        dup.append(file)  
    else:  
        record[compound_key] = file  
....:  
....:  
In [9]: print dup  
['/tmp/duplicates_directory/image2']
```

这段代码中，仅有一段在之前的示例中没有介绍过，也就是line[8]。创建一个空字典，然后使用一个关键字来存储产生的校验结果，这是用来判断检验和是否已存在的简单方法。如果存在，将文件加入重复列表。现在，将代码分为可再次使用的若干段。例6-9显示了如何实现。

#### 例6-9：查找重复

```
from checksum import create_checksum  
from diskwalk_api import diskwalk  
from os.path import getsize  
  
def findDuplicates(path = '/tmp'):  
    dup = []  
    record = {}  
    d = diskwalk(path)  
    files = d.enumeratePaths()
```

```
for file in files:
    compound_key = (getsize(file),create_checksum(file))
    if compound_key in record:
        dup.append(file)
    else:
        #print "Creating compound key record:", compound_key
        record[compound_key] = file
return dup

if __name__ == "__main__":
dups = findDuplicates()
for dup in dups:
    print "Duplicate: %s" % dup
```

执行该脚本时，得到的输出如下：

```
[ngift@Macintosh-7]:~$ python find_dupes.py
Duplicate: /tmp/duplicates_directory/image2
```

从这里你可以看到，哪怕是非常小的一段代码也可以复用成功。现在有了一个通用模块，可以取得目录树并返回所有重复文件的列表。这用起来非常方便。接下来我们再进一步，实现自动删除重复的功能。

正如你可以使用`os.remove(file)`一样，在Python中删除文件非常简单。这个示例中，在`/tmp`目录中有一个10MB大小的文件，我们使用`os.remove(file)`删除其中的一个：

```
In [1]: import os

In [2]: os.remove("10
10mbfile.0 10mbfile.1 10mbfile.2 10mbfile.3 10mbfile.4
10mbfile.5 10mbfile.6 10mbfile.7 10mbfile.8

In [2]: os.remove("10mbfile.1")

In [3]: os.remove("10
10mbfile.0 10mbfile.2 10mbfile.3 10mbfile.4 10mbfile.5
10mbfile.6 10mbfile.7 10mbfile.8
```

值得注意的是，IPython中的tab自动完成功能允许我们查看匹配并为我们显示图像文件的名称。`os.remove(file)`方法是静态的、持久的。这或许是你希望的，但也可能不是。记住这一点，可以实现一个简单的方法来删除重复的文件，之后再对其进行修改。由于在IPython中测试交互代码十分简单，我们写一个`test`函数，然后进行测试：

```
In [1]: from find_dupes import findDuplicates

In [2]: dups = findDuplicates("/tmp")

In [3]: def delete(file):
            import os
...:     print "deleting %s" % file
...:     os.remove(file)
```

```
...:  
...:  
  
In [4]: for dupe in dupes:  
...:     delete(dupe)  
...:  
...:  
In [5]: for dupe in dupes:  
delete(dupe)  
...:  
...:  
deleting /tmp/10mbfile.2  
deleting /tmp/10mbfile.3  
deleting /tmp/10mbfile.4  
deleting /tmp/10mbfile.5  
deleting /tmp/10mbfile.6  
deleting /tmp/10mbfile.7  
deleting /tmp/10mbfile.8
```

在这个示例中，通过添加打印自动删除文件的语句，增加了delete模块的复杂度。我们已经创建了不少可复用的代码，但不会马上停止。下一步还要创建另一个模块，该模块可以作为一个file对象，执行相关删除操作。它甚至不需要查找重复，可以用来删除任何内容。参见例6-10。

#### 例6-10：删除模块

```
#!/usr/bin/env python  
import os  
  
class Delete(object):  
    """Delete Methods For File Objects"""  
  
    def __init__(self, file):  
        self.file = file  
  
    def interactive(self):  
        """interactive deletion mode"""  
  
        input = raw_input("Do you really want to delete %s [N]/Y" % self.file)  
        if input.upper():  
            print "DELETING: %s" % self.file  
            status = os.remove(self.file)  
        else:  
            print "Skipping: %s" % self.file  
        return  
    def dryrun(self):  
        """simulation mode for deletion"""  
  
        print "Dry Run: %s [NOT DELETED]" % self.file  
        return  
  
    def delete(self):  
        """Performs a delete on a file, with additional conditions  
        """
```

```

print "DELETING: %s" % self.file
try:
    status = os.remove(self.file)
except Exception, err:
    print err
    return status

if __name__ == "__main__":
    from find_dupes import findDupes
    dupes = findDupes('/tmp')

for dupe in dupes:
    delete = Delete(dupe)
    #delete.dryrun()
    #delete.delete()
    #delete.interactive()

```

在这个模块中，你将看到三种不同的删除类型。交互（interactive）删除方法提示用户对每一个即将删除的文件进行确认。这似乎有点烦人，但是当其他程序员维护或升级代码时这是非常好的保护措施。

Dry run方法模拟了删除操作。实际上，该方法最终执行了删除方法，能够永久地删除文件。在模块的底部，有一个被注释掉的示例，该示例展示了如何使用这三种不同的删除方法。下面的示例演示了每一方法的执行过程：

- Dry run



```

ngift@Macintosh-7[H:10197][J:0]# python delete.py
Dry Run: /tmp/10mbfile.1 [NOT DELETED]
Dry Run: /tmp/10mbfile.2 [NOT DELETED]
Dry Run: /tmp/10mbfile.3 [NOT DELETED]
Dry Run: /tmp/10mbfile.4 [NOT DELETED]
Dry Run: /tmp/10mbfile.5 [NOT DELETED]
Dry Run: /tmp/10mbfile.6 [NOT DELETED]
Dry Run: /tmp/10mbfile.7 [NOT DELETED]
Dry Run: /tmp/10mbfile.8 [NOT DELETED]

```

- Interactive



```

ngift@Macintosh-7[H:10201][J:0]# python delete.py
Do you really want to delete /tmp/10mbfile.1 [N]/YY
DELETING: /tmp/10mbfile.1
Do you really want to delete /tmp/10mbfile.2 [N]/Y
Skipping: /tmp/10mbfile.2
Do you really want to delete /tmp/10mbfile.3 [N]/Y

```

- Delete



```

[ngift@Macintosh-7[H:10203][J:0]# python delete.py
DELETING: /tmp/10mbfile.1
DELETING: /tmp/10mbfile.2
DELETING: /tmp/10mbfile.3
DELETING: /tmp/10mbfile.4

```

```
DELETING: /tmp/10mbfile.5
DELETING: /tmp/10mbfile.6
DELETING: /tmp/10mbfile.7
DELETING: /tmp/10mbfile.8
```

在处理数据的时候，你或许会发现使用封装技术非常简便。因为这样可以通过充分地抽象现有的工作，使问题一般化，从而避免可能出现的新问题。在这种情况下，为了自动删除重复文件，创建了一个普通的查找和删除文件的模块。还可以创建另一个工具来获得文件对象并应用某种形式的压缩。我们实际上仅介绍了该示例的一小部分。

## 对文件和目录的模式匹配

到目前为止，已经介绍了如何处理目录和文件，并执行查找重复、删除目录、移动目录等操作。要掌握了目录树之后，接下来要学习使用模式进行匹配，不论是单独使用还是与之前的技术联合使用。正如Python中的其他操作一样，对文件扩展名或文件名执行模式匹配也十分简单。这一节将介绍通用模式匹配的问题，应用之前使用的技术来创建简单但却功能强大的复用工具。

系统管理员们有一个常见的问题要解决，那就是他们需要追踪、删除、移动、重命名或是复制某一类型的文件。在Python中，最直接的方法是使用fnmatch模块或是glob模块。这两个模块之间的主要差异是，对于Unix通配符，fnmatch返回True或False，而glob返回匹配模式的路径列表。作为选择，正则表达式可以用来创建一些更复杂的匹配工具。请参考第3章以获得更多关于使用正则表达式来匹配模式的说明。

例6-11演示了fnmatch和glob是如何使用的。这里将通过从diskwalk\_api模块加载diskwalk复用之前的代码。

### 例6-11：分别使用fnmatch 和glob查找文件匹配

```
In [1]: from diskwalk_api import diskwalk
In [2]: files = diskwalk("/tmp")
In [3]: from fnmatch import fnmatch
In [4]: for file in files:
...:     if fnmatch(file,"*.txt"):
...:         print file
...:
...:
/tmp/file.txt
In [5]: from glob import glob
In [6]: import os
In [7]: os.chdir("/tmp")
```

```
In [8]: glob("*")
Out[8]: ['file.txt', 'image.iso', 'music.mp3']
```

在示例中，复用了之前的diskwalk模块后得到一个列表，列表中包含/tmp目录中的全路径。之后，使用fnmatch来决定是否每个文件都可以匹配模式“\*.txt”。glob模块则不同，它是严格地“全局”匹配一个模式，并返回全路径。glob是一个比fnmatch更高级的函数。对于差异比较而言，两者都是非常有用的工具。

当与其他代码联合使用，创建一个在目录树中搜索数据的过滤器时，fnmatch函数尤其有用。在处理目录树时，你往往会对匹配某些模式的文件进行处理。明白这一点之后，我们就可以解决一个经典的系统管理问题，即对目录树中的所有匹配文件重命名。记住，重命名实际上与删除、压缩或对文件进行处理一样简单。其模式如下：

1. 取得目录中某个文件的路径。
2. 执行某种过滤器的可选类别；这会涉及许多过滤器，如文件名、扩展名、大小、唯一值等。
3. 在过滤器上执行操作，包括复制、删除，压缩，读取等。例6-12演示了这操作。

#### 例6-12：将目录树中全部MP3文件重命名为文本文件

```
► In [1]: from diskwalk_api import diskwalk
In [2]: from shutil import move
In [3]: from fnmatch import fnmatch
In [4]: files = diskwalk("/tmp")
In [5]: for file in files:
        if fnmatch(file, "*.mp3"):
            #here we can do anything we want, delete, move, rename...hmmm rename
            move(file, "%s.txt" % file)
In [6]: ls -l /tmp/
total 0
-rw-r--r-- 1 ngift wheel 0 Apr 1 21:50 file.txt
-rw-r--r-- 1 ngift wheel 0 Apr 1 21:50 image.iso
-rw-r--r-- 1 ngift wheel 0 Apr 1 21:50 music.mp3.txt
-rw-r--r-- 1 ngift wheel 0 Apr 1 22:45 music1.mp3.txt
-rw-r--r-- 1 ngift wheel 0 Apr 1 22:45 music2.mp3.txt
-rw-r--r-- 1 ngift wheel 0 Apr 1 22:45 music3.mp3.txt
```

通过使用上面的示例，我们只用了四行简单的Python代码，就将一个由mp3组成的目录树中的全部文件重命名为文本文件。如果你是一个没有阅读过BOFH或Bastard Operator From Hell中任何一个章节的系统管理员，那么，可能对这些代码没有明显的感觉。

想象一下，有一个生产文件服务器，具有高性能的I/O存储功能，但却容量有限。由于一两个不良用户在里面放了几百GB的 MP3文件导致它的空间经常被占满。你可以对每一

个用户所使用的文件空间大小指定一个配额，但是这会引来更多麻烦。一个解决方案是每晚创建一个cron作业，查找所有的MP3文件并对它们做一些操作。每周一，将它们重命名为文本文件，每周二将它们压缩为ZIP文件，每周三，则将它们移动到/tmp目录中，到了星期四，则会进行删除，并给文件的所有者发送一个已删除的MP3文件列表。当然，除非你是公司的老板，否则我们不会建议你真的这么去做。而实际上，在BOFH，早期的代码就是这样进行操作的。

## 包装rsync

你可能已经知道，rsync是一个最初由Andrew Tridgell和Mackerra编写的命令行工具。2007年下半年，rsync version 3测试版发布，它包括了比原始版本更多的类型选项。

多年来，我们都使用rsync作为把数据从点A移动到点B的主要工具。我们建议你仔细阅读rsync的帮助和选项，因为rsync可能会是为系统管理员编写的最有用的命令行工具。

正如前面提到的，Python可以帮助控制或粘贴rsync的方法。有一个问题是确保数据在计划的时间内被复制。在需要对文件服务器之间进行数TB的数据同步的情况下，我们不希望去手工监视复制的进程，这也就是Python真正发挥作用的时候。

你可以使用Python为同步添加一定的智能，或是根据需要进行自定义。将Python作为glue代码使用能够让UNIX工具能够完成通常情况下难以完成的工作，你可以创建各种高灵活性和可定制的工具，正所谓想象力有多远，就可以走多远。例6-13显示了一个非常简单的示例，演示了如何对rsync进行包装（wrap）。

### 例6-13：rsync的简单包装

```
#!/usr/bin/env python
#wraps up rsync to synchronize two directories
from subprocess import call
import sys

source = "/tmp/sync_dir_A/" #Note the trailing slash
target = "/tmp/sync_dir_B"
rsync = "rsync"
arguments = "-a"
cmd = "%s %s %s %s" % (rsync, arguments, source, target)

def sync():
    ret = call(cmd, shell=True)
    if ret !=0:
        print "rsync failed"
        sys.exit(1)
sync()
```

这个示例是一个对两个目录进行同步的代码，如果命令执行失败，将打印失败信息。可以做一些更有趣的事，解决自己会频繁遇到的问题。我们经常需要对两个非常大的目录进行同步，但不想整晚都去监视数据同步过程。然而，如果没有监视同步过程，一旦进程在执行过程中被破坏、数据也被破坏、整晚时间都被浪费，后果不堪设想，处理过程也需要在第二天重新开始。这种情况下，可以使用Python创建一个更先进的、高度机动的rsync命令。

一个高度机动的rsync命令又会如何去做呢？如果你正在监视两个目录之间的同步，它能完成所有需要做的工作：它会一直对目录进行同步，直到完成，然后会发送一个邮件告之任务已完成。例6-14中的 rsync代码实现了这一功能。

#### 例6-14：一个直至工作完成才结束的rsync命令

```
#!/usr/bin/env python
#wraps up rsync to synchronize two directories
from subprocess import call
import sys
import time

"""this motivated rsync tries to synchronize forever"""

source = "/tmp/sync_dir_A/" #Note the trailing slash
target = "/tmp/sync_dir_B"
rsync = "rsync"
arguments = "-av"
cmd = "%s %s %s %s" % (rsync, arguments, source, target)

def sync():
    while True:
        ret = call(cmd, shell=True)
        if ret !=0:
            print "resubmitting rsync"
            time.sleep(30)
        else:
            print "rsync was successful"
            subprocess.call("mail -s 'jobs done' bofh@example.com", shell=True)
            sys.exit(0)
sync()
```

这个示例十分简单，并且包含了部分硬编码数据，但是这是一个非常有用工具，使用该工具可以使一些需要人工完成的事情自动化。此外，你还可以添加一些其他特性，包括设置下次尝试的时间间隔，连接次数，以及检测所连接主机的磁盘空间使用情况等。

## 元数据：关于数据的数据

绝大多数系统管理员所关注的不仅局限于数据，还包括与数据相关的数据。元数据，或者说是关于数据的数据，通常比数据本身更为重要。这里给出一个示例，在电影和电视

中，相同的数据经常出现在文件系统的多个位置或者是多个文件系统上。跟踪这些数据经常涉及创建元数据类型管理系统。

描述文件是如何组织和使用的数据，对于应用程序、动画生产线或是恢复一个备份来说，都是十分关键的。Python可以在这方面有所作为，因为，在Python中使用元数据和写元数据都非常简单。

现在看一个使用ORM，SQLAlchemy创建的关于文件系统的元数据。幸运的是，SQLAlchemy的文档非常好，并且SQLAlchemy可以与SQLite一起使用。我们认为，这对于创建自定义的元数据解决方案是一个非常棒的组合。

在上面的示例中，我们体验了实时搜索文件系统，执行操作和查询路径。搜索由几百万个文件组成的大文件系统是非常耗时的，因此实时搜索非常有意义。例6-15中通过将ORM和前面介绍的目录遍历技术结合，展示了一个基本的元数据系统。

#### 例6-15：利用SQLAlchemy创建文件系统的元数据

```
#!/usr/bin/env python
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey
from sqlalchemy.orm import mapper, sessionmaker
import os

#path
path = "/tmp"

#Part 1: create engine
engine = create_engine('sqlite:///memory:', echo=False)

#Part 2: metadata
metadata = MetaData()

filesystem_table = Table('filesystem', metadata,
    Column('id', Integer, primary_key=True),
    Column('path', String(500)),
    Column('file', String(255)),
)
metadata.create_all(engine)

#Part 3: mapped class
class Filesystem(object):

    def __init__(self, path, file):
        self.path = path
        self.file = file

    def __repr__(self):
        return "[Filesystem('%s','%s')]" % (self.path, self.file)

#Part 4: mapper function
mapper(Filesystem,filesystem_table)
```

```
#Part 5: create session
Session = sessionmaker(bind=engine, autoflush=True, transactional=True)
session = Session()

#Part 6: crawl file system and populate database with results
for dirname, dirnames, filenames in os.walk(path):
    for file in filenames:
        fullpath = os.path.join(dirname, file)
        record = Filesystem(fullpath, file)
        session.save(record)

#Part 7: commit to the database
session.commit()

#Part 8: query
for record in session.query(Filesystem):
    print "Database Record Number: %s, Path: %s , File: %s " \
        % (record.id, record.path, record.file)
```

可以将代码视为过程的集合，一段代码可以看作由一个过程接着另一个过程组成。在第一部分，创建了一个引擎，这是定义将要使用的数据库的非常不错的方法。在第二部分，定义了一个元数据实例，并且创建了数据库表。在第三部分，创建了一个类，该类会映射到所创建的数据库中的数据表。在第四部分，调用放入ORM的映射函数，它实际上已经将类映射到数据表上了。在第五部分，创建到数据库的会话。需要注意的是，这里设置了一些关键字参数，包括`autoflush`和`transactional`。

现在已经非常明确地描述了ORM的安装，在第六部分，完成常规操作，在遍历一个目录树时，获得文件名和完整路径。尽管这里会有一些改动。注意，这里在数据库中为每一个完整路径和每一个文件创建了一个记录，并且在创建时，保存了每一个新创建的记录。之后在第七部分，提交这一处理到SQLite数据库中。

最后，在第八部分，在Python中执行了一个查询，返回了在数据库中放置的记录结果。这一示例创建自定义SQLAlchemy元数据解决方案，对你的公司和客户而言都是非常好的方法。也可以扩展这一代码来做一些更有趣的事，如执行相关查询或将结果写入文件等。

## 存档、压缩、映像和恢复

处理大量数据是系统管理员每天都要面对的问题。他们通常会使用tar、dd、gzip、bzip、bzip2、hdutil、asr以及其他工具来完成这些工作。

不管你信不信，“连电池都包括在内”的Python标准库对TAR文件、zlib文件和gzip文件都提供内建支持。如果压缩和归档是你的目标，那么Python为此提供了丰富的工具。让我们来看一个重要的归档包：tar，并且一起了解标准库是如何实现tar的。

# 使用tarfile模块创建TAR归档

创建一个TAR归档包非常简单，甚至显得过于简单了。在例6-16中，作为示例，我们创建了一个非常大的文件。这里用到的语法对使用者非常友好，在这一点上甚至超过了tar命令本身。

## 例6-16：创建大文本文件

```
▶ In [1]: f = open("largeFile.txt", "w")

In [2]: statement = "This is a big line that I intend to write over and over again."
In [3]: x = 0
In [4]: for x in xrange(20000):
....:     x += 1
....:     f.write("%s\n" % statement)
....:
....:
In [4]: ls -l
-rw-r--r-- 1 root root 1236992 Oct 25 23:13 largeFile.txt
```

好了，现在有了一个大文件，让我们对其使用TAR命令。参见例6-17。

## 例6-17：对文件内容使用TAR命令

```
▶ In [1]: import tarfile

In [2]: tar = tarfile.open("largefile.tar", "w")

In [3]: tar.add("largeFile.txt")

In [4]: tar.close()

In [5]: ll
-rw-r--r-- 1 root root 1236992 Oct 25 23:15 largeFile.txt
-rw-r--r-- 1 root root 1236992 Oct 26 00:39 largefile.tar
```

可以看到，与普通的tar命令相比，vanilla TAR归档有更为简单的语法。这确实也令使用IPython shell来完成每天全部的系统管理工作成为可能。

由于使用Python来创建TAR文件非常方便，因此仅对一个文件使用TAR，几乎没有什么意义。使用本章中多次用到的目录遍历模式，我们对/tmp目录创建了一个TAR文件，实现方法是遍历目录树然后将每一个文件添加到/tmp目录的归档包中。参见例6-18。

## 例6-18：对目录树使用TAR命令

```
▶ In [27]: import tarfile

In [28]: tar = tarfile.open("temp.tar", "w")

In [29]: import os
```

```
In [30]: for root, dir, files in os.walk("/tmp"):
....:     for file in filenames:
....:
KeyboardInterrupt

In [30]: for root, dir, files in os.walk("/tmp"):
for file in files:
....:     fullpath = os.path.join(root,file)
....:     tar.add(fullpath)
....:
....:

In [33]: tar.close()
```

通过遍历目录来添加目录树中的内容非常简单方便，因为它可以与本章中介绍的其他技术相结合。你或许正在归档一个全部由媒体文件组成的目录。如果对重复的文件进行归档，显然有些不明智，因此，在创建TAR文件之前，需要将重复的文件替换为符号链接文件或做其他需要进行的操作。使用本章介绍的方法，可以很容易地编写实现上述目标的代码，从而节省一些磁盘空间。

由于创建一个普通的TAR文档有些乏味，让我们换个口味使用bzip2压缩，它会使你的CPU全速工作，甚至会抱怨怎么会有这么多工作。bzip2压缩算法就十分了不起。下面看一个示例，相信它会给我们留下深刻印象。

#### 例6-19：创建bzip2 的TAR 归档

```
► In [1]: tar = tarfile.open("largefilecompressed.tar.bzip2", "w|bz2")

In [2]: tar.add("largeFile.txt")

In [3]: ls -h
foo1.txt fooDir1/ largeFile.txt largefilecompressed.tar.bzip2*
foo2.txt fooDir2/ largefile.tar

In [4]: tar.close()

In [5]: ls -lh
-rw-r--r-- 1 root root 61M Oct 25 23:15 largeFile.txt
-rw-r--r-- 1 root root 61M Oct 26 00:39 largefile.tar
-rwxr-xr-x 1 root root 10K Oct 26 01:02 largefilecompressed.tar.bzip2*
```

bzip2可以将61M的文本文件压缩到只有10K大小，多么令人惊叹。当然这也不是零代价的，因为即使在双核AMD系统上，它也会花几分钟时间才能完成文件压缩。

让我们总结一下，然后使用其他可用的选项进行另一次压缩归档。接下来使用gzip。gzip的语法略有不同。参见例6-20。

### 例6-20：创建gzip的TAR存档

```
In [10]: tar = tarfile.open("largefile.tar.gz", "w|gz")
In [11]: tar.add("largeFile.txt")
In [12]: tar.close()
In [13]: ls -lh
-rw-r--r-- 1 root root 61M Oct 26 01:20 largeFile.txt
-rw-r--r-- 1 root root 61M Oct 26 00:39 largefile.tar
-rwxr--r-x 1 root root 160K Oct 26 01:24 largefile.tar.gz*
```

这个gzip文件依然是难以置信地小，只有160K左右。但是在我的机器上，完成压缩TAR包的创建只用了几秒钟时间。看起来在大多数环境中，gzip都非常适用。

## 使用tarfile模块检查TAR文件内容

现在我们有了创建TAR文件的工具，接下来，对TAR文件的内容进行检查就变得非常有意义了。盲目地创建TAR文件却不做内容的检查不是一件好事。作为一名系统管理员，无论你干了多久，都有可能会为一个坏的备份而气恼或是因创建了一个坏的备份而被指责。

为了将这一问题讲得更生动，进一步强调检验TAR文档内容的重要性，这里讲一个朋友的故事，让我们称之为“TAR文档丢失事件”。故事中人物的姓名、身份和事情都是虚构的；如果这个故事与现实生活雷同，纯属偶然。

这个朋友作为系统管理员，工作在第一流的电视演播室，为一个由某个狂人领导的部门提供支持服务。这个领导以不讲真话，行事冲动，甚至有些疯狂出名。一旦出现由他导致的错误，例如错过了客户的最后期限，不能按指定的说明书完成工作，他会很容易地撒谎并将责任怪罪到其他人头上。这个替罪羊经常是我们的朋友，系统管理员。

不幸的是，这个朋友一直充当着替罪羊的角色。他也曾想过离开，再找一份新的工作，但是，他已经在这个演播室工作许多年了，在这里有许多朋友，他不想因为一些临时的坏遭遇而失去所有的一切。于是他决定从最基本的工作做起，并为此建立了一个日志系统，可以对那个狂人的所有文档自动生成TAR文档，并进行分类。他认定，那个狂人迟早会因为错过了某个最后期限，需要找个借口怪罪到他头上。

一天，我们的朋友Willian从他的老板那里接到一个电话，“Willian，我需要你立刻到我的办公室来，备份出了点问题。” Willian立即走进他的办公室，被告之那个狂人Alex，责怪Willian破坏了给他的文档，导致他错过了与客户约定的最后期限，这令Alex的老板Bob非常不安。老板告诉Willian，Alex告诉他备份里只有空的和受损的文件，而他本来是依靠这些文档来工作的。这时，Willian告诉老板，他早料到自己有可能会因为文档被

破坏而遭到指责，所以悄悄地写了一个Python代码，能够对所有Alex创建的TAR文档进行检测，同时记录文件备份前后的文件属性。检测表明，Alex从来没有创建一个演示文档，并且几个月来，他创建的目录下内容一直是空的。

当Alex面对这些信息时，很快就收回之前的言论，并寻找转移注意力的新焦点。不幸的是，这是Alex最后的救命稻草，几个月后，他再也没有出现在工作场合。他或者离开了或者被解雇了，但这已经再不重要，我们的朋友Willian已经成功解决了“TAR文档丢失事件”。

从这个故事中得出的道理就是，当你处理备份时，可以将它们视为核武器，因为它们能够帮你避免许多从未想到的危险。

下面是一些检测TAR文件内容的方法，这些TAR文件是我们之前创建的：

```
In [1]: import tarfile  
  
In [2]: tar = tarfile.open("temp.tar","r")  
  
In [3]: tar.list()  
-rw-r--r-- ngift/wheel          2 2008-04-04 15:17:14 tmp/file00.txt  
-rw-r--r-- ngift/wheel          2 2008-04-04 15:15:39 tmp/file1.txt  
-rw-r--r-- ngift/wheel          0 2008-04-04 20:50:57 tmp/temp.tar  
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:07 tmp/dirA/file0.txt  
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:07 tmp/dirA/file00.txt  
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:07 tmp/dirA/file1.txt  
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:52 tmp/dirB/file0.txt  
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:52 tmp/dirB/file00.txt  
-rw-r--r-- ngift/wheel          2 2008-04-04 16:19:52 tmp/dirB/file1.txt  
-rw-r--r-- ngift/wheel          3 2008-04-04 16:21:50 tmp/dirB/file11.txt  
  
In [4]: tar.name  
Out[4]: '/private/tmp/temp.tar'  
  
In [5]: tar.getnames()  
Out[5]:  
['tmp/file00.txt',  
'tmp/file1.txt',  
'tmp/temp.tar',  
'tmp/dirA/file0.txt',  
'tmp/dirA/file00.txt',  
'tmp/dirA/file1.txt',  
'tmp/dirB/file0.txt',  
'tmp/dirB/file00.txt',  
'tmp/dirB/file1.txt',  
'tmp/dirB/file11.txt']  
  
In [10]: tar.members  
Out[10]: [,  
<TarInfo 'tmp/file1.txt' at 0x109ef30>,  
<TarInfo 'tmp/temp.tar' at 0x10a4310>,  
<TarInfo 'tmp/dirA/file0.txt' at 0x10a4350>]
```

```
<TarInfo 'tmp/dirA/file00.txt' at 0x10a43b0>,
<TarInfo 'tmp/dirA/file1.txt' at 0x10a4410>,
<TarInfo 'tmp/dirB/file0.txt' at 0x10a4470>,
<TarInfo 'tmp/dirB/file00.txt' at 0x10a44d0>,
<TarInfo 'tmp/dirB/file1.txt' at 0x10a4530>,
<TarInfo 'tmp/dirB/file11.txt' at 0x10a4590>]
```

这些示例演示了如何检查TAR文档中的文件名，这在数据验证脚本中是十分有效的。提取文件不是非常费劲的工作。如果想要把整个TAR文档中的内容提取到当前工作目录下，可以简单地执行下面的操作：

► In [60]: tar.extractall()

```
drwxrwxrwx 7 ngift wheel 238 Apr 4 22:59 tmp/
```

如果你有疑虑，可以增加这个操作：提取归档文件的内容，并对来自归档中的文件执行随机MD5校验，并将结果与备份之前生成的MD5校验和进行比较。这对于监测数据是否完整非常有效。

没有任何归档解决方法是完全可靠的。至少，对文档进行随机检测应当是自动进行的，每一个单独的文档应该在创建之后被重新打开并验证其有效性。

## 第7章

# SNMP

## 引言

如果你是一名系统管理员，可以说SNMP能改变你的生活。使用SNMP的好处不像编写几行Python代码来对日志文件进行解析那样立竿见影，但是当一个SNMP基础结构被建立起来之后，在其上工作会让你感到惊喜。

在这一章中，我们涉及SNMP的这样一些方面：自动发现，投票/监测，写代理，设备控制，以及最后的企业SNMP整合。当然，所有这些事情都是在Python下完成的。

如果你对SNMP不熟悉，或许需要重新温习SNMP。我们强烈建议阅读由Douglas Mauro和Kevin Schmidt（O'Reilly）编著的《Essential SNMP》，或者至少将这本书放在手边以便查阅。一本好的SNMP参考书对于真正理解SNMP，明白使用SNMP到底可以做些什么，是非常重要的。接下来的一节仅介绍了SNMP的一些基本知识，太多的细节超出了本书的范围。事实上，关于SNMP的内容非常丰富，需要完整的一本书才能把在Python中如何使用SNMP介绍清楚。

## 对SNMP的简要介绍

### SNMP概述

总体来看，SNMP是IP网络的设备管理协议。典型地，这是通过UDP端口161和162来实现的，尽管也有极小的可能性会选择使用TCP协议。作为数据中心的现代设备支持SNMP，这表示它能够管理的不仅是交换机和路由器，也包括打印服务器，UPS，存储器等设备。

使用SNMP的基本方法是发送UDP包到主机，然后等待响应。这是监测设备非常常用的做法。使用SNMP协议也可以完成其他一些工作，例如控制设备，对特定条件进行响应的写代理。

一些使用SNMP的非常典型的应用包括监测CPU负载，磁盘使用情况和内存空闲状态。你可以使用SNMP来管理和控制交换机，或者更进一步通过SNMP来重新加载配置文件。通常很少人知道，使用SNMP也可以监测软件，例如web应用程序和数据库。最后，SNMP还可以提供对RMON MIB的远程监控，这是支持基于数据流的监控，不同于常规的基于设备的SNMP监控。

既然已经谈到了MIB，接下来对其做进一步说明。SNMP仅是一个协议，并没有对数据进行约定。在被监测的设备上运行代理snmpd。snmpd具有一个保持追踪的对象列表。实际的对象列表由MIB（管理信息基础）进行控制。每一个代理至少实现一个MIB，并且是MIB-II，其在RFC1213中被定义。一种看待MIB的思路是将其视为一个能够将名称翻译为数字的文件，就象DNS一样，但是MIB要更为复杂一些。

MIB文件内是被管理对象的定义。每一个对象有三个属性：名称，类型和语法，以及编码。在这些属性当中，名称是第一个需要知道的内容。名称经常作为OID（对象标识）被引用。OID让你可以告诉代理需要操作的对象是什么。名称有两种类型：数字类型和文本类型（适于阅读）。大多数情况下你更愿意使用适于阅读的文本类型的OID名称，因为数字名称非常长而且很难记忆。最普通的OID是sysDescr。如果使用命令行工具snmpwalk来设置sysDescr OID的值，既可以将其设为文本也可以设为数字：

```
[root@rhel][H:4461][J:0]# snmpwalk -v 2c -c public localhost .1.3.6.1.2.1.1.1.0  
SNMPv2-MIB::sysDescr.0 = STRING: Linux localhost  
2.6.18-8.1.15.el5 #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686  
  
[root@rhel][H:4461][J:0]# snmpwalk -v 2c -c public localhost sysDescr  
SNMPv2-MIB::sysDescr.0 = STRING: Linux localhost  
2.6.18-8.1.15.el5 #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686
```

至此，我们给出了一堆术语和一个RFC，面对这么多术语你或许会比较困惑，正对是继续阅读还是放弃而犹豫不决。我们保证很快你就会感觉好一些，并在几分钟内写出代码。

## SNMP安装及配置

出于简化的考虑，我们仅介绍Net-SNMP，以及绑定到Net-SNMP的对应Python。仅介绍Net-SNMP并不表示其他一些基于Python的SNMP库（包括PySNMP，这是TwistedSNMP和Zenoss都使用的）不好用。在Zenoss和TwistedSNMP中，PySNMP以异步方式使用。这是非常有效的方式，也值得介绍，但在本章中没有足够的篇幅来对这两者进行介绍。

对于Net-SNMP自身，可以使用两类不同的API。方法一使用subprocess模块来封装Net-SNMP命令行工具；方法二是使用新的Python绑定。每一个方法都有各自的优势和劣势，这取决于其在什么环境中被实现。

最后，我们也介绍了Zenoss，它是开源的，是一个非常出色的Python下的企业级SNMP监测解决方案。通过Zenoss，你不必从头来编写SNMP管理解决方案，并且可以通过它的公共API来与其通信。为zenoss写一个插件、补丁或是扩展Zenoss本身都是可行的。

要使SNMP发挥作用，尤其是Net-SNMP，你必须首先对其进行安装。幸运的是，绝大多数Unix和Linux操作系统已经安装了Net-SNMP，因此如果需要监测一个设备，通常只需要修改*snmpd.conf*文件来适应具体需要，并启动守护进程。如果计划使用Python绑定的Net-SNMP进行开发（这正是本章将介绍的内容），需要编译源代码来安装Python绑定。如果计划封装Net-SNMP命令行工具，例如*snmpget*, *snmpwalk*, *snmpdf*以及其他，只要Net-SNMP已经安装，那么就不需要再做任何事情了。

可以从<http://www.oreilly.com/9780596515829>下载虚拟机，以及这本书的源代码。也可以访问[www.py4sa.com](http://www.py4sa.com)，这是这本书的合作图书网站，它有最新的关于如何运行本节中示例的信息。

我们已经配置了具有Net-SNMP和Python的虚拟机。你可以通过使用虚拟机来运行所有的示例。如果有足够的可供使用的硬件资源，还可以制作一些虚拟机的副本，并模拟本章中提及的同时与多台主机进行通信的代码。

安装Python绑定，你需要从sourceforge.net下载Net-SNMP，并且需要Net-SNMP 5.4或更高版本。绑定不是默认的，因此需要仔细地依照Python/README目录中的说明来构建。简单地说，首先需要编译该版本的Net-SNMP，然后运行Python目录中的*setup.py*脚本。我们发现在Red Hat上安装最为容易，因为有RPM资源可以利用。如果决定进行编译，你或许希望首先在Red Hat上试验一下，看其是否能够成功，然后再尝AIX，Solaris，OS X，HPUX等系统平台。如果遇到麻烦，可以先使用虚拟机来运行示例，然后再估计一下之后如何进行编译。

最后，自己编译时还需要注意一点：要确保运行*python setup.py*进行创建，并且运行*python setup.py test*进行测试。如果NetSNMP可以与Python协同工作了，你应该会找到适合的编译方法。如果在编译Python时遇到问题，有一个技巧是像下面这样手工执行*ldconfig*：

► *ldconfig -v /usr/local/lib/*

在配置方面，如果碰巧正在想要监测的客户端安装Net-SNMP，应该将Host Resources MIB与Net-SNMP一同编译。一般来说，可以像下面这样操作：

► *./configure -with-mib-modules=host*

值得注意的是，在运行配置时，它会试图运行一个自动配置脚本。如果不想要，就没必要