HZ BOOKS
华章科技

Python for Data Analysis

利用Python进行数据分析

O'REILLY®
机械工业出版社
China Machine Press

Wes McKinney 著
唐学韬 等译

利用Python进行数据分析

利用Python进行数据分析

Wes McKinney 著

唐学韬 等译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权机械工业出版社出版

机械工业出版社

图书在版编目（CIP）数据

利用Python进行数据分析 /（美）麦金尼（McKinney, W.）著；唐学韬等译.
—北京：机械工业出版社，2013.9
（O'Reilly精品图书系列）

书名原文：Python for Data Analysis
ISBN 978-7-111-43673-7

I. 利… II. ①麦… ②唐… III.①统计分析－应用软件 IV. C819
中国版本图书馆CIP数据核字（2013）第187885号

北京市版权局著作权合同登记
图字：01-2013-4246号

书　　名/　利用Python进行数据分析
书　　号/　ISBN 978-7-111-43673-7
责任编辑/　秦健
封面设计/　Karen Montgomery，张健
出版发行/　机械工业出版社
地　　址/　北京市西城区百万庄大街22号（邮政编码 100037）
印　　刷/
开　　本/　178毫米×233毫米　16开本　29印张
版　　次/　2014年1月第1版　2014年1月第1次印刷
定　　价/　89.00元（册）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：(010)88378991；88361066
购书热线：(010)68326294；88379649；68995259
投稿热线：(010)88379604
读者信箱：hzjsj@hzbook.com

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序

说句真心话，我非常感谢有机会翻译这本书，所以这可算是第一篇我自己真正想写的译者序。虽然之前也翻译过好几本书，但都没有这次的感悟这么多、这么深！这本书是我花精力和时间最多，同时也是最不满意的一本，就是因为这些感悟 —— 我始终觉得，如果再多点时间的话，我还可以翻译得更好。

本书的内容非常好，至少有一点非常好 —— 集中火力对付特定的应用领域。市面上介绍编程的书多如牛毛，但几乎没有几本书是针对特定应用场景的。这本书对新手来说绝对是福音，因为每看完一点就可以马上将自己手上的工作直接拿来当例子练手，这种立竿见影的学习效果，绝对会增强新手的学习信心。

本书内容虽好，但由于作者是编辑界牛人，平时的工作肯定不少，写书方面的精力自然就不可能太多。加之美式英语本来就很口语化，导致原书口水话非常多，有些地方的从句跟绕口令似的。我在翻译的过程中尽量排除了一些，两次校稿的过程中又删除或大幅修改了一些废话，虽然这种“口水话”还存在不少，但至少不会对阅读造成太大影响。如果实在觉得语言不通顺，请随时发邮件给我，欢迎大家的善意指导（*tonytang1999@126.com*）。

此外，在翻译的过程中发现了不少小问题，用词方面的错误几乎都是直接改的（小部分写了译者注，因为编辑要求我尽量标出一些来以便核对），而其他错误则几乎全部采用译者注的形式说明，还有一些原文有歧义或不详尽的地方也通过译者注的形式给出了简单说明。

本书共12章，除非你已经什么都会了，否则我建议全部阅读。如果没有学过Python，建议先看看本书后面的附录。本书所用到的Python编程基础知识很少，所以只看那个附录完全

足够了。但是，如果你一点儿编程基础都没有的话，可能需要再看一本有关Python入门的书才行（比如《Python编程实践》[编注1]）。

对了，还有几件事情需要说明一下：

- 每章的代码示例最好在一个IPython会话中完成，否则可能会出现一些不必要的麻烦，比如“xxx未定义”。
- 如果在Windows里面用IPython，复制代码的时候建议使用cpaste，这个不多解释了。
- 有关地图的那段代码可能需要找英文资料看才行，我在译者注中也说明了。这可能需要花不少时间和精力。
- 由于原文各种说法不统一（甚至包括术语），虽然我尽量做了统一处理，但由于精力和时间有限，无法完全修改，所以译文中的“xxx接受yyy”、“将yyy传入xxx”说的都是“xxx函数有yyy这么个参数”；“选项”、“位置参数”、“关键字参数”、“形参”、“实参”说的都是“参数”……还有不少，我也记不清了。
- “金融和经济数据”那一章翻译得非常痛苦，因为我根本不了解那个行业，原文的术语又不标准，于是我基本都是用wikipedia和bing查英文资料，看懂之后再到baidu找中文资料，并最终确定译文。因此，可能会有不准确的情况，如果您发现了，请及时通过邮件告诉我，万分感谢。

此外，我必须感谢华章公司的编辑们。非常感谢他们能够给我这样的机会，也非常感谢他们在整个过程中给予我的各种支持和理解。希望以后还能有更加愉快的合作。

本书大部分内容的翻译工作以及全书的统稿工作由我完成，参与本书翻译校对工作的还有黄惠庄、卢彦良、蒲巧惠、陈丽丽、胡元江、张杨、赵杰、吴斌、郭敏、林丹、王跃等。

由于译者水平有限，书中肯定会存在一些错误或不妥之处，因此，在阅读过程中发现有任何问题，请随时联系我们（*tonytang1999@126.com*）或机械工业出版社，我们将及时更新本书的勘误表。当然，也非常欢迎大家对本书提出宝贵的意见和建议。

唐学韬

2013年6月于广州

编注1：本书已由机械工业出版社出版，ISBN:978-7-111-36478-8。

目录

前言

针对科学计算领域的Python开源库生态系统在过去10年中得到了飞速发展。2011年底，我深深地感觉到，由于缺乏集中的学习资源，刚刚接触数据分析和统计应用的Python程序员举步维艰。针对数据分析的关键项目（尤其是NumPy、matplotlib和pandas）已经很成熟了，也就是说，写一本专门介绍它们的图书貌似不会很快过时。因此，我下定决心要开始这样的一个写作项目。我在2007年刚开始用Python进行数据分析工作时就希望能够得到这样一本书。希望你也能觉得本书有用，同时也希望你能将书中介绍的那些工具高效地运用到实际工作中去。

本书的约定

本书使用了以下排版约定：

斜体（Italic）
: 用于新术语、URL、电子邮件地址、文件名与文件扩展名。

等宽字体（`Constant width`）
: 用于表明程序清单，以及在段落中引用的程序中的元素，如变量、函数名、数据库、数据类型、环境变量、语句、关键字等。

等宽粗体（**`Constant width bold`**）
: 用于表明命令，或者需要读者逐字输入的文本内容。

等宽斜体（*`Constant width italic`*）
: 用于表示需要使用用户提供的值或者由上下文决定的值来替代的文本内容。

注意： 代表一个技巧、建议或一般性说明。

警告：代表一个警告或注意事项。

示例代码的使用

本书提供代码的目的是帮你快速完成工作。一般情况下，你可以在你的程序或文档中使用本书中的代码，而不必取得我们的许可，除非你想复制书中很大一部分代码。例如，你在编写程序时，用到了本书中的几个代码片段，这不必取得我们的许可。但若将O'Reilly图书中的代码制作成光盘并进行出售或传播，则需获得我们的许可。引用示例代码或书中内容来解答问题无需许可。将书中很大一部分的示例代码用于你个人的产品文档，这需要我们的许可。

如果你引用了本书的内容并标明版权归属声明，我们对此表示感谢，但这不是必需的。版权归属声明通常包括：标题、作者、出版社和ISBN号，例如："*Python for Data Analysis* by William Wesley McKinney (O'Reilly). Copyright 2013 William Wesley McKinney, 978-1-449-31979-3"。

如果你认为你对示例代码的使用已经超出上述范围，或者你对是否需要获得示例代码的授权还不清楚，请随时联系我们：*permissions@oreilly.com*。

联系我们

有关本书的任何建议和疑问，可以通过下列方式与我们取得联系：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

我们会在本书的网页中列出勘误表、示例和其他信息。可以通过*http://oreil.ly/Python_for_Data_Analysis*访问该页面。

要评论或询问本书的技术问题，请发送电子邮件到：

bookquestions@oreilly.com

想了解关于O'Reilly图书、课程、会议和新闻的更多信息，请访问以下网站：

http://www.oreilly.com.cn

http://www.oreilly.com

还可以通过以下网站关注我们：

我们在Facebook上的主页：*http://facebook.com/oreilly*

我们在Twitter上的主页：*http://twitter.com/oreillymedia*

我们在YouTube上的主页：*http://www.youtube.com/oreillymedia*

第1章

准备工作

本书主要内容

本书讲的是利用Python进行数据控制、处理、整理、分析等方面的具体细节和基本要点。同时，它也是利用Python进行科学计算的实用指南（专门针对数据密集型应用）。本书重点介绍了用于高效解决各种数据分析问题的Python语言和库。本书没有阐述如何利用Python实现具体的分析方法。

当书中出现“数据”时，究竟指的是什么呢？主要指的是结构化数据（structured data），这个故意含糊其辞的术语代指了所有通用格式的数据，例如：

- 多维数组（矩阵）。
- 表格型数据，其中各列可能是不同的类型（字符串、数值、日期等）。比如保存在关系型数据库中或以制表符/逗号为分隔符的文本文件中的那些数据。
- 通过关键列（对于SQL用户而言，就是主键和外键）相互联系的多个表。
- 间隔平均或不平均的时间序列。

这绝不是一个完整的列表。大部分数据集都能被转化为更加适合分析和建模的结构化形式，虽然有时这并不是很明显。如果不行的话，也可以将数据集的特征提取为某种结构化形式。例如，一组新闻文章可以被处理为一张词频表，而这张词频表就可以用于情感分析。

大部分电子表格软件（比如Microsoft Excel，它可能是世界上使用最广泛的数据分析工具了）的用户不会对此类数据感到陌生。

为什么要使用Python进行数据分析

许许多多的人（包括我自己）都很容易爱上Python这门语言。自从1991年诞生以来，Python现在已经成为最受欢迎的动态编程语言之一，其他还有Perl、Ruby等。由于拥有大量的Web框架（比如Rails（Ruby）和Django（Python）），最近几年非常流行使用Python和Ruby进行网站建设工作。这些语言常被称作脚本（scripting）语言，因为它们可以用于编写简短而粗糙的小程序（也就是脚本）。我个人并不喜欢“脚本语言”这个术语，因为它好像在说这些语言无法用于构建严谨的软件。在众多解释型语言中，Python最大的特点是拥有一个巨大而活跃的科学计算（scientific computing）社区。进入21世纪以来，在行业应用和学术研究中采用Python进行科学计算的势头越来越猛。

在数据分析和交互、探索性计算以及数据可视化等方面，Python将不可避免地接近于其他开源和商业的领域特定编程语言/工具，如R、MATLAB、SAS、Stata等。近年来，由于Python有不断改良的库（主要是pandas），使其成为数据处理任务的一大替代方案。结合其在通用编程方面的强大实力，我们完全可以只使用Python这一种语言去构建以数据为中心的应用程序。

把Python当做粘合剂

作为一个科学计算平台，Python的成功部分源于其能够轻松地集成C、C++以及Fortran代码。大部分现代计算环境都利用了一些Fortran和C库来实现线性代数、优选、积分、快速傅里叶变换以及其他诸如此类的算法。许多企业和国家实验室也利用Python来“粘合”那些已经用了30多年的遗留软件系统。

大多数软件都是由两部分代码组成的：少量需要占用大部分执行时间的代码，以及大量不经常执行的“粘合剂代码”。粘合剂代码的执行时间通常是微不足道的。开发人员的精力几乎都是花在优化计算瓶颈上面的，有时更是直接转用更低级的语言（比如C）。

最近这几年，Cython项目（*http://cython.org*）已经成为Python领域中创建编译型扩展以及对接C/C++代码的一大途径。

解决“两种语言”问题

很多组织通常都会用一种类似于领域特定的计算语言（如MATLAB和R）对新的想法进行研究、原型构建和测试，然后再将这些想法移植到某个更大的生产系统中去（可能是用Java、C#或C++编写的）。人们逐渐意识到，Python不仅适用于研究和原型构建，同时也适用于构建生产系统。我相信越来越多的企业也会这样看，因为研究人员和工程技术人员使用同一种编程工具将会给企业带来非常显著的组织效益。

为什么不选Python

虽然Python非常适合构建计算密集型科学应用程序以及几乎各种各样的通用系统，但它对于不少应用场景仍然力有不逮。

由于Python是一种解释型编程语言，因此大部分Python代码都要比用编译型语言（比如Java和C++）编写的代码运行慢得多。由于程序员的时间通常都比CPU时间值钱，因此许多人也愿意在这里做一些权衡。但是，在那些要求延迟非常小的应用程序中（例如高频交易系统），为了尽最大可能地优化性能，耗费时间使用诸如C++这样更低级、更低生产率的语言进行编程也是值得的。

对于高并发、多线程的应用程序而言（尤其是拥有许多计算密集型线程的应用程序），Python并不是一种理想的编程语言。这是因为Python有一个叫做全局解释器锁（Global Interpreter Lock，GIL）的东西，这是一种防止解释器同时执行多条Python字节码指令的机制。有关“为什么会存在GIL”的技术性原因超出了本书的范围，但是就目前来看，GIL并不会在短时间内消失。虽然很多大数据处理应用程序为了能在较短的时间内完成数据集的处理工作都需要运行在计算机集群上，但是仍然有一些情况需要用单进程多线程系统来解决。

这并不是说Python不能执行真正的多线程并行代码，只不过这些代码不能在单个Python进程中执行而已。比如说，Cython项目可以集成OpenMP（一个用于并行计算的C框架）以实现并行处理循环进而大幅度提高数值算法的速度。

重要的Python库

考虑到那些还不太了解Python科学计算生态系统和库的读者，下面我先对各个库做一个简单的介绍。

NumPy

NumPy（Numerical Python的简称）是Python科学计算的基础包。本书大部分内容都基于NumPy以及构建于其上的库。它提供了以下功能（不限于此）：

- 快速高效的多维数组对象ndarray。
- 用于对数组执行元素级计算以及直接对数组执行数学运算的函数。
- 用于读写硬盘上基于数组的数据集的工具。
- 线性代数运算、傅里叶变换，以及随机数生成。
- 用于将C、C++、Fortran代码集成到Python的工具。

除了为Python提供快速的数组处理能力，NumPy在数据分析方面还有另外一个主要作用，即作为在算法之间传递数据的容器。对于数值型数据，NumPy数组在存储和处理数据时要比内置的Python数据结构高效得多。此外，由低级语言（比如C和Fortran）编写的库可以直接操作NumPy数组中的数据，无需进行任何数据复制工作。

pandas

pandas提供了使我们能够快速便捷地处理结构化数据的大量数据结构和函数。你很快就会发现，它是使Python成为强大而高效的数据分析环境的重要因素之一。本书用得最多的pandas对象是DataFrame，它是一个面向列（column-oriented）的二维表结构，且含有行标和列标：

```
>>> frame
    total_bill  tip   sex     smoker  day  timesize
1   16.99       1.01  Female  No      Sun  Dinner  2
2   10.34       1.66  Male    No      Sun  Dinner  3
3   21.01       3.5   Male    No      Sun  Dinner  3
4   23.68       3.31  Male    No      Sun  Dinner  2
5   24.59       3.61  Female  No      Sun  Dinner  4
6   25.29       4.71  Male    No      Sun  Dinner  4
7   8.77        2     Male    No      Sun  Dinner  2
8   26.88       3.12  Male    No      Sun  Dinner  4
9   15.04       1.96  Male    No      Sun  Dinner  2
10  14.78       3.23  Male    No      Sun  Dinner  2
```

pandas兼具NumPy高性能的数组计算功能以及电子表格和关系型数据库（如SQL）灵活的数据处理功能。它提供了复杂精细的索引功能，以便更为便捷地完成重塑、切片和切块、聚合以及选取数据子集等操作。pandas将是我在本书中使用的主要工具。

对于金融行业的用户，pandas提供了大量适用于金融数据的高性能时间序列功能和工具。事实上，我一开始就是想把pandas设计为一款适用于金融数据分析应用的工具。

对于使用R语言进行统计计算的用户，肯定不会对DataFrame这个名字感到陌生，因为它源自于R的data.frame对象。但是这两个对象并不相同。R的data.frame对象所提供的功能只是DataFrame对象所提供的功能的一个子集。虽然本书讲的是Python，但我偶尔还是会用R做对比，因为它毕竟是最流行的开源数据分析环境，而且很多读者都对它很熟悉。

pandas这个名字本身源自于panel data（面板数据，这是计量经济学中关于多维结构化数据集的一个术语）以及Python data analysis（Python数据分析）。

matplotlib

matplotlib是最流行的用于绘制数据图表的Python库。它最初由John D. Hunter（JDH）创建，目前由一个庞大的开发人员团队维护。它非常适合创建出版物上用的图表。它跟IPython（马上就会讲到）结合得很好，因而提供了一种非常好用的交互式数据绘图环境。绘制的图表也是交互式的，你可以利用绘图窗口中的工具栏放大图表中的某个区域或对整个图表进行平移浏览。

IPython

IPython是Python科学计算标准工具集的组成部分，它将其他所有的东西联系到了一起。它为交互式和探索式计算提供了一个强健而高效的环境。它是一个增强的Python shell，目的是提高编写、测试、调试Python代码的速度。它主要用于交互式数据处理和利用matplotlib对数据进行可视化处理。我在用Python编程时，经常会用到IPython，包括运行、调试和测试代码。

除标准的基于终端的IPython shell外，该项目还提供了：

- 一个类似于Mathematica的HTML笔记本（通过Web浏览器连接IPython，稍后将对此进行详细介绍）。
- 一个基于Qt框架的GUI控制台，其中含有绘图、多行编辑以及语法高亮显示等功能。
- 用于交互式并行和分布式计算的基础架构。

我将在一章中专门讲解IPython，详细地介绍其大部分功能。强烈建议在阅读本书的过程中使用IPython。

SciPy

SciPy是一组专门解决科学计算中各种标准问题域的包的集合，主要包括下面这些包：

- `scipy.integrate`：数值积分例程和微分方程求解器。
- `scipy.linalg`：扩展了由`numpy.linalg`提供的线性代数例程和矩阵分解功能。
- `scipy.optimize`：函数优化器（最小化器）以及根查找算法。
- `scipy.signal`：信号处理工具。
- `scipy.sparse`：稀疏矩阵和稀疏线性系统求解器。

- `scipy.special`：SPECFUN（这是一个实现了许多常用数学函数（如伽玛函数）的Fortran库）的包装器。
- `scipy.stats`：标准连续和离散概率分布（如密度函数、采样器、连续分布函数等）、各种统计检验方法，以及更好的描述统计法。
- `scipy.weave`：利用内联C++代码加速数组计算的工具。

NumPy跟SciPy的有机结合完全可以替代MATLAB的计算功能（包括其插件工具箱）。

安装和设置

由于人们用Python所做的事情不同，所以没有一个普适的Python及其插件包的安装方案。由于许多读者的Python科学计算环境都不能完全满足本书的需要，所以接下来我将详细介绍各个操作系统上的安装方法。我建议使用下列Python安装包之一：

- Enthought Python Distribution[译注1]：来自Enthought（*http://continumm.io/downloads*）的面向科学计算的Python安装包。包括EPDFree（免费的基本版，带有NumPy、SciPy、matplotlib、Chaco以及IPython）和EPD Full（完整版，含有100多个针对各种领域的科学计算包）。EPD Full对高校免费，非高校用户需要缴纳年费。
- Python(x,y)（*http://pythonxy.googlecode.com*）：Windows平台上免费的Python科学计算安装包。

我将用EPDFree来说明安装过程，当然如果有需要的话，你也可以选择其他产品。编写本书时，EPD用的是Python 2.7，今后可能会有些变动。安装完毕之后，你将可以用到下面这些包：

- Python科学计算基础库：NumPy、SciPy、matplotlib以及IPython。这些都包含在EPDFree中了。
- IPython Notebook依赖项：tornado和pyzmq。这些也都包含在EPDFree中了。
- pandas（0.8.2版或更高版本）。

在阅读本书的过程中，你可能还需要安装：statsmodels、PyTables、PyQt（PySide也行）、xlrd、lxml、basemap、pymongo以及requests等（它们被用在不同的示例中）。现在暂时还不需要安装这些库，我建议你在需要的时候再安装。例如，在OS X或Linux上

译注1：已经更名为Enthought Canopy。EPDFree对应的是Enthought Canopy Express。相比来说EPDFree自然更好用，不过为了保证阅读本书时不遇到麻烦，建议按照本书介绍法操作。（其实就算按照书上的说明操作，一样会遇到不少麻烦，我会尽量给出说明。）

安装PyQt或PyTables可能会很困难。目前最重要的事情是先用EPDFree和pandas这种最小配置运行起来再说。

关于各个Python库及其安装文件和帮助信息，请访问Python Package Index（即PyPI，*http://pypi.python.org*）。你还可以在这里找到不少新的Python库。

注意： 为了简单起见，我将不会讨论pip[译注2]和virtualenv这类比较复杂的环境管理工具。网上可以找到许多介绍这些工具的优秀教程。

警告： 有些用户可能会对诸如IronPython、Jython、PyPy之类的Python实现感兴趣。为了使用本书所介绍的那些工具，（目前）就需要使用基于C的标准Python解释器（也就是CPython）。

Windows

先从*http://www.enthought.com*下载EPDFree的安装包，它可能是一个名字类似于`epd_free-7.3-1-win-x86.msi`的MSI安装包[译注3]。运行该安装包并接受默认的安装位置`C:\Python27`。如果你之前在这里安装过Python，可能需要先将其删除（可以手工删除，也可以使用控制面板中的“添加/或删除程序”功能）。

接下来，你需要验证是否已经成功将Python添加到系统路径，并且没有跟早期安装的Python版本发生冲突。首先，打开命令提示符（打开“开始”菜单，启动“命令提示符”应用程序，即`cmd.exe`）。输入`python`尝试启动Python解释器。你应该可以看到与已安装的EPDFree版本相匹配的一段消息：

```
C:\Users\Wes>python
Python 2.7.3 |EPD_free 7.3-1 (32-bit)| (default, Apr 12 2012, 14:30:37) on win32
Type "credits", "demo" or "enthought" for more information.
>>>
```

如果你看到的是其他版本的EPD信息或根本什么也看不到，那就需要清理Windows环境变量。在Windows 7上，可以在程序搜索框中输入“environment variables”，然后编辑你的账户下的环境变量。在Windows XP上，需要进入“控制面板”→“系统”→“高级”→“ 环境变量”。在弹出窗口中找到Path变量。它需要含有下面这两个以分号隔开的目录路径：

译注2：虽然安装过程不大轻松，但还是建议后面装一下，因为它可以使你在安装那些库的时候更轻松愉快。

译注3：由于软件版本更新较快，所以建议到网上找一个一模一样的安装包，不然有些例子的结果可能会跟书上介绍的不一样。

```
C:\Python27;C:\Python27\Scripts
```

如果你之前安装了其他版本的Python，那就需要删除系统和用户Path变量中与之相关的一切路径。修改路径之后，需要重启命令提示符才能使修改生效。

能够在命令提示符中成功启动Python之后，就该安装pandas了。最简单的办法就是直接到*http://pypi.python.org/pypi/pandas*下载合适的二进制安装包。对于EPDFree，应该选择*pandas-0.9.0.win32-py2.7.exe*。将其安装好之后，接下来启动IPython来验证一下是不是万事俱备了：引入pandas，然后绘制一个简单的matplotlib图形。

```
C:\Users\Wes>ipython --pylab
Python 2.7.3 |EPD_free 7.3-1 (32-bit)|
Type "copyright", "credits" or "license" for more information.

IPython 0.12.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg]. For more
information, type 'help(pylab)'.

In [1]: import pandas

In [2]: plot(arange(10))
```

如果成功，就不会出现错误信息，而且会弹出一个绘图窗口。还可以输入下列指令[译注4]来检查IPython HTML notebook是否安装成功：

```
$ ipython notebook --pylab=inline
```

警告： 如果你是在Windows上使用IPython notebook应用程序而且通常使用的是Internet Explorer的话，那你可能需要改用Mozilla Firefox或Google Chrome了[译注5]。

Windows上的EPDFree只有32位版本。如果需要使用64位版本，最简单的办法就是直接使用EPD Full[译注6]。如果你不想购买EPD订阅且愿意自己动手一步步安装，可以试

译注4：如果只用过Windows，要注意前面的“$”，这是Linux或UNIX或Mac的默认命令提示符。本书应该就是用Mac测试代码的，所以这样的提示符不少，后面代码中还有很多，请读者注意。

译注5：为什么？难道作者以为全世界人民都还在用IE6不成？译者使用IE9/IE10均无压力完成。

译注6：还是建议使用32位版本的，最主要的原因仍然是“跟原书一致，以免抓狂”。

试由加州大学欧文分校的Christoph Gohlke提供的非官方安装包（*http://www.lfd.uci.edu/~gohlke/pythonlibs/*），它既有32位版也有64位版，且包含本书所需的所有库。

苹果OS X

在OS X上，首先需要安装Xcode，它含有苹果的软件开发工具套件。我们所需的部分是gcc C和C++编译器。Xcode安装包可以在随计算机发布的OS X安装光盘中找到，也可以直接从苹果公司的网站上下载。

装好Xcode之后，到“Applications→Utilities”去启动终端（Terminal.app）。输入gcc并按回车键。你将会看到如下信息：

```
$ gcc
i686-apple-darwin10-gcc-4.2.1: no input files
```

现在就该安装EPDFree了。下载一个名为epd_free-7.3-1-macosx-i386.dmg的磁盘镜像文件。双击该.dmg文件以将其挂载到系统，然后双击其中的.mpkg文件来运行安装程序。

安装文件启动之后，会自动将EPDFree可执行文件的路径添加到你的.bash_profile文件中。该文件位于/Users/your_uname/.bash_profile：

```
# Setting PATH for EPD_free-7.3-1
PATH="/Library/Frameworks/Python.framework/Versions/Current/bin:${PATH}"
export PATH
```

如果在后续步骤中遇到任何问题，首先应该检查一下你的.bash_profile，看看是否需要将上面那个目录添加进去。

现在就该安装pandas了。在终端中执行下面这条命令：

```
$ sudo easy_install pandas
Searching for pandas
Reading http://pypi.python.org/simple/pandas/
Reading http://pandas.pydata.org
Reading http://pandas.sourceforge.net
Best match: pandas 0.9.0
Downloading http://pypi.python.org/packages/source/p/pandas/pandas-0.9.0.zip
Processing pandas-0.9.0.zip
Writing /tmp/easy_install-H5mIX6/pandas-0.9.0/setup.cfg
Running pandas-0.9.0/setup.py -q bdist_egg --dist-dir /tmp/easy_install-H5mIX6/
pandas-0.9.0/egg-dist-tmp-RhLGOz
Adding pandas 0.9.0 to easy-install.pth file

Installed /Library/Frameworks/Python.framework/Versions/7.3/lib/python2.7/
site-packages/pandas-0.9.0-py2.7-macosx-10.5-i386.egg
Processing dependencies for pandas
```

```
Finished processing dependencies for pandas
```

为了验证是否一切正常，我们以Pylab模式启动IPython，然后尝试加载pandas并绘制一张图片：

```
$ ipython --pylab
22:29 ~/VirtualBox VMs/WindowsXP $ ipython
Python 2.7.3 |EPD_free 7.3-1 (32-bit)| (default, Apr 12 2012, 11:28:34)
Type "copyright", "credits" or "license" for more information.

IPython 0.12.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg].
For more information, type 'help(pylab)'.

In [1]: import pandas

In [2]: plot(arange(10))
```

如果成功，将会弹出一个绘图窗口，其中画的是一条直线。

GNU/Linux

注意：有些（但不是全部）Linux产品自带的Python包版本较新，且可以通过内置的包管理工具（如apt）进行安装。我将详细讲解EPDFree的安装步骤，因为它在不同的Linux发行版之间是差不多的。

对于不同的Linux产品，具体的安装过程会有一些不同，我这里将以基于Debian的GNU/Linux系统（如Ubuntu和Mint）为例来进行讲解。除EPDFree之外，其他的安装过程跟OS X差不多。其安装包是一个只能在终端中执行的shell脚本。根据系统是32位还是64位，需要相应地安装`x86`版（32位）或`x86_64`版（64位）。然后你将会得到一个名为`epd_free-7.3-1-rh5-x86_64.sh`的文件。通过bash执行该脚本即可开始安装：

```
$ bash epd_free-7.3-1-rh5-x86_64.sh
```

在接受了许可协议之后，你需要选择EPDFree文件的存放位置。我建议将这些文件安装在你的home目录中，比如`/home/wesm/epd`（将`wesm`替换为你的用户名即可）。

安装完毕之后，你需要将EPDFree的`bin`目录添加到`$PATH`变量中去。如果你用的是bash shell（比如Ubuntu默认用的就是这个），则在你的`.bashrc`中加上下面这句路径添加指令：

```
export PATH=/home/wesm/epd/bin:$PATH
```

很明显，需要将/home/wesm/epd/替换为你所使用的安装目录。做完这些事情之后，你可以启动一个新的终端进程，也可以通过source ~/.bashrc重启你的.bashrc。

接下来还需要用到一个C编译器（比如gcc）。许多Linux产品都含有gcc，但有些则没有。在Debian系统中，可以执行下面这条指令来安装gcc：

```
sudo apt-get install gcc
```

如果在命令行中输入gcc，就可以看到：

```
$ gcc
gcc: no input files
```

现在可以安装pandas了：

```
$ easy_install pandas
```

如果你是以root权限来安装EPDFree的，就需要在命令中加上sudo并输入sudo或root密码。使用跟OS X相同的检测方式即可验证是否一切正常。

Python 2和Python 3

Python社区正在慢慢地从Python 2系列解释器过渡到Python 3系列。在Python 3.0问世以前，所有的Python代码都是向后兼容的。为了让Python语言更加先进，Python社区认为作出一些向后不兼容的修改是必要的。

本书是基于Python 2.7编写的，这是因为大部分Python科学计算社区还没有转向Python 3。好消息是，如果你碰巧正在使用Python 3.2，在学习本书的过程中一般也不会遇到什么麻烦。

集成开发环境（IDE）

当有人问我“你的标准开发环境是怎样的”时，我几乎总是回答“IPython外加一个文本编辑器”。我通常都在IPython中编写和调试程序，而且它可以交互式地处理数据，并通过可视化的方式验证某个数据操作的结果是否正确。诸如pandas和NumPy这样的库也可以轻松便捷地在这个shell中使用。

但是相对于文本编辑器，总有人会更喜欢IDE。因为它们提供了许多不错的“代码智能化”功能，比如自动完成以及快速获取函数和类的文档等。你可以试试下面这些：

- Eclipse + PyDev插件

- Python Tools for Visual Studio（针对Windows用户）
- PyCharm
- Spyder
- Komodo IDE

社区和研讨会

除搜索引擎之外，Python科学计算邮件列表也是很不错的资源，其上的问题几乎都会有人回答。可以看看下面这些：

- pydata：这是一个Google Group邮件列表，其中的问题都是Python数据分析和pandas方面的。
- pystatsmodels：针对与statsmodels和pandas相关的问题。
- numpy-discussion：针对与NumPy相关的问题。
- scipy-user：针对与SciPy和Python科学计算相关的问题。

我没有给出具体的URL，因为它们经常在变。通过搜索引擎即可轻松地找到它们。

全世界每年都会召开许多针对Python程序员的研讨会。PyCon和EuroPython分别是美国和欧洲最主要的Python研讨会。在阅读本书之后，如果你越来越深入地用Python进行数据分析，就可以在SciPy和EuroSciPy这两个面向科学计算的Python研讨会上找到许多“臭味相投的同道中人”。

使用本书

如果之前从未使用过Python，那你可能需要先看看本书最后的附录部分，那是一个讲解Python的语法、语言特性以及内置数据结构（如元组、列表和字典等）的简单教程。这部分内容可以看做本书其他内容的预备知识。

本书首先讲解的是IPython环境，然后简单地介绍了NumPy的关键特性，NumPy其他的高级功能则在本书最后一章讲解。然后我介绍了pandas。本书其余部分则介绍了综合运用pandas、NumPy和matplotlib（用于可视化）进行数据分析的相关知识。我尽量以增量的形式组织各种材料，但偶尔还是会出现一些跨章节的知识点。

各章的数据文件及相关材料存放在GitHub上[译注7]：

译注7：拿到书就马上去下载，一来是防止链接不可用，二来是数据有点大，宽带较小的话……

```
http://github.com/pydata/pydata-book
```

我强烈建议你下载这些数据，然后用各章所介绍的工具重写示例代码。我非常欢迎大家为本书的git库提供文稿、脚本、IPython笔记本以及其他各种有用的资源。

代码示例

本书大部分代码示例的输入形式和输出结果都会按照其在IPython shell中执行时的样子进行排版。

```
In [5]: code
Out[5]: output
```

有时，为了简洁明了，多个代码示例将会并排显示。这些代码示例应该从左到右进行阅读，且应该分别执行。

```
In [5]: code          In [6]: code2
Out[5]: output        Out[6]: output2
```

示例数据

各章的示例数据都存放在GitHub上：*http://github.com/pydata/pydata-book*。下载这些数据的方法有二：使用git版本控制命令行程序；直接从网站上下载该GitHub库的zip文件。

为了让所有示例都能重现，我尽量使其包含所有必需的东西，但仍然可能会有一些错误或遗漏。如果出现这种情况的话，请给我发邮件：`wesmckinn@gmail.com`。

引入惯例

Python社区已经广泛接受了一些常用模块的命名惯例：

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

也就是说，当你看到np.arange时，就应该想到它引用的是NumPy中的arange函数。这样做的原因是：在Python软件开发过程中，不建议直接引入类似NumPy这种大型库的全部内容（`from numpy import *`）。

行话

由于你可能不太熟悉书中使用的一些有关编程和数据科学方面的常用术语，所以我在这里先给出其简单定义：

数据规整（Munge/Munging/Wrangling）[译注8]

指的是将非结构化和（或）散乱数据处理为结构化或整洁形式的整个过程。这几个词已经悄悄成为当今数据黑客们的行话了。Munge这个词跟Lunge押韵。

伪码（Pseudocode）

算法或过程的“代码式”描述，而这些代码本身并不是实际有效的源代码。

语法糖（Syntactic sugar）

这是一种编程语法，它并不会带来新的特性，但却能使代码更易读、更易写。

致谢

如果没有那一大帮子人的帮助，我想我是写不出这本书的。

先说说O’Reilly的工作人员，我非常感谢我的编辑Meghan Blanchette和Julie Steele，他们在整个写作过程中给予了我大量的指导。Mike Loukides在本书的提案阶段也给予了很大的帮助。

许多人向我提供了大量的技术评论。具体点说，Martin Blais和Hugh Brown帮助我改进了本书的示例、简洁性以及内容组织形式。James Long、Drew Conway、Fernando Pérez、Brian Granger、Thomas Kluyver、Adam Klein、Josh Klein、Chang She以及Stéfan van der Walt都审阅了一章或几章，从多个角度给出了一些重要的反馈。

我从身边和数据社区中的好友那里得到了关于本书示例和数据集的大量灵感：Mike Dewar、Jeff Hammerbacher、James Johndrow、Kristian Lum、Adam Klein、Hilary Mason、Chang She以及Ashley Williams。

当然我还要感谢开源科学计算Python社区的许多大佬们，是他们建立了我开发工作的基础，在我编写本书时也给予了不少的鼓励：IPython核心团队（Fernando Pérez、Brian Granger、Min Ragan-Kelly、Thomas Kluyver等）、John Hunter、Skipper Seabold、Travis Oliphant、Peter Wang、Eric Jones、Robert Kern、Josef Perktold、Francesc Alted、Chris Fonnesbeck，还有很多人就不一一列举了。另外还有许多人在整个过程中也给予了大量的支持、建议和鼓励：Drew Conway、Sean Taylor、Giuseppe Paleologo、Jared Lander、David Epstein、John Krowas、Joshua Bloom、Den Pilsworth、John Myles-White，还有许多我都已经不记得了。

还要感谢我整个成长岁月中的一些人。首先，我原来在AQR公司的同事们，他们在我

译注8：本来想不翻译的，但是原文中这几个到处混用，搞得我强迫症爆发，直接全翻译成“数据规整”。

从事pandas项目时给予了不少的支持：Alex Reyfman、Michael Wong、Tim Sargen、Oktay Kurbanov、Matthew Tschantz、Roni Israelov、Michael Katz、Chris Uga、Prasad Ramanan、Ted Square，以及Hoon Kim。然后是我的导师Haynes Miller（麻省理工学院）和Mike West（杜克大学）。

私人方面，Casey Dinkin在我写书期间给予了大量的关心和照顾，并忍受了我一切的情绪波动，因为我在过了预定时间之后才东拼西凑出了最终的手稿。然后是我的父母Bill和Kim，从我很小时他们就教育我要有理想，而且绝不退而求其次。

第2章

引言

本书将要向你介绍的是用于高效处理数据的Python工具。虽然读者各自工作的最终目的千差万别，但基本都需要完成以下几个大类的任务：

与外界进行交互
: 读写各种各样的文件格式和数据库。

准备
: 对数据进行清理、修整、整合、规范化、重塑、切片切块、变形等处理以便进行分析。

转换
: 对数据集做一些数学和统计运算以产生新的数据集。比如说，根据分组变量对一个大表进行聚合。

建模和计算
: 将数据跟统计模型、机器学习算法或其他计算工具联系起来。

展示
: 创建交互式的或静态的图片或文字摘要。

我将在本章中给出一些范例数据集，并讲解我们能对其做些什么。这些例子仅仅是为了提起你的兴趣，因此只会在一个比较高的层次进行讲解。即使你从来没用过这些东西也没关系，本书后续的章节将会对此进行非常详细的讲解。在这些代码示例中，你可以看到诸如`In [15]:`之类的输入输出提示符，它们来自IPython shell。

来自bit.ly的1.usa.gov数据

2011年，URL缩短服务bit.ly跟美国政府网站usa.gov合作，提供了一份从生成.gov或.mil短链接的用户那里收集来的匿名数据[译注1]。直到编写本书时为止，除实时数据[译注2]之外，还可以下载文本文件形式的每小时快照[注1]。

以每小时快照为例，文件中各行的格式为JSON（即JavaScript Object Notation，这是一种常用的Web数据格式）。例如，如果我们只读取某个文件中的第一行，那么你所看到的结果应该是下面这样：

```
In [15]: path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'

In [16]: open(path).readline()
Out[16]: '{ "a": "Mozilla\\/5.0 (Windows NT 6.1; WOW64) AppleWebKit\\/535.11 (KHTML,
like Gecko) Chrome\\/17.0.963.78 Safari\\/535.11", "c": "US", "nk": 1, "tz":
"America\\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf", "l": "orofrog",
"al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r": "http:\\/\\/www.facebook.com\\/
l\\/7AQEFzjSi\\/1.usa.gov\\/wfLQtf", "u": "http:\\/\\/www.ncbi.nlm.nih.gov\\/
pubmed\\/22415991", "t": 1331923247, "hc":1331822918, "cy": "Danvers", "ll": [
42.576698, -70.954903 ] }\n'
```

Python有许多内置或第三方模块可以将JSON字符串转换成Python字典对象。这里，我将使用json模块及其loads函数逐行加载已经下载好的数据文件：

```
import json
path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'
records = [json.loads(line) for line in open(path)]
```

你可能之前没用过Python，解释一下上面最后那行表达式，它叫做列表推导式（list comprehension），这是一种在一组字符串（或一组别的对象）上执行一条相同操作（如`json.loads`）的简洁方式。在一个打开的文件句柄上进行迭代即可获得一个由行组成的序列。现在，`records`对象就成为一组Python字典了：

```
In [18]: records[0]
Out[18]:
{u'a': u'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko)
Chrome/17.0.963.78 Safari/535.11', u'al': u'en-US,en;q=0.8',
u'c': u'US',
```

译注1：由于可以通过短链接伪造.gov后缀的URL，导致用户访问恶意域名，所以美国政府开始着手处理这种事情了。

译注2：以Feed形式提供。

注1：网址：*http://www.usa.gov/About/developer-resources/1usagov.shtml*。

```
u'cy': u'Danvers',
u'g': u'A6qOVH',
u'gr': u'MA',
u'h': u'wfLQtf',
u'hc': 1331822918,
u'hh': u'1.usa.gov',
u'l': u'orofrog',
u'll': [42.576698, -70.954903],
u'nk': 1,
u'r': u'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
u't': 1331923247,
u'tz': u'America/New_York',
u'u': u'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

注意，Python的索引是从0开始的，不像其他某些语言是从1开始的（如R）。现在，只要以字符串的形式给出想要访问的键就可以得到当前记录中相应的值了：

```
In [19]: records[0]['tz']
Out[19]: u'America/New_York'
```

单引号前面的u表示*unicode*（一种标准的字符串编码格式）。注意，IPython在这里给出的是时区的字符串对象形式，而不是其打印形式：

```
In [20]: print records[0]['tz']
America/New_York
```

用纯Python代码对时区进行计数

假设我们想要知道该数据集中最常出现的是哪个时区（即tz字段），得到答案的办法有很多。首先，我们用列表推导式取出一组时区：

```
In [25]: time_zones = [rec['tz'] for rec in records]
---------------------------------------------------------------------------
KeyError     Traceback (most recent call last)
/home/wesm/book_scripts/whetting/<ipython> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]

KeyError: 'tz'
```

晕！原来并不是所有记录都有时区字段。这个好办，只需在列表推导式末尾加上一个`if 'tz' in rec`判断即可：

```
In [26]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [27]: time_zones[:10]
Out[27]:
[u'America/New_York',
u'America/Denver',
u'America/New_York',
```

```
u'America/Sao_Paulo',
u'America/New_York',
u'America/New_York',
u'Europe/Warsaw',
u'',
u'',
u'']
```

只看前10个时区，我们发现有些是未知的（即空的）。虽然可以将它们过滤掉，但现在暂时先留着。接下来，为了对时区进行计数，这里介绍两个办法：一个较难（只使用标准Python库），另一个较简单（使用pandas）。计数的办法之一是在遍历时区的过程中将计数值保存在字典中：

```
def get_counts(sequence):
    counts = {}
    for x in sequence:
        if x in counts:
            counts[x] += 1
        else:
            counts[x] = 1
    return counts
```

如果非常了解Python标准库，那么你可能会将代码写得更简洁一些：

```
from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # 所有的值均会被初始化为0
    for x in sequence:
        counts[x] += 1
    return counts
```

我将代码写到函数中是为了获得更高的可重用性。要用它对时区进行处理，只需将time_zones传入即可：

```
In [31]: counts = get_counts(time_zones)

In [32]: counts['America/New_York']
Out[32]: 1251

In [33]: len(time_zones)
Out[33]: 3440
```

如果想要得到前10位的时区及其计数值，我们需要用到一些有关字典的处理技巧：

```
def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in count_dict.items()]
    value_key_pairs.sort()
    return value_key_pairs[-n:]
```

现在我们就可以：

```
In [35]: top_counts(counts)
Out[35]:
[(33, u'America/Sao_Paulo'),
 (35, u'Europe/Madrid'),
 (36, u'Pacific/Honolulu'),
 (37, u'Asia/Tokyo'),
 (74, u'Europe/London'),
 (191, u'America/Denver'),
 (382, u'America/Los_Angeles'),
 (400, u'America/Chicago'),
 (521, u''),
 (1251, u'America/New_York')]
```

你可以在Python标准库中找到collections.Counter类，它能使这个任务变得更简单：

```
In [49]: from collections import Counter

In [50]: counts = Counter(time_zones)

In [51]: counts.most_common(10)
Out[51]:
[(u'America/New_York', 1251),
 (u'', 521),
 (u'America/Chicago', 400),
 (u'America/Los_Angeles', 382),
 (u'America/Denver', 191),
 (u'Europe/London', 74),
 (u'Asia/Tokyo', 37),
 (u'Pacific/Honolulu', 36),
 (u'Europe/Madrid', 35),
 (u'America/Sao_Paulo', 33)]
```

用pandas对时区进行计数

*DataFrame*是pandas中最重要的数据结构，它用于将数据表示为一个表格。从一组原始记录中创建DataFrame是很简单的：

```
In [289]: from pandas import DataFrame, Series

In [290]: import pandas as pd; import numpy as np

In [291]: frame = DataFrame(records)

In [292]: frame
Out[292]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3560 entries, 0 to 3559
Data columns:
_heartbeat_    120  non-null values
a              3440  non-null values
```

```
al              3094  non-null values
c               2919  non-null values
cy              2919  non-null values
g               3440  non-null values
gr              2919  non-null values
h               3440  non-null values
hc              3440  non-null values
hh              3440  non-null values
kw              93  non-null values
l               3440  non-null values
ll              2919  non-null values
nk              3440  non-null values
r               3440  non-null values
t               3440  non-null values
tz              3440  non-null values
u               3440  non-null values
dtypes: float64(4), object(14)
```

```
In [293]: frame['tz'][:10]
Out[293]:
0     America/New_York
1       America/Denver
2     America/New_York
3    America/Sao_Paulo
4     America/New_York
5     America/New_York
6        Europe/Warsaw
7
8
9
Name: tz
```

这里frame的输出形式是摘要视图（summary view），主要用于较大的DataFrame对象。frame['tz']所返回的Series对象有一个value_counts方法，该方法可以让我们得到所需的信息：

```
In [294]: tz_counts = frame['tz'].value_counts()
```

```
In [295]: tz_counts[:10]
Out[295]:
America/New_York       1251
                        521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Europe/London            74
Asia/Tokyo               37
Pacific/Honolulu         36
Europe/Madrid            35
America/Sao_Paulo        33
```

然后，我们想利用绘图库（即matplotlib）为这段数据生成一张图片。为此，我们先给记

录中未知或缺失的时区填上一个替代值。fillna函数可以替换缺失值（NA），而未知值（空字符串）则可以通过布尔型数组索引加以替换：

```
In [296]: clean_tz = frame['tz'].fillna('Missing')

In [297]: clean_tz[clean_tz == ''] = 'Unknown'

In [298]: tz_counts = clean_tz.value_counts()

In [299]: tz_counts[:10]
Out[299]:
America/New_York        1251
Unknown                  521
America/Chicago          400
America/Los_Angeles      382
America/Denver           191
Missing                  120
Europe/London             74
Asia/Tokyo                37
Pacific/Honolulu          36
Europe/Madrid             35
```

利用counts[译注3]对象的plot方法即可得到一张水平条形图[译注4]：

```
In [301]: tz_counts[:10].plot(kind='barh', rot=0)
```

最终结果如图2-1所示。我们还可以对这种数据进行很多处理。比如说，a字段含有执行URL短缩操作的浏览器、设备、应用程序的相关信息：

```
In [302]: frame['a'][1]
Out[302]: u'GoogleMaps/RochesterNY'

In [303]: frame['a'][50]
Out[303]: u'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'

In [304]: frame['a'][51]
Out[304]: u'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P925/V10e Build/FRG83G)
AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1'
```

译注3：应该是tz_counts 。

译注4：注意，一定要以pylab模式打开，否则这条代码没效果。包括很多缩写，pylab都直接弄好了，如果不是用这种模式打开，后面很多代码一样会遇到问题，虽然不是什么大毛病，但毕竟麻烦。后面如果遇到这没定义那找不到的情况，就请注意是不是因为这个。

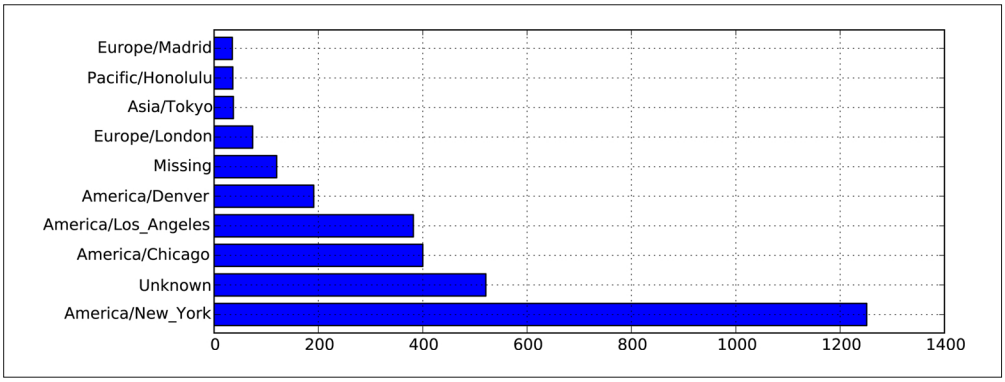

图2-1：1.usa.gov示例数据中最常出现的时区

将这些“agent”字符串[译注5]中的所有信息都解析出来是一件挺郁闷的工作。不过只要你掌握了Python内置的字符串函数和正则表达式，事情就好办了。比如说，我们可以将这种字符串的第一节（与浏览器大致对应）分离出来并得到另外一份用户行为摘要：

```
In [305]: results = Series([x.split()[0] for x in frame.a.dropna()])

In [306]: results[:5]
Out[306]:
0               Mozilla/5.0
1    GoogleMaps/RochesterNY
2               Mozilla/4.0
3               Mozilla/5.0
4               Mozilla/5.0

In [307]: results.value_counts()[:8]
Out[307]:
Mozilla/5.0                 2594
Mozilla/4.0                  601
GoogleMaps/RochesterNY       121
Opera/9.80                    34
TEST_INTERNET_AGENT           24
GoogleProducer                21
Mozilla/6.0                    5
BlackBerry8520/5.0.0.681       4
```

现在，假设你想按Windows和非Windows用户对时区统计信息进行分解。为了简单起见，我们假定只要agent字符串中含有“Windows”就认为该用户为Windows用户。由于有的agent缺失，所以首先将它们从数据中移除：

```
In [308]: cframe = frame[frame.a.notnull()]
```

其次根据a值计算出各行是否是Windows：

译注5：即浏览器的USER_AGENT信息。

```
In [309]: operating_system = np.where(cframe['a'].str.contains('Windows'),
   ...:                               'Windows', 'Not Windows')

In [310]: operating_system[:5]
Out[310]:
0        Windows
1    Not Windows
2        Windows
3    Not Windows
4        Windows
Name: a
```

接下来就可以根据时区和新得到的操作系统列表对数据进行分组了：

```
In [311]: by_tz_os = cframe.groupby(['tz', operating_system])
```

然后通过size对分组结果进行计数（类似于上面的value_counts函数），并利用unstack对计数结果进行重塑：

```
In [312]: agg_counts = by_tz_os.size().unstack().fillna(0)

In [313]: agg_counts[:10]
Out[313]:
a                               Not Windows  Windows
tz
                                        245      276
Africa/Cairo                              0        3
Africa/Casablanca                         0        1
Africa/Ceuta                              0        2
Africa/Johannesburg                       0        1
Africa/Lusaka                             0        1
America/Anchorage                         4        1
America/Argentina/Buenos_Aires            1        0
America/Argentina/Cordoba                 0        1
America/Argentina/Mendoza                 0        1
```

最后，我们来选取最常出现的时区。为了达到这个目的，我根据agg_counts中的行数构造了一个间接索引数组：

```
# 用于按升序排列
In [314]: indexer = agg_counts.sum(1).argsort()

In [315]: indexer[:10]
Out[315]:
tz
                                24
Africa/Cairo                    20
Africa/Casablanca               21
Africa/Ceuta                    92
Africa/Johannesburg             87
Africa/Lusaka                   53
America/Anchorage               54
```

```
America/Argentina/Buenos_Aires    57
America/Argentina/Cordoba         26
America/Argentina/Mendoza         55
```

然后我通过take按照这个顺序截取了最后10行：

```
In [316]: count_subset = agg_counts.take(indexer)[-10:]

In [317]: count_subset
Out[317]:
a                    Not Windows  Windows
tz
America/Sao_Paulo             13       20
Europe/Madrid                 16       19
Pacific/Honolulu              0        36
Asia/Tokyo                     2       35
Europe/London                 43       31
America/Denver               132       59
America/Los_Angeles          130      252
America/Chicago              115      285
                             245      276
America/New_York             339      912
```

这里也可以生成一张条形图。我将使用stacked=True来生成一张堆积条形图（如图2-2所示）：

```
In [319]: count_subset.plot(kind='barh', stacked=True)
```

由于在这张图中不太容易看清楚较小分组中Windows用户的相对比例，因此我们可以将各行规范化为“总计为1”并重新绘图（如图2-3所示）：

```
In [321]: normed_subset = count_subset.div(count_subset.sum(1), axis=0)

In [322]: normed_subset.plot(kind='barh', stacked=True)
```

这里所用到的所有方法都会在本书后续的章节中详细讲解。

MovieLens 1M数据集

GroupLens Research（*http://www.grouplens.org/node/73*）采集了一组从20世纪90年末到21世纪初由MovieLens用户提供的电影评分数据。这些数据中包括电影评分、电影元数据（风格类型和年代）以及关于用户的人口统计学数据（年龄、邮编、性别和职业等）。基于机器学习算法的推荐系统一般都会对此类数据感兴趣。虽然我不会在本书中详细介绍机器学习技术，但我会告诉你如何对这种数据进行切片切块以满足实际需求。

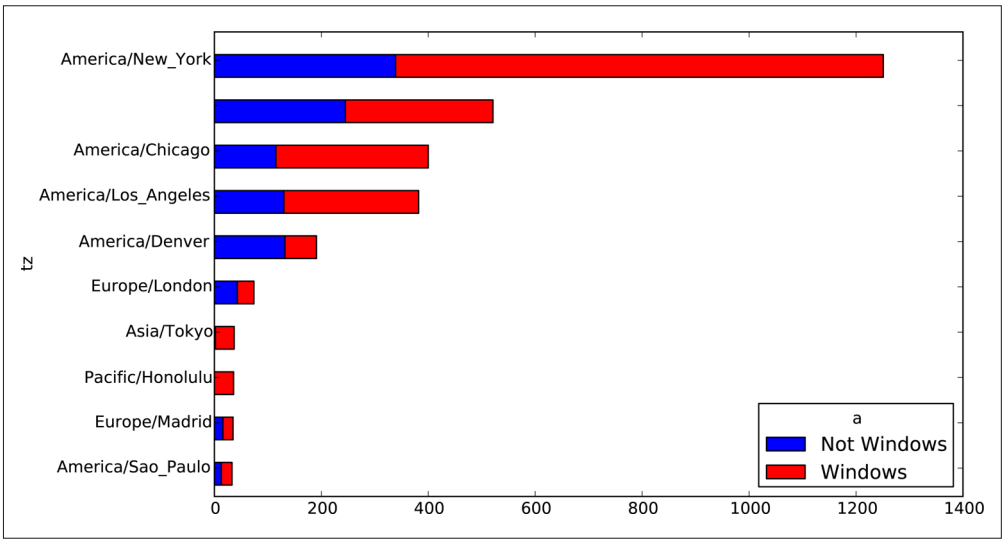

图2-2：按Windows和非Windows用户统计的最常出现的时区

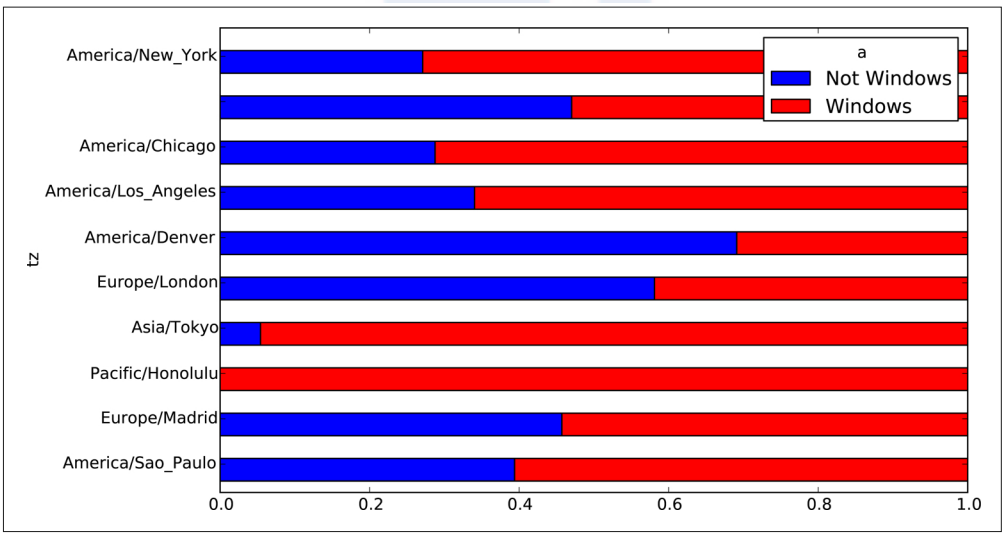

图2-3：按Windows和非Windows用户比例统计的最常出现的时区

MovieLens 1M数据集含有来自6000名用户对4000部电影的100万条评分数据。它分为三个表：评分、用户信息和电影信息。将该数据从zip文件中解压出来之后，可以通过pandas.read_table将各个表分别读到一个pandas DataFrame对象中：

```
import pandas as pd

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
```

```
users = pd.read_table('ml-1m/users.dat', sep='::', header=None, names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('ml-1m/ratings.dat', sep='::', header=None, names=rnames)

mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('ml-1m/movies.dat', sep='::', header=None, names=mnames)
```

利用Python的切片语法，通过查看每个DataFrame的前几行即可验证数据加载工作是否一切顺利：

```
In [334]: users[:5]
Out[334]:
   user_id gender  age  occupation    zip
0        1      F    1          10  48067
1        2      M   56          16  70072
2        3      M   25          15  55117
3        4      M   45           7  02460
4        5      M   25          20  55455

In [335]: ratings[:5]
Out[335]:
   user_id  movie_id  rating  timestamp
0        1      1193       5  978300760
1        1       661       3  978302109
2        1       914       3  978301968
3        1      3408       4  978300275
4        1      2355       5  978824291

In [336]: movies[:5]
Out[336]:
   movie_id                                title                        genres
0         1                    Toy Story (1995)   Animation|Children's|Comedy
1         2                      Jumanji (1995)  Adventure|Children's|Fantasy
2         3             Grumpier Old Men (1995)                Comedy|Romance
3         4            Waiting to Exhale (1995)                  Comedy|Drama
4         5  Father of the Bride Part II (1995)                        Comedy

In [337]: ratings
Out[337]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000209 entries, 0 to 1000208
Data columns:
user_id      1000209  non-null values
movie_id     1000209  non-null values
rating       1000209  non-null values
timestamp    1000209  non-null values
dtypes: int64(4)
```

注意，其中的年龄和职业是以编码形式给出的，它们的具体含义请参考该数据集的README文件。分析散布在三个表中的数据可不是一件轻松的事情。假设我们想要根据性别和年龄计算某部电影的平均得分，如果将所有数据都合并到一个表中的话问题就简单

多了。我们先用pandas的merge函数将ratings跟users合并到一起，然后再将movies也合并进去。pandas会根据列名的重叠情况推断出哪些列是合并（或连接）键：

```
In [338]: data = pd.merge(pd.merge(ratings, users), movies)

In [339]: data
Out[339]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000209 entries, 0 to 1000208
Data columns:
user_id       1000209  non-null values
movie_id      1000209  non-null values
rating        1000209  non-null values
timestamp     1000209  non-null values
gender        1000209  non-null values
age           1000209  non-null values
occupation    1000209  non-null values
zip           1000209  non-null values
title         1000209  non-null values
genres        1000209  non-null values
dtypes: int64(6), object(4)

In [340]: data.ix[0]
Out[340]:
user_id                                   1
movie_id                                  1
rating                                    5
timestamp                         978824268
gender                                    F
age                                       1
occupation                               10
zip                                   48067
title                     Toy Story (1995)
genres        Animation|Children's|Comedy
Name: 0
```

现在，只要稍微熟悉一下pandas，就能轻松地根据任意个用户或电影属性对评分数据进行聚合操作了。为了按性别计算每部电影的平均得分，我们可以使用pivot_table方法：

```
In [341]: mean_ratings = data.pivot_table('rating', rows='title',
   ....:                                   cols='gender', aggfunc='mean')

In [342]: mean_ratings[:5]
Out[342]:
gender                                F         M
title
$1,000,000 Duck (1971)         3.375000  2.761905
'Night Mother (1986)           3.388889  3.352941
'Til There Was You (1997)      2.675676  2.733333
'burbs, The (1989)             2.793478  2.962085
...And Justice for All (1979)  3.828571  3.689024
```

该操作产生了另一个DataFrame，其内容为电影平均得分，行标为电影名称，列标为性别。现在，我打算过滤掉评分数据不够250条的电影（随便选的一个数字）。为了达到这个目的，我先对title进行分组，然后利用size()得到一个含有各电影分组大小的Series对象：

```
In [343]: ratings_by_title = data.groupby('title').size()

In [344]: ratings_by_title[:10]
Out[344]:
title
$1,000,000 Duck (1971)                37
'Night Mother (1986)                  70
'Til There Was You (1997)             52
'burbs, The (1989)                   303
...And Justice for All (1979)        199
1-900 (1994)                           2
10 Things I Hate About You (1999)    700
101 Dalmatians (1961)                565
101 Dalmatians (1996)                364
12 Angry Men (1957)                  616

In [345]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [346]: active_titles
Out[346]:
Index(['burbs, The (1989), 10 Things I Hate About You (1999),
       101 Dalmatians (1961), ..., Young Sherlock Holmes (1985),
       Zero Effect (1998), eXistenZ (1999)], dtype=object)
```

该索引中含有评分数据大于250条的电影名称，然后我们就可以据此从前面的mean_ratings中选取所需的行了：

```
In [347]: mean_ratings = mean_ratings.ix[active_titles]

In [348]: mean_ratings
Out[348]:
<class 'pandas.core.frame.DataFrame'>
Index: 1216 entries, 'burbs, The (1989) to eXistenZ (1999)
Data columns:
F    1216  non-null values
M    1216  non-null values
dtypes: float64(2)
```

为了了解女性观众最喜欢的电影，我们可以对F列降序排列：

```
In [350]: top_female_ratings = mean_ratings.sort_index(by='F', ascending=False)

In [351]: top_female_ratings[:10]
Out[351]:
gender                                                   F         M
title
Close Shave, A (1995)                             4.644444  4.473795
```

```
Wrong Trousers, The (1993)                                  4.588235  4.478261
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)               4.572650  4.464589
Wallace & Gromit: The Best of Aardman Animation (1996)      4.563107  4.385075
Schindler's List (1993)                                     4.562602  4.491415
Shawshank Redemption, The (1994)                            4.539075  4.560625
Grand Day Out, A (1992)                                     4.537879  4.293255
To Kill a Mockingbird (1962)                                4.536667  4.372611
Creature Comforts (1990)                                    4.513889  4.272277
Usual Suspects, The (1995)                                  4.513317  4.518248
```

计算评分分歧

假设我们想要找出男性和女性观众分歧最大的电影。一个办法是给mean_ratings加上一个用于存放平均得分之差的列，并对其进行排序：

```
In [352]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']
```

按“diff”排序即可得到分歧最大且女性观众更喜欢的电影：

```
In [353]: sorted_by_diff = mean_ratings.sort_index(by='diff')

In [354]: sorted_by_diff[:15]
Out[354]:
gender                                     F         M      diff
title
Dirty Dancing (1987)                3.790378  2.959596 -0.830782
Jumpin' Jack Flash (1986)           3.254717  2.578358 -0.676359
Grease (1978)                       3.975265  3.367041 -0.608224
Little Women (1994)                 3.870588  3.321739 -0.548849
Steel Magnolias (1989)              3.901734  3.365957 -0.535777
Anastasia (1997)                    3.800000  3.281609 -0.518391
Rocky Horror Picture Show, The (1975)  3.673016  3.160131 -0.512885
Color Purple, The (1985)            4.158192  3.659341 -0.498851
Age of Innocence, The (1993)        3.827068  3.339506 -0.487561
Free Willy (1993)                   2.921348  2.438776 -0.482573
French Kiss (1995)                  3.535714  3.056962 -0.478752
Little Shop of Horrors, The (1960)  3.650000  3.179688 -0.470312
Guys and Dolls (1955)               4.051724  3.583333 -0.468391
Mary Poppins (1964)                 4.197740  3.730594 -0.467147
Patch Adams (1998)                  3.473282  3.008746 -0.464536
```

对排序结果反序并取出前15行，得到的则是男性观众更喜欢的电影：

```
# 对行反序，并取出前15行
In [355]: sorted_by_diff[::-1][:15]
Out[355]:
gender                                     F         M      diff
title
Good, The Bad and The Ugly, The (1966)  3.494949  4.221300  0.726351
Kentucky Fried Movie, The (1977)        2.878788  3.555147  0.676359
Dumb & Dumber (1994)                    2.697987  3.336595  0.638608
Longest Day, The (1962)                 3.411765  4.031447  0.619682
```

```
Cable Guy, The (1996)                    2.250000  2.863787  0.613787
Evil Dead II (Dead By Dawn) (1987)       3.297297  3.909283  0.611985
Hidden, The (1987)                       3.137931  3.745098  0.607167
Rocky III (1982)                         2.361702  2.943503  0.581801
Caddyshack (1980)                        3.396135  3.969737  0.573602
For a Few Dollars More (1965)            3.409091  3.953795  0.544704
Porky's (1981)                           2.296875  2.836364  0.539489
Animal House (1978)                      3.628906  4.167192  0.538286
Exorcist, The (1973)                     3.537634  4.067239  0.529605
Fright Night (1985)                      2.973684  3.500000  0.526316
Barb Wire (1996)                         1.585366  2.100386  0.515020
```

如果只是想要找出分歧最大的电影（不考虑性别因素），则可以计算得分数据的方差或标准差：

```
# 根据电影名称分组的得分数据的标准差
In [356]: rating_std_by_title = data.groupby('title')['rating'].std()

# 根据active_titles进行过滤
In [357]: rating_std_by_title = rating_std_by_title.ix[active_titles]

# 根据值对Series进行降序排列
In [358]: rating_std_by_title.order(ascending=False)[:10]
Out[358]:
title
Dumb & Dumber (1994)                     1.321333
Blair Witch Project, The (1999)          1.316368
Natural Born Killers (1994)              1.307198
Tank Girl (1995)                         1.277695
Rocky Horror Picture Show, The (1975)    1.260177
Eyes Wide Shut (1999)                    1.259624
Evita (1996)                             1.253631
Billy Madison (1995)                     1.249970
Fear and Loathing in Las Vegas (1998)    1.246408
Bicentennial Man (1999)                  1.245533
Name: rating
```

可能你已经注意到了，电影分类是以竖线（|）分隔的字符串形式给出的。如果想对电影分类进行分析的话，就需要先将其转换成更有用的形式才行。我将在本书后续章节中讲到这种转换处理，到时还会再用到这个数据。

1880—2010年间全美婴儿姓名

美国社会保障总署（SSA）提供了一份从1880年到2010年的婴儿名字频率数据。Hadley Wickham（许多流行R包的作者）经常用这份数据来演示R的数据处理功能。

```
In [4]: names.head(10)
Out[4]:
    name     sex      births   year
0   Mary     F        7065     1880
```

```
1        Anna  F      2604     1880
2        Emma  F      2003     1880
3   Elizabeth  F      1939     1880
4      Minnie  F      1746     1880
5    Margaret  F      1578     1880
6         Ida  F      1472     1880
7       Alice  F      1414     1880
8     Bertha  F      1320     1880
9       Sarah  F      1288     1880
```

你可以用这个数据集做很多事，例如：

- 计算指定名字（可以是你自己的，也可以是别人的）的年度比例。
- 计算某个名字的相对排名。
- 计算各年度最流行的名字，以及增长或减少最快的名字。
- 分析名字趋势：元音、辅音、长度、总体多样性、拼写变化、首尾字母等。
- 分析外源性趋势：圣经中的名字、名人、人口结构变化等。

利用前面介绍过的那些工具，这些分析工作都能很轻松地完成，因此我会尽量多讲一些。我建议你下载这些数据并亲自试一试。如果你在这些数据中找到了某个有趣的模式，我将非常乐意听上一听。

到编写本书时为止，美国社会保障总署将该数据库按年度制成了多个数据文件，其中给出了每个性别/名字组合的出生总数。这些文件的原始档案可以在这里获取：[译注6]

```
http://www.ssa.gov/oact/babynames/limits.html?
```

如果你在阅读本书的时候这个页面已经不见了，也可以用搜索引擎找找。下载“National data”文件names.zip，解压后的目录中含有一组文件（如yob1880.txt）。我用UNIX的head命令查看了其中一个文件的前10行（在Windows上，你可以用more命令，或直接在文本编辑器中打开）：

```
In [367]: !head -n 10 names/yob1880.txt
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

译注6：如下链接可能不可用，读者可直接在本书的github上下载。

由于这是一个非常标准的以逗号隔开的格式，所以可以用pandas.read_csv将其加载到DataFrame中：

```
In [368]: import pandas as pd

In [369]: names1880 = pd.read_csv('names/yob1880.txt', names=['name', 'sex',
'births'])

In [370]: names1880
Out[370]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2000 entries, 0 to 1999
Data columns:
name      2000  non-null values
sex       2000  non-null values
births    2000  non-null values
dtypes: int64(1), object(2)
```

这些文件中仅含有当年出现超过5次的名字。为了简单起见，我们可以用births列的sex分组小计表示该年度的births总计：

```
In [371]: names1880.groupby('sex').births.sum()
Out[371]:
sex
F      90993
M     110493
Name: births
```

由于该数据集按年度被分隔成了多个文件，所以第一件事情就是要将所有数据都组装到一个DataFrame里面，并加上一个year字段。使用pandas.concat即可达到这个目的：

```
# 2010是目前最后一个有效统计年度
years = range(1880, 2011)

pieces = []
columns = ['name', 'sex', 'births']

for year in years:
     path = 'names/yob%d.txt' % year
     frame = pd.read_csv(path, names=columns)

     frame['year'] = year
     pieces.append(frame)

# 将所有数据整合到单个DataFrame中
names = pd.concat(pieces, ignore_index=True)
```

这里需要注意几件事情。第一，concat默认是按行将多个DataFrame组合到一起的；第二，必须指定ignore_index=True，因为我们不希望保留read_csv所返回的原始行号。现在我们得到了一个非常大的DataFrame，它含有全部的名字数据。

现在names这个DataFrame对象看上去应该是这个样子：

```
In [373]: names
Out[373]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1690784 entries, 0 to 1690783
Data columns:
name      1690784  non-null values
sex       1690784  non-null values
births    1690784  non-null values
year      1690784  non-null values
dtypes: int64(2), object(2)
```

有了这些数据之后，我们就可以利用groupby或pivot_table在year和sex级别上对其进行聚合了，如图2-4所示：

```
In [374]: total_births = names.pivot_table('births', rows='year',
     ...:                                  cols='sex', aggfunc=sum)

In [375]: total_births.tail()
Out[375]:
sex         F        M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382

In [376]: total_births.plot(title='Total births by sex and year')
```

下面我们来插入一个prop列，用于存放指定名字的婴儿数相对于总出生数的比例。prop值为0.02表示每100名婴儿中有2名取了当前这个名字。因此，我们先按year和sex分组，然后再将新列加到各个分组上：

```
def add_prop(group):
    # 整数除法会向下圆整
    births = group.births.astype(float)

    group['prop'] = births / births.sum()
    return group
names = names.groupby(['year', 'sex']).apply(add_prop)
```

注意：由于births是整数，所以我们在计算分式时必须将分子或分母转换成浮点数（除非你正在使用Python 3！）。

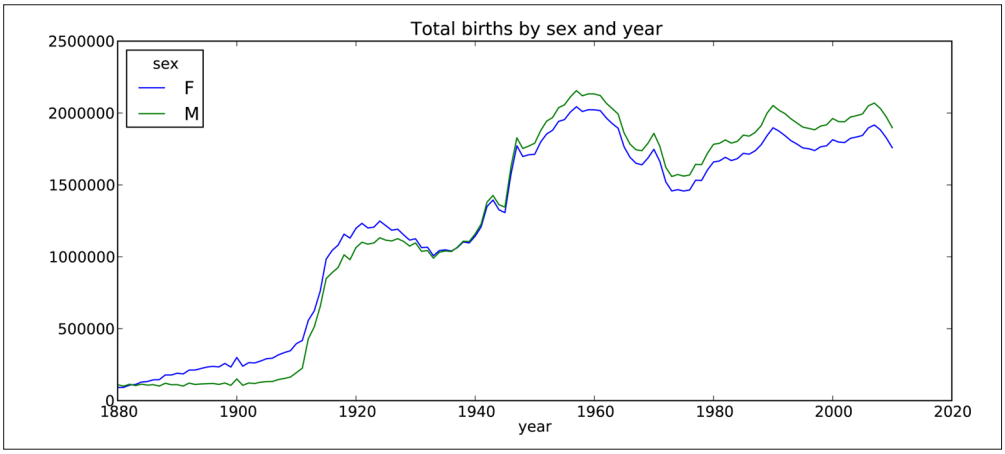

图2-4：按性别和年度统计的总出生数

现在，完整的数据集就有了下面这些列：

```
In [378]: names
Out[378]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1690784 entries, 0 to 1690783
Data columns:
name      1690784  non-null values
sex       1690784  non-null values
births    1690784  non-null values
year      1690784  non-null values
prop      1690784  non-null values
dtypes: float64(1), int64(2), object(2)
```

在执行这样的分组处理时，一般都应该做一些有效性检查，比如验证所有分组的prop的总和是否为1。由于这是一个浮点型数据，所以我们应该用np.allclose来检查这个分组总计值是否足够近似于（可能不会精确等于）1：

```
In [379]: np.allclose(names.groupby(['year', 'sex']).prop.sum(), 1)
Out[379]: True
```

这样就算完活了。为了便于实现更进一步的分析，我需要取出该数据的一个子集：每对sex/year组合的前1000个名字。这又是一个分组操作：

```
def get_top1000(group):
    return group.sort_index(by='births', ascending=False)[:1000]

grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
```

如果你喜欢DIY的话，也可以这样：

```
pieces = []
for year, group in names.groupby(['year', 'sex']):
    pieces.append(group.sort_index(by='births', scending=False)[:1000])
top1000 = pd.concat(pieces, ignore_index=True)
```

现在的结果数据集就小多了：

```
In [382]: top1000
Out[382]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 261877 entries, 0 to 261876
Data columns:
name      261877  non-null values
sex       261877  non-null values
births    261877  non-null values
year      261877  non-null values
prop      261877  non-null values
dtypes: float64(1), int64(2), object(2)
```

接下来的数据分析工作就针对这个top1000数据集了。

分析命名趋势

有了完整的数据集和刚才生成的top1000数据集，我们就可以开始分析各种命名趋势了。首先将前1000个名字分为男女两个部分：

```
In [383]: boys = top1000[top1000.sex == 'M']

In [384]: girls = top1000[top1000.sex == 'F']
```

这是两个简单的时间序列，只需稍作整理即可绘制出相应的图表（比如每年叫做John和Mary的婴儿数）。我们先生成一张按year和name统计的总出生数透视表：

```
In [385]: total_births = top1000.pivot_table('births', rows='year', cols='name',
     ...:                                     aggfunc=sum)
```

现在，我们用DataFrame的plot方法绘制几个名字的曲线图：

```
In [386]: total_births
Out[386]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6865 entries, Aaden to Zuri
dtypes: float64(6865)

In [387]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]

In [388]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
     ...:             title="Number of births per year")
```

最终结果如图2-5所示。从图中可以看出，这几个名字在美国人民的心目中已经风光不再了。但事实并非如此简单，我们在下一节中就能知道是怎么一回事了。

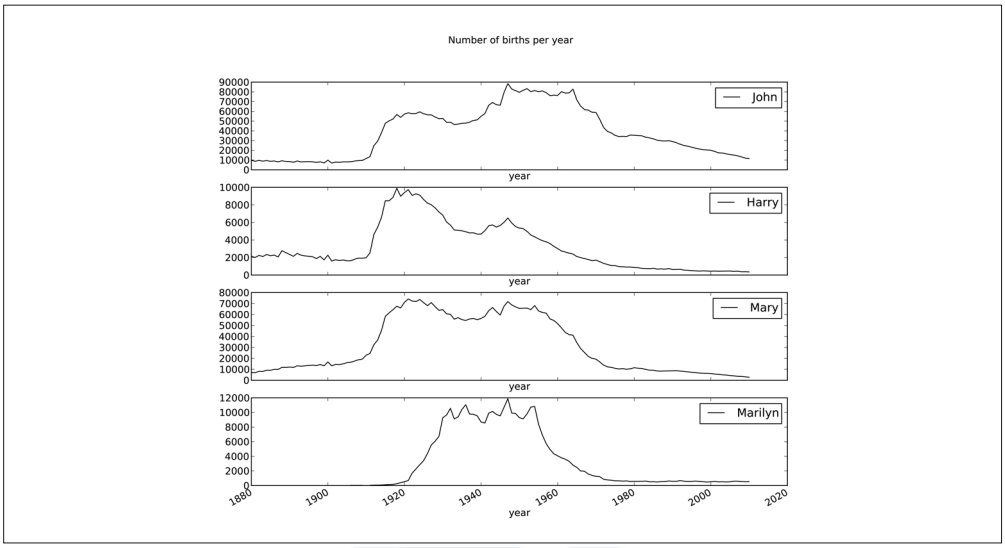

图2-5：几个男孩和女孩名字随时间变化的使用数量

评估命名多样性的增长

图2-5所反映的降低情况可能意味着父母愿意给小孩起常见的名字越来越少。这个假设可以从数据中得到验证。一个办法是计算最流行的1000个名字所占的比例，我按year和sex进行聚合并绘图：

```
In [390]: table = top1000.pivot_table('prop', rows='year',
     ...:                              cols='sex', aggfunc=sum)

In [391]: table.plot(title='Sum of table1000.prop by year and sex',
     ...:            yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020, 10))
```

结果如图2-6所示。从图中可以看出，名字的多样性确实出现了增长（前1000项的比例降低）。另一个办法是计算占总出生人数前50%的不同名字的数量，这个数字不太好计算。我们只考虑2010年男孩的名字：

```
In [392]: df = boys[boys.year == 2010]

In [393]: df
Out[393]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 260877 to 261876
Data columns:
name      1000  non-null values
```

```
sex       1000  non-null values
births    1000  non-null values
year      1000  non-null values
prop      1000  non-null values
dtypes: float64(1), int64(2), object(2)
```

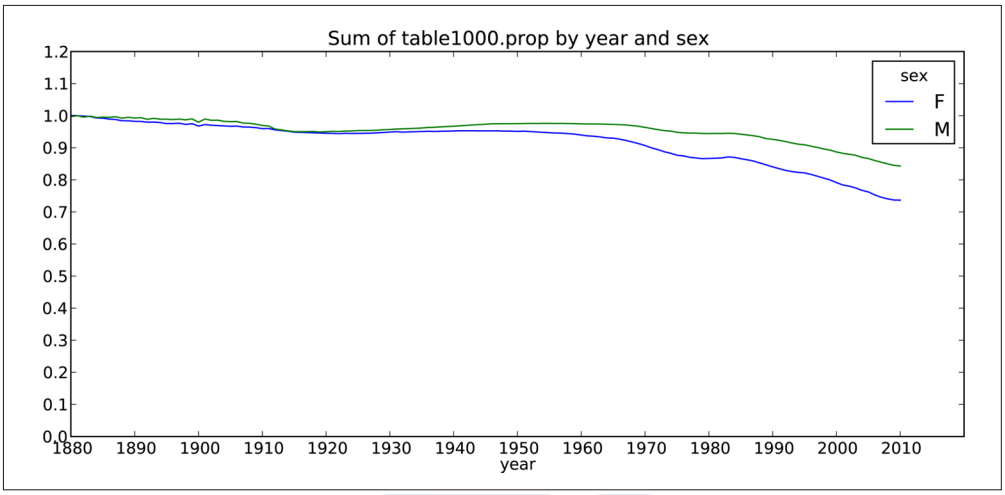

图2-6：分性别统计的前1000个名字在总出生人数中的比例

在对prop降序排列之后，我们想知道前面多少个名字的人数加起来才够50%。虽然编写一个for循环确实也能达到目的，但NumPy有一种更聪明的矢量方式。先计算prop的累计和cumsum，然后再通过searchsorted方法找出0.5应该被插入在哪个位置才能保证不破坏顺序：

```
In [394]: prop_cumsum = df.sort_index(by='prop', ascending=False).prop.cumsum()

In [395]: prop_cumsum[:10]
Out[395]:
260877  0.011523
260878  0.020934
260879  0.029959
260880  0.038930
260881  0.047817
260882  0.056579
260883  0.065155
260884  0.073414
260885  0.081528
260886  0.089621

In [396]: prop_cumsum.searchsorted(0.5)
Out[396]: 116
```

由于数组索引是从0开始的，因此我们要给这个结果加1，即最终结果为117。拿1900年的数据来做个比较，这个数字要小得多：

```
In [397]: df = boys[boys.year == 1900]
In [398]: in1900 = df.sort_index(by='prop', ascending=False).prop.cumsum()
In [399]: in1900.searchsorted(0.5) + 1
Out[399]: 25
```

现在就可以对所有year/sex组合执行这个计算了。按这两个字段进行groupby处理，然后用一个函数计算各分组的这个值：

```
def get_quantile_count(group, q=0.5):
    group = group.sort_index(by='prop', ascending=False)
    return group.prop.cumsum().searchsorted(q) + 1

diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)
diversity = diversity.unstack('sex')
```

现在，diversity这个DataFrame拥有两个时间序列（每个性别各一个，按年度索引）。通过IPython，你可以查看其内容，还可以像之前那样绘制图表（如图2-7所示）：

```
In [401]: diversity.head()
Out[401]:
sex    F   M
year
1880  38  14
1881  38  14
1882  38  15
1883  39  15
1884  39  16

In [402]: diversity.plot(title="Number of popular names in top 50%")
```

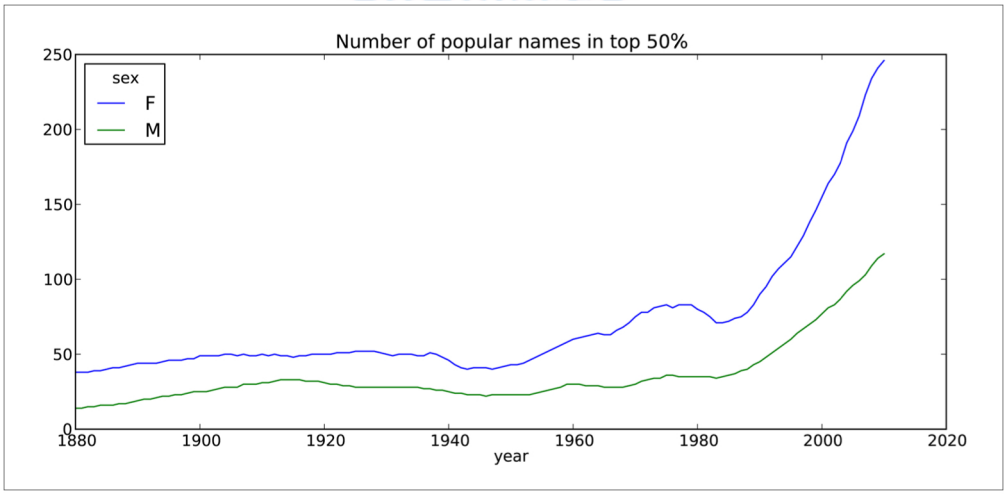

图2-7：按年度统计的密度表

从图中可以看出，女孩名字的多样性总是比男孩的高，而且还在变得越来越高。读者们可以自己分析一下具体是什么在驱动这个多样性（比如拼写形式的变化）。

“最后一个字母”的变革

2007年，一名婴儿姓名研究人员Laura Wattenberg在她自己的网站上指出（*http://www.babynamewizard.com*）：近百年来，男孩名字在最后一个字母上的分布发生了显著的变化。为了了解具体的情况，我首先将全部出生数据在年度、性别以及末字母上进行了聚合：

```
# 从name列取出最后一个字母
get_last_letter = lambda x: x[-1]
last_letters = names.name.map(get_last_letter)
last_letters.name = 'last_letter'

table = names.pivot_table('births', rows=last_letters,
                          cols=['sex', 'year'], aggfunc=sum)
```

然后，我选出具有一定代表性的三年，并输出前面几行：

```
In [404]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')

In [405]: subtable.head()
Out[405]:
sex                F                       M
year            1910    1960    2010    1910    1960    2010
last_letter
a             108376  691247  670605     977    5204   28438
b                NaN     694     450     411    3912   38859
c                  5      49     946     482   15476   23125
d               6750    3729    2607   22111  262112   44398
e             133569  435013  313833   28655  178823  129012
```

接下来我们需要按总出生数对该表进行规范化处理，以便计算出各性别各末字母占总出生人数的比例：

```
In [406]: subtable.sum()
Out[406]:
sex  year
F    1910     396416
     1960    2022062
     2010    1759010
M    1910     194198
     1960    2132588
     2010    1898382

In [407]: letter_prop = subtable / subtable.sum().astype(float)
```

有了这个字母比例数据之后，就可以生成一张各年度各性别的条形图了，如图2-8所示：

```
import matplotlib.pyplot as plt
```

```
fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0], title='Male')
letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female', legend=False)
```

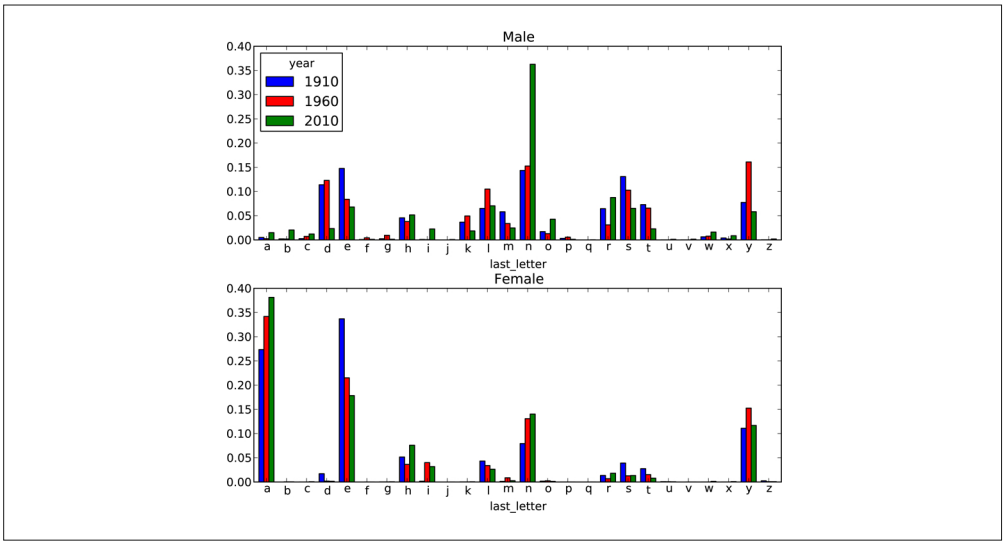

图2-8：男孩女孩名字中各个末字母的比例

从图2-8中可以看出，从20世纪60年代开始，以字母“n”结尾的男孩名字出现了显著的增长。回到之前创建的那个完整表，按年度和性别对其进行规范化处理，并在男孩名字中选取几个字母，最后进行转置以便将各个列做成一个时间序列：

```
In [410]: letter_prop = table / table.sum().astype(float)

In [411]: dny_ts = letter_prop.ix[['d', 'n', 'y'], 'M'].T

In [412]: dny_ts.head()
Out[412]:
             d         n         y
year
1880  0.083055  0.153213  0.075760
1881  0.083247  0.153214  0.077451
1882  0.085340  0.149560  0.077537
1883  0.084066  0.151646  0.079144
1884  0.086120  0.149915  0.080405
```

有了这个时间序列的DataFrame之后，就可以通过其plot方法绘制出一张趋势图了（如图2-9所示）：

```
In [414]: dny_ts.plot()
```

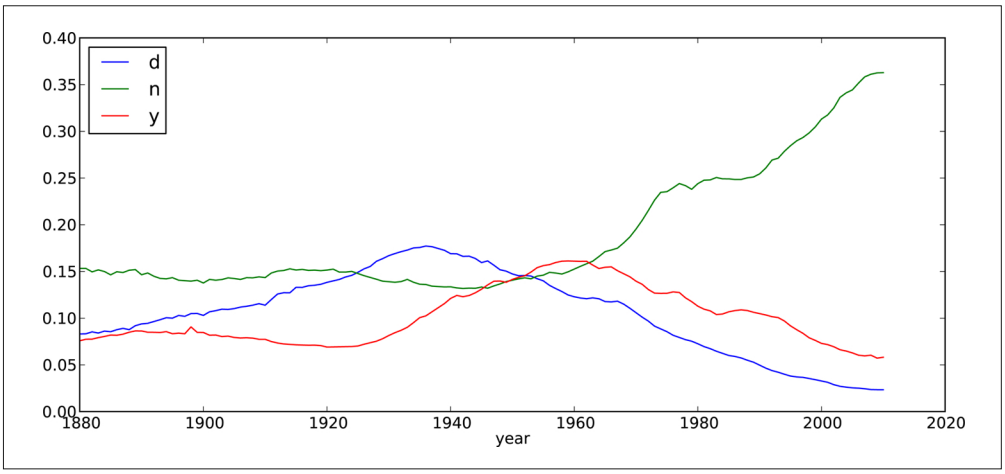

图2-9：各年出生的男孩中名字以d/n/y结尾的人数比例

变成女孩名字的男孩名字（以及相反的情况）

另一个有趣的趋势是，早年流行于男孩的名字近年来“变性了”，例如Lesley或Leslie。回到top1000数据集，找出其中以“lesl”开头的一组名字：

```
In [415]: all_names = top1000.name.unique()

In [416]: mask = np.array(['lesl' in x.lower() for x in all_names])

In [417]: lesley_like = all_names[mask]

In [418]: lesley_like
Out[418]: array([Leslie, Lesley, Leslee, Lesli, Lesly], dtype=object)
```

然后利用这个结果过滤其他的名字，并按名字分组计算出生数以查看相对频率：

```
In [419]: filtered = top1000[top1000.name.isin(lesley_like)]

In [420]: filtered.groupby('name').births.sum()
Out[420]:
name
Leslee      1082
Lesley     35022
Lesli        929
Leslie    370429
Lesly      10067
Name: births
```

接下来，我们按性别和年度进行聚合，并按年度进行规范化处理：

```
In [421]: table = filtered.pivot_table('births', rows='year',
     ...:                              cols='sex', aggfunc='sum')
```

```
In [422]: table = table.div(table.sum(1), axis=0)
In [423]: table.tail()
Out[423]:
sex   F   M
year
2006  1 NaN
2007  1 NaN
2008  1 NaN
2009  1 NaN
2010  1 NaN
```

现在，我们就可以轻松绘制一张分性别的年度曲线图了（如图2-10所示）：

```
In [425]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

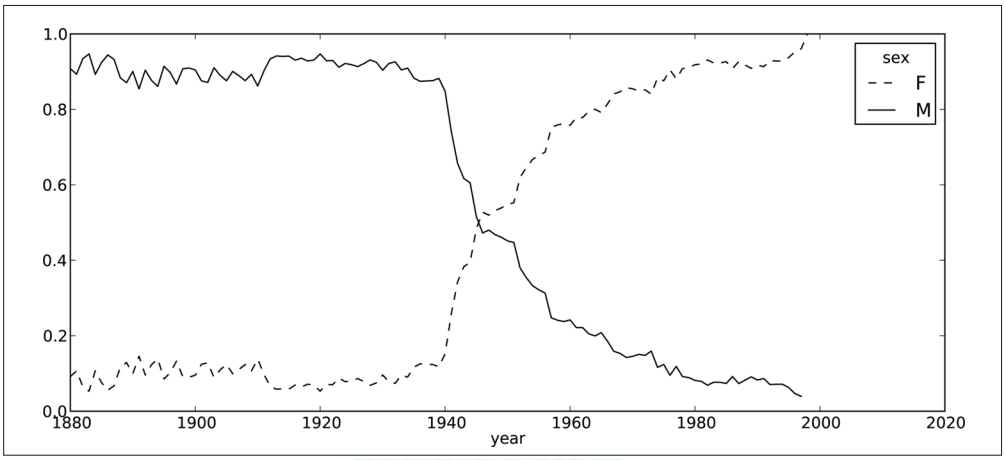

图2-10：各年度使用“Lesley型”名字的男女比例

小结及展望

本章中的这些例子都非常简单，但它们可以让你大致了解后续章节的相关内容。本书关注的焦点是工具而不是那些复杂精妙的分析方法。掌握本书所介绍的技术将使你能够立马开展自己的分析工作（假设你已经知道要做什么了）。

第3章

IPython：一种交互式计算和开发环境

为无为，事无事，味无味。大小多少。报怨以德。
图难于其易，为大于其细；
天下难事，必作于易；天下大事，必作于细。
——老子

人们经常问我，“你的Python开发环境是什么？”我的回答基本永远都是“IPython外加一个文本编辑器”。如果想要得到更加高级的图形化工具和代码自动完成功能，你也可以考虑用一款集成开发环境（IDE）来代替文本编辑器。即便如此，我仍然强烈建议你将IPython作为工作中的重要工具。有的IDE甚至本身就集成了IPython，所以说两全其美的办法还是有的。

2001年，Fernando Pérez为了得到一个更为高效的交互式Python解释器而启动了一个业余项目，于是IPython项目诞生了。在接下来的11年中，它逐渐被公认为现代科学计算中最重要的Python工具之一。IPython本身并没有提供任何的计算或数据分析功能，其设计目的是在交互式计算和软件开发这两个方面最大化地提高生产力。它鼓励一种“执行－探索”（execute explore）的工作模式，而不是许多其他编程语言那种“编辑－编译－运行”（edit-complie-run）的传统工作模式。此外，它跟操作系统shell和文件系统之间也有着非常紧密的集成。由于大部分的数据分析代码都含有探索式操作（试误法和迭代法），因此IPython（在绝大多数情况下）将有助于提高你的工作效率。

当然了，IPython项目现在已经不再只是一个加强版的交互式Python shell，它还有一个可以直接进行绘图操作的GUI控制台、一个基于Web的交互式笔记本，以及一个轻量级的快速并行计算引擎。此外，跟许多其他专为程序员设计（以及由程序员设计）的工具一样，它也是高度可定制的。我将在本章最后介绍一些这样的功能。

由于IPython的核心功能是交互，所以在没有动态控制台的情况下，本章中的某些功能很难说得清楚。如果这是你第一次学习IPython，那我建议你照着例子实际动手试试，感觉一下到底是怎么一回事。跟所有由键盘驱动的控制台环境一样，锻炼对常用命令的肌肉记忆是学习曲线中不可或缺的一部分。

注意： 在第一次阅读时，本章的许多内容都可跳过不看，因为它们对理解本书其余的内容没有影响。本章的目的是让你对IPython所提供的功能有一个全面的了解。

IPython基础

你可以通过命令行启动IPython，就像启动标准Python解释器那样，只是把命令改为ipython罢了：

```
$ ipython
Python 2.7.2 (default, May 27 2012, 21:26:12)
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- An enhanced Interactive Python.
?   -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help-> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]: a = 5

In [2]: a
Out[2]: 5
```

你可以在这里执行任何Python语句，只需将其输入然后按下回车键就行了。如果只是在IPython中输入了一个变量，那么它将会显示出该对象的一个字符串表示：[译注1]

```
In[541]: import numpy as np

In[542]: data = {i : randn() for i in range(7)}?

In [543]: data
Out[543]:
{0: 0.5580886709219381,
 1: 0.25701015249982423,
 2: 0.8876099192477288,
 3: 1.0210657329557034,
 4: -0.21799201419817044,
```

译注1：从输入输出提示符来看，作者在这两段之间做了不少事情，所以如果出现randn找不到的情况，请先执行from numpy.random import randn。

```
 5: 1.1388001234975833,
 6: -0.5209890532927175}
```

许多Python对象都被格式化为可读性更好（或者说排版更好）的形式，这跟print的普通输出形式有着显著区别。如果在标准Python解释器中打印上面那个字典对象，其可读性就没那么好了：

```
>>> from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> print data
{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135977295,
3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.2904897053651216,
6: 0.3308507317325902}
```

IPython还可以方便地执行任意代码块（通过少量优雅的复制粘贴操作）和整个Python脚本。稍后就会对该功能进行介绍。

Tab键自动完成

从表面上看，IPython就像是一个化了淡妆的交互式Python解释器。数学软件（Mathematica）用户可能会对标号式的输入输出提示符眼熟。Tab键自动完成功能是对标准Python shell的主要改进之一，大部分交互式数据分析环境都有这个功能。在shell中输入表达式时，只要按下Tab键，当前命名空间中任何与已输入的字符串相匹配的变量（对象、函数等）就会被找出来：

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>译注2
an_apple  and  an_example  any译注3
```

在这个例子中可以看到，IPython将我定义的两个变量都显示出来了，此外还显示了Python关键字and和内置函数any。当然，你也可以在任何对象后面输入一个句点以便自动完成方法和属性的输入：

```
In [3]: b = [1, 2, 3]

In [4]: b.<Tab>
b.append  b.extend  b.insert  b.remove  b.sort
b.count   b.index   b.pop     b.reverse
```

译注2：后面的<Tab>只是一个按键说明而已，不用输入。顺便说明一下，按完Tab键之后，已输入的内容会在下一行重复出现，行号也是一样的，直接接着往下输入就行了。

译注3：根据软件版本、配置以及当前上下文的不同，这里得到的结果可能会比书上的多。

还可以应用在模块上：

```
In [1]: import datetime

In [2]: datetime.<Tab>
datetime.MAXYEAR       datetime.datetime      datetime.timedelta
datetime.MINYEAR       datetime.datetime_CAPI datetime.tzinfo
datetime.date          datetime.time
```

注意：IPython默认会隐藏那些以下划线开头的方法和属性，比如魔术方法（magic method）以及内部的“私有”方法和属性，其目的是避免在屏幕上显示一堆乱七八糟的东西（也为了避免把Python新人搞晕！）。其实这些也是可以通过Tab键自动完成的，只是你得先输入一个下划线才行。如果你就是喜欢能总是看到这些方法，直接修改IPython配置文件中的相关设置就可以了。

Tab键自动完成功能不只可以用于搜索命名空间和自动完成对象或模块属性。当你输入任何看上去像是文件路径的东西时（即使是在一个Python字符串中），按下Tab键即可找出电脑文件系统中与之匹配的东西：

```
In [3]: book_scripts/<Tab>译注4
book_scripts/cprof_example.py          book_scripts/ipython_script_test.py
book_scripts/ipython_bug.py            book_scripts/prof_mod.py

In [3]: path = 'book_scripts/<Tab>
book_scripts/cprof_example.py          book_scripts/ipython_script_test.py
book_scripts/ipython_bug.py            book_scripts/prof_mod.py
```

如果再结合%run命令（参见后面内容），该功能将显著减少你敲键盘的次数。

Tab键自动完成功能还可用于函数关键字参数（包括等号（=）！）。

内省

在变量的前面或后面加上一个问号（?）就可以将有关该对象的一些通用信息显示出来：

```
In [545]: b?
Type:       list
String Form:[1, 2, 3]
Length:     3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

译注4：注意，要使用正斜杠（/），不然认不出来。此外，文件夹或文件名中间不能有空格，不然也无法正常继续操作。

这就叫做对象内省（object introspection）[译注5]。如果该对象是一个函数或实例方法，则其docstring（如果有的话）也会被显示出来：

```
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
```

然后可以利用?来显示这段docstring：

```
In [547]: add_numbers?
Type:   function
String Form:<function add_numbers at 0x5fad848>
File:   book_scripts/<ipython-input-546-5473012eeb65>
Definition: add_numbers(a, b)
Docstring:
Add two numbers together
Returns
-------
the_sum : type of arguments
```

使用??还将显示出该函数的源代码（如果可能的话）：

```
In [548]: add_numbers??
Type:   function
String Form:<function add_numbers at 0x5fad848>
File:   book_scripts/<ipython-input-546-5473012eeb65>
Definition: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Add two numbers together
    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
```

?还有一个用法，即搜索IPython命名空间，类似于标准UNIX或Windows命令行中的那种用法。一些字符再配以通配符（*）即可显示出所有与该通配符表达式相匹配的名称。例如，我们可以列出NumPy顶级命名空间中含有“load”的所有函数：

```
In [549]: np.*load*?
np.load
np.loads
```

译注5：也有译作内视、自省的，不过更多译作内省。

```
np.loadtxt
np.pkgload
```

%run命令

在IPython会话环境中，所有文件都可以通过%run命令当做Python程序来运行。假设你在ipython_script_test.py中存放了一段简单的脚本，如下所示：

```
def f(x, y, z):
    return (x + y) / z
a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

只要将文件名传给%run就可以运行了：

```
In [550]: %run ipython_script_test.py
```
译注6

脚本是在一个空的命名空间中运行的（没有任何import，也没有定义任何其他的变量），所以其行为应该跟在标准命令行环境（通过python script.py启动的）中执行时一样。此后，该文件中所定义的全部变量（还有各种import、函数和全局变量）就可以在当前IPython shell中访问了（除非发生了异常）：

```
In [551]: c
Out[551]: 7.5

In [552]: result
Out[552]: 1.4666666666666666
```

如果Python脚本需要用到命令行参数（通过sys.argv访问），可以将参数放到文件路径的后面，就像在命令行上执行那样。

注意： 如果希望脚本能够访问在交互式IPython命名空间[译注7]中定义的变量，那就应该使用%run -i而不是%run。

中断正在执行的代码

任何代码在执行时（无论是通过%run执行的脚本，还是长时间运行的命令），只要按下

译注6：注意文件的路径，这里实际上用的是默认路径。简单一点的办法就是直接写绝对路径，肯定不出错。

译注7：该命名空间的名字就是interactive。

“Ctrl-C”，就会引发一个KeyboardInterrupt。除一些非常特殊的情况之外，绝大部分Python程序都将立即停止执行。

警告： 当Python代码已经调用了某个已编译的扩展模块时，按下“Ctrl-C”将无法使程序立即停止执行。在这种情况下，要么只能等待Python解释器重新获得控制权，要么只能通过操作系统的任务管理器强制终止Python进程（比较极端的情况下才需要这么干）。

执行剪贴板中的代码

在IPython中执行代码的最简单方式是粘贴剪贴板中的代码。虽然这种做法很粗糙，但在实际工作中却很有用。比如说，在开发一个复杂或费时的应用程序时，你可能希望能一段一段地执行脚本，以便查看各个阶段所加载的数据以及产生的结果。又比如说，你在网上找了一段合用的代码，但又不想专门为其新建一个.py文件。

多数情况下，我们都可以通过“Ctrl-Shift-V”将剪贴板中的代码片段粘贴出来[译注8]。注意，这并不是万试万灵的，因为这种粘贴方式模拟的是在IPython中逐行输入代码，换行符会被处理为<return>。也就是说，如果你所粘贴的是一段缩进代码，且其中有一个空行，IPython就会认为缩进在空行那里结束了。当执行到缩进块后面那行代码时，就会引发一个IndentationError。例如下面这段代码：

```
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
```

直接粘贴是不行的：

```
In [1]: x = 5

In [2]: y = 7

In [3]: if x > 5:
   ...:         x += 1
   ...:

In [4]:     y = 8
IndentationError: unexpected indent

If you want to paste code into IPython, try the %paste and %cpaste magic functions.
```

译注8：Windows中此法行不通，需要用右键菜单中的粘贴功能，否则仅显示第一行。

正如错误提示信息所说的那样，我们应该使用%paste和%cpaste这两个魔术函数。%paste可以承载剪贴板中的一切文本[译注9]，并在shell中以整体形式执行[译注10]：

```
In [6]: %paste
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
## -- End pasted text --
```

警告： 根据你的系统平台以及Python的安装情况，%paste可能会不起作用。EPDFree（在第1章中介绍过）等打包发布的版本应该没有问题。

%cpaste跟%paste差不多[译注11]，只不过它多出了一个用于粘贴代码的特殊提示符而已：

```
In [7]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
:    x += 1
:
:    y = 8
:--
```

对于%cpaste块，在最终执行之前，你想粘贴多少代码就粘贴多少。如果想在执行那些粘贴进去的代码之前先检查一番，就可以考虑使用%cpaste。如果发现粘贴的代码有错，只需按下“Ctrl-C”即可终止%cpaste提示符。

后面我将会介绍IPython HTML Notebook，它使我们能以一种基于浏览器的notebook格式逐段对可执行代码单元进行分析。

IPython跟编辑器和IDE之间的交互

某些文本编辑器（如Emacs和vim）带有一些能将代码块直接发送到IPython shell的第三方扩展。详情请参考IPython网站或搜索引擎。

译注9：注意，这里说的是“一切”。

译注10：注意，由于是立即整体执行，所以不要复制%paste。没事干的话倒是可以试试。

译注11：建议始终用这个，虽然稍微麻烦一点，但是出错的可能性小很多。

某些IDE（如PyDev plugin for Eclipse和Python Tools for Visual Studio（微软出品））都集成了IPython终端应用程序。如果你既想用IDE又不想放弃IPython控制台，这可能是个不错的选择。

键盘快捷键

IPython提供了许多用于提示符导航（Emacs文本编辑器或UNIX bash shell的用户对此会很熟悉）和查阅历史shell命令（详见下一节）的键盘快捷键。表3-1总结了最常用的一些快捷键。图3-1说明了几个光标移动快捷键的功能。

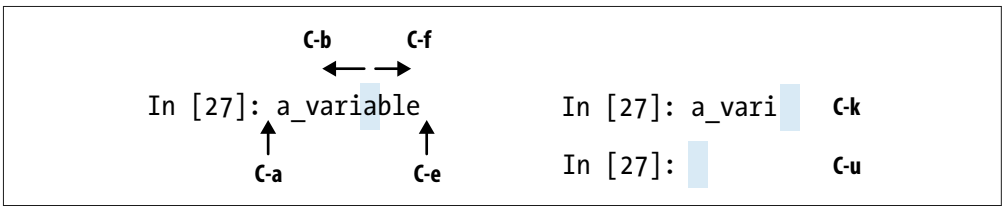

图3-1：几个IPython键盘快捷键的用法

表3-1：IPython标准键盘快捷键

命令	说明
Ctrl-P或上箭头键	后向搜索命令历史中以当前输入的文本开头的命令
Ctrl-N或下箭头键	前向搜索命令历史中以当前输入的文本开头的命令
Ctrl-R	按行读取的反向历史搜索（部分匹配）
Ctrl-Shift-v	从剪贴板粘贴文本
Ctrl-C	中止当前正在执行的代码
Ctrl-A	将光标移动到行首
Ctrl-E	将光标移动到行尾
Ctrl-K	删除从光标开始至行尾的文本
Ctrl-U	清除当前行的所有文本[译注12]
Ctrl-F	将光标向前移动一个字符
Ctrl-b	将光标向后移动一个字符
Ctrl-L	清屏

译注12：这个快捷键的功能只是跟Ctrl-K相反而已，即删除从光标开始至行首的文本，并非完全删除。

异常和跟踪

如果%run某段脚本或执行某条语句时发生了异常，IPython默认会输出整个调用栈跟踪（traceback），其中还会附上调用栈各点附近的几行代码作为上下文参考。

```
In [553]: %run ch03/ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/home/wesm/code/ipython/IPython/utils/py3compat.pyc in execfile(fname, *where)

    176             else:
    177                 filename = fname
--> 178             __builtin__.execfile(filename, *where)
book_scripts/ch03/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
---> 15 calling_things()
book_scripts/ch03/ipython_bug.py in calling_things()
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()
book_scripts/ch03/ipython_bug.py in throws_an_exception()
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():
AssertionError:
```

拥有额外的上下文代码参考是它相对于标准Python解释器的一大优势。上下文代码参考的数量可以通过%xmode魔术命令进行控制，既可以少（与标准Python解释器相同）也可以多（带有函数参数值以及其他信息）。本章稍后还会讲到如何在发生异常之后进入跟踪栈进行交互式的事后调试（post-mortem debugging）。

魔术命令

IPython有一些特殊命令（被称为魔术命令（Magic Command）），它们有的为常见任务提供便利，有的则使你能够轻松控制IPython系统的行为。魔术命令是以百分号%为前缀的命令。例如，你可以通过%timeit这个魔术命令检测任意Python语句（如矩阵乘法）的执行时间（稍后将对此进行详细讲解）：

```
In [554]: a = np.random.randn(100, 100)

In [555]: %timeit np.dot(a, a)
10000 loops, best of 3: 69.1 us per loop
```

魔术命令可以看做运行于IPython系统中的命令行程序。它们大都还有一些“命令行选项”，使用?即可查看其选项：

```
In [1]: %reset?
Resets the namespace by removing all names defined by the user.

Parameters
----------
  -f : force reset without asking for confirmation.

  -s : 'Soft' reset: Only clears your namespace, leaving history intact.
  References to objects may be kept. By default (without this option),
  we do a 'hard' reset, giving you a new session and removing all
  references to objects from the current session.

Examples
--------
In [6]: a = 1

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

In [8]: 'a' in _ip.user_ns
Out[8]: True

In [9]: %reset -f

In [1]: 'a' in _ip.user_ns
Out[1]: False
```

魔术命令默认是可以不带百分号使用的，只要没有定义与其同名的变量即可。这个技术叫做*automagic*，可以通过%automagic打开或关闭。

由于可以在IPython系统中直接访问它的文档，因此我建议你浏览一下所有这些特殊的命令（输入%quickref或%magic即可）。我将着重讲解几个重要的有助于交互式计算和Python开发的魔术命令。

表3-2：常用的IPython魔术命令

命令	说明
%quickref	显示IPython的快速参考
%magic	显示所有魔术命令的详细文档
%debug	从最新的异常跟踪的底部进入交互式调试器
%hist	打印命令的输入（可选输出）历史
%pdb	在异常发生后自动进入调试器
%paste	执行剪贴板中的Python代码

表3-2：常用的IPython魔术命令（续）

命令	说明
%cpaste	打开一个特殊提示符以便手工粘贴待执行的Python代码
%reset	删除interactive命名空间中的全部变量/名称
%page *OBJECT*	通过分页器打印输出OBJECT
%run *script.py*	在IPython中执行一个Python脚本文件
%prun *statement*	通过cProfile执行statement，并打印分析器的输出结果
%time *statement*	报告statement的执行时间
%timeit *statement*	多次执行statement以计算系综平均执行时间。对那些执行时间非常小的代码很有用
%who、%who_ls、%whos	显示interactive命名空间中定义的变量，信息级别/冗余度可变
%xdel *variable*	删除variable，并尝试清除其在IPython中的对象上的一切引用

基于Qt的富GUI控制台

IPython团队开发了一个基于Qt框架（其目的是为终端应用程序提供诸如内嵌图片、多行编辑、语法高亮之类的富文本编辑功能）的GUI控制台（见图3-2）。如果你已经安装了PyQt或PySide，使用下面这条命令来启动的话即可为其添加绘图功能：

```
ipython qtconsole --pylab=inline
```

Qt控制台可以通过标签页的形式启动多个IPython进程，这就使你能够在多个任务之间轻松切换。它也可以跟IPython HTML Notebook应用程序共享同一个进程，稍后我将专门对此进行讲解。

matplotlib集成与pylab模式

导致IPython广泛应用于科学计算领域的部分原因是它能跟matplotlib这样的库以及其他GUI工具集默契配合。即使你从未使用过matplotlib也不用担心，本书稍后会对其进行详细讲解。如果在标准Python shell中创建一个matplotlib绘图窗口，你就会郁闷地发现，GUI的事件循环会接管Python会话的控制权，直到该绘图窗口关闭为止。这自然无法实现交互式的数据分析和可视化，因此IPython对各个GUI框架进行了专门的处理以使其能够跟shell配合得天衣无缝。

通常，我们通过在启动IPython时加上`--pylab`（注意是两个短划线）标记来集成matplotlib（见图3-3）。

```
$ ipython --pylab
```

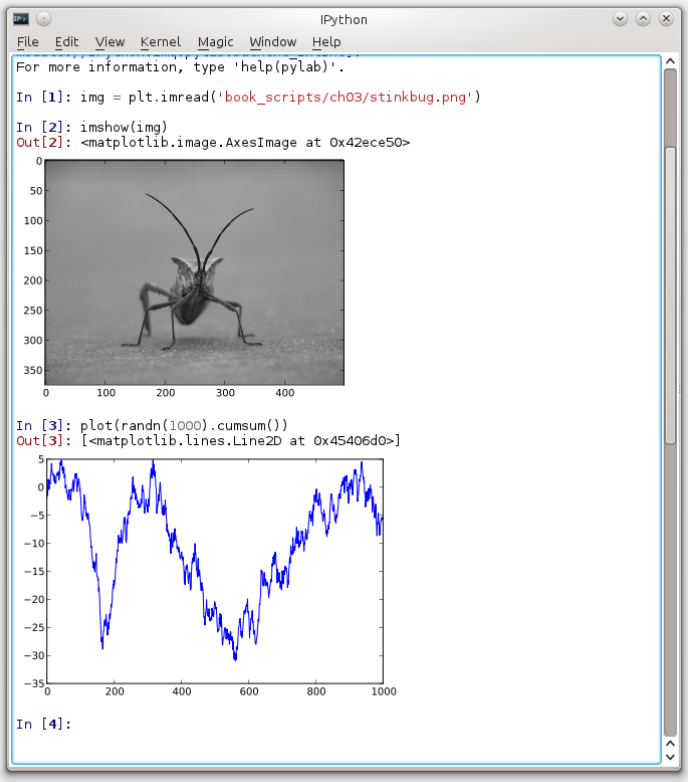

图3-2：IPython的Qt 控制台

这样会导致几个结果。第一，IPython会启用默认GUI后台集成，这样matplotlib绘图窗口的创建就没问题了。第二，NumPy和matplotlib的大部分功能会被引入到最顶层的interactive命名空间以产生一个交互式的计算环境（就像MATLAB和其他领域特定型科学计算环境那样）。也可以通过%gui对此进行手工设置（详情请执行%gui?）。

使用命令历史

IPython维护着一个位于硬盘上的小型数据库，其中含有你执行过的每条命令的文本。这样做有几个目的：

- 只需很少的按键次数即可搜索、自动完成并执行之前已经执行过的命令。
- 在会话间持久化命令历史。
- 将输入/输出历史记录到日志文件。

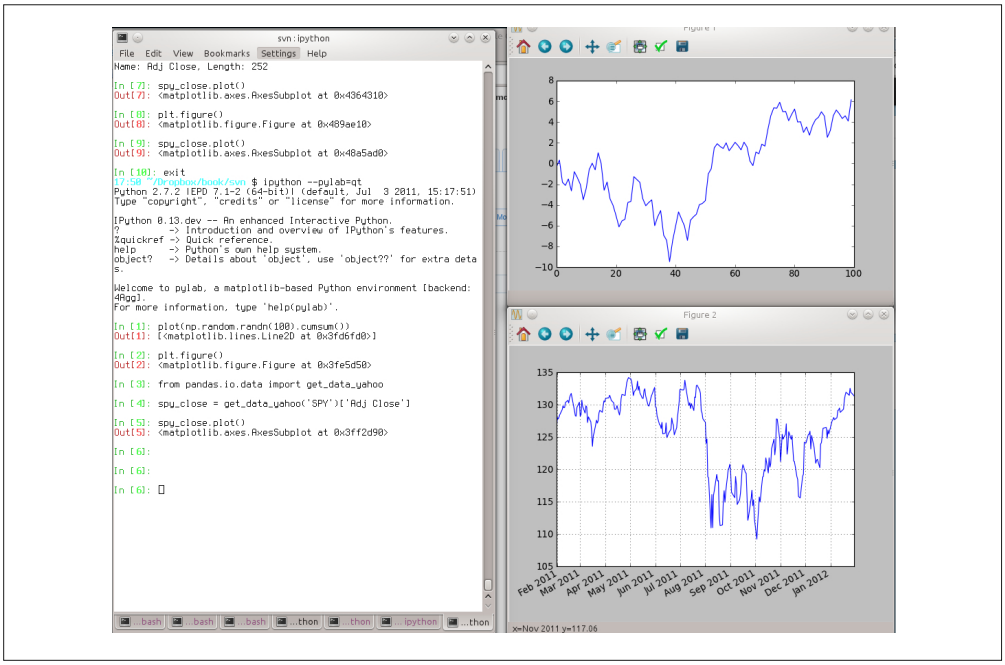

图3-3：pylab模式：IPython和matplotlib窗口

搜索并重用命令历史

对于许多人来说，能够搜索并执行前面的命令是非常有用的功能。IPython倡导的是一种迭代的、交互式的开发模式：你可能常常会发现自己总是在重复输入相同的命令（比如%run命令或其他的代码片段）。假设你已经执行了：

```
In[7]: %run first/second/third/data_script.py
```

而在查看其执行结果后（假设其已经成功执行完毕）发现计算过程不对。在找出问题原因并修改了data_script.py之后，只需输入%run命令的前几个字符并按“Ctrl-P”键或上箭头键即可。这样就会搜索出命令历史中第一个与你输入的字符相匹配的命令。多次按“Ctrl-P”键或上箭头键就会在命令历史中不断搜索。如果你错过了想要的那条命令也没关系，你可以按“Ctrl-N”键或下箭头键在命令历史中前向搜索。只要多操作几次，以后你会想都不想地按下这些键！

“Ctrl-R”用于实现部分增量搜索，跟UNIX型shell中的readline所提供的功能一样。在Windows上，IPython模拟了readline功能。按下“Ctrl-R”并输入你想搜索的行中的几个字符：

```
In [1]: a_command = foo(x, y, z)
```

```
(reverse-i-search)`com': a_command = foo(x, y, z)
```

按下“Ctrl-R”将会循环搜索命令历史中每一条与输入相符的行。

输入和输出变量

忘记把函数结果赋值给变量是一件让人很郁闷的事情。好在IPython会将输入（你输入的那些文本）和输出（返回的对象）的引用保存在一些特殊变量中。最近的两个输出结果分别保存在_（一个下划线）和__（两个下划线）变量中：

```
In [556]: 2 ** 27
Out[556]: 134217728

In [557]: _
Out[557]: 134217728
```

输入的文本被保存在名为_iX的变量中，其中X是输入行的行号。每个输入变量都有一个对应的输出变量_X。比如说，在输入完第27行后，就会产生两个新变量_27（输出变量）和_i27（输入变量）。

```
In [26]: foo = 'bar'

In [27]: foo
Out[27]: 'bar'

In [28]: _i27
Out[28]: u'foo'

In [29]: _27
Out[29]: 'bar'
```

由于输入变量是字符串，因此可以用Python的exec关键字重新执行：

```
In [30]: exec _i27
```

有几个魔术命令可用于控制输入和输出历史。%hist用于打印全部或部分输入历史，可以选择是否带行号。%reset用于清空interactive命名空间，并可选择是否清空输入和输出缓存。%xdel用于从IPython系统中移除特定对象的一切引用。详细信息请参考相应魔术命令的文档。

警告： 在处理非常大的数据集时，一定要注意IPython的输入输出历史，它会导致所有对象引用都无法被垃圾收集器处理（即释放内存），即使用del关键字将变量从interactive命名空间中删除也不行。对于这种情况，谨慎地使用%xdel和%reset将有助于避免出现内存方面的问题。

记录输入和输出

IPython能够记录整个控制台会话，包括输入和输出。执行%logstart即可开始记录日志：

```
In [3]: %logstart
Activating auto-logging. Current session state plus future input saved.
Filename       : ipython_log.py
Mode           : rotate
Output logging : False
Raw input log  : False
Timestamping   : False
State          : active
```

IPython的日志功能可以在任何时刻开启，它将记录你的整个会话（包括此前的命令）。因此，如果你在写代码的过程中，突然想要保存所有工作的时候，直接启动日志功能就行了。%logstart的具体选项（比如修改输出文件路径）请参考其文档，此外还可以看看几个与之配套的魔术命令%logoff、%logon、%logstate以及%logstop。

与操作系统交互

IPython的另一个重要特点就是它跟操作系统shell结合得非常紧密。也就是说，你可以直接在IPython中实现标准的Windows或UNIX（Linux、OS X）命令行活动。比如执行shell命令、更改目录、将命令的执行结果保存在Python对象（列表或字符串）中等。此外，它还提供了shell命令别名以及目录书签等功能。

表3-3总结了用于调用shell命令的魔术命令及其语法。我将在后面几节中简要介绍这些功能。

表3-3：跟系统相关的IPython魔术命令

命令	说明
!cmd	在系统shell中执行cmd
output = !cmd args	执行cmd，并将stdout存放在output中
%alias *alias_name cmd*	为系统shell命令定义别名
%bookmark	使用IPython的目录书签系统
%cd *directory*	将系统工作目录更改为directory
%pwd	返回系统的当前工作目录
%pushd *directory*	将当前目录入栈，并转向目标目录
%popd	弹出栈顶目录，并转向该目录
%dirs	返回一个含有当前目录栈的列表

表3-3：跟系统相关的IPython魔术命令（续）

命令	说明
%dhist	打印目录访问历史
%env	以dict形式返回系统环境变量

shell命令和别名

在IPython中，以感叹号（!）开头的命令行表示其后的所有内容需要在系统shell中执行。也就是说，你可以删除文件（根据OS的不同，使用rm或del）、修改目录或执行任意其他处理过程。甚至还可以启动一些能将控制权从IPython手中夺走的进程（比如另外再启动一个Python解释器）：

```
In [2]: !python
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul  3 2011, 15:17:51)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
Type "packages", "demo" or "enthought" for more information.
>>>
```

此外，还可以将shell命令的控制台输出存放到变量中，只需将!开头的表达式赋值给变量即可。例如，我的Linux电脑通过以太网连接到互联网，于是可以将我的IP地址存到一个Python变量中去：[译注13]

```
In [1]: ip_info = !ifconfig eth0 | grep "inet"

In [2]: ip_info[0].strip()
Out[2]: 'inet addr:192.168.1.137  Bcast:192.168.1.255  Mask:255.255.255.0'
```

返回的Python对象ip_info实际上是一个含有控制台输出结果的自定义列表类型。

在使用!时，IPython还允许使用当前环境中定义的Python值。只需在变量名前面加上美元符号（$）即可：[译注14]

```
In [3]: foo = 'test*'

In [4]: !ls $foo
test4.py  test.py  test.xml
```

魔术命令%alias可以为shell命令自定义简称。例如：

译注13：之前已经说过，作者用的不是Windows操作系统，所以这个命令自然无法执行。Windows上可以用ipconfig，但毕竟不是一样东西，这里的代码自己能看明白即可。

译注14：在Windows中，将ls换成dir。

```
In [1]: %alias ll ls -l

In [2]: ll /usr
total 332
drwxr-xr-x   2  root  root  69632  2012-01-29  20:36  bin/
drwxr-xr-x   2  root  root   4096  2010-08-23  12:05  games/
drwxr-xr-x 123  root  root  20480  2011-12-26  18:08  include/
drwxr-xr-x 265  root  root 126976  2012-01-29  20:36  lib/
drwxr-xr-x  44  root  root  69632  2011-12-26  18:08  lib32/
lrwxrwxrwx   1  root  root      3  2010-08-23  16:02  lib64 -> lib/
drwxr-xr-x  15  root  root   4096  2011-10-13  19:03  local/
drwxr-xr-x   2  root  root  12288  2012-01-12  09:32  sbin/
drwxr-xr-x 387  root  root  12288  2011-11-04  22:53  share/
drwxrwsr-x  24  root  src    4096  2011-07-17  18:38  src/
```

可以一次执行多条命令，只需将它们写在一行上并以分号隔开即可：

```
In [558]: %alias test_alias (cd ch08; ls; cd ..)

In [559]: test_alias
macrodata.csv  spx.csv        tips.csv
```

注意，IPython会在会话结束时立即“忘记”你所定义的一切别名。为了创建永久性的别名，你需要使用配置系统。本章稍后会对此进行介绍。

目录书签系统

IPython有一个简单的目录书签系统，它使你能保存常用目录的别名以便实现快速跳转。比如说，作为一名狂热的Dropbox用户，为了能够快速地转到我的Dropbox目录，我可以定义一个书签：

```
In [6]: %bookmark db /home/wesm/Dropbox/
```

在定义好书签之后，就可以在执行魔术命令%cd时使用这些书签了：

```
In [7]: cd db
(bookmark:db) -> /home/wesm/Dropbox/
/home/wesm/Dropbox
```

如果书签名与当前工作目录中的某个目录名冲突，可以通过-b标记（其作用是覆写）使用书签目录。%bookmark的-l选项的作用是列出所有书签：

```
In [8]: %bookmark -l
Current bookmarks:
db -> /home/wesm/Dropbox/
```

书签跟别名的区别在于，它们会被自动持久化。

软件开发工具

IPython不仅是一种舒适的交互式计算和数据分析环境，同时也非常适合成为一种软件开发环境。在数据分析应用程序中，最重要的事情就是拥有正确的代码。幸运的是，IPython紧密集成并加强了Python内置的pdb调试器。此外，你还希望代码运行能足够快。为此，IPython提供了一些简单易用的代码运行时间及性能分析工具。下面，我将对这些工具做一个详细介绍。

交互式调试器

IPython的调试器增强了pdb，如Tab键自动完成、语法高亮、为异常跟踪的每条信息添加上下文参考等。调试代码的最佳时机之一就是错误刚刚发生那会儿。%debug命令（在发生异常之后马上输入）将会调用那个“事后”调试器，并直接跳转到引发异常的那个栈帧（stack frame）：

```
In [2]: run ch03/ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/home/wesm/book_scripts/ch03/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
---> 15 calling_things()

/home/wesm/book_scripts/ch03/ipython_bug.py in calling_things()
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()

/home/wesm/book_scripts/ch03/ipython_bug.py in throws_an_exception()
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:

In [3]: %debug
> /home/wesm/book_scripts/ch03/ipython_bug.py(9)throws_an_exception()
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb>
```

在这个调试器中，你可以执行任意Python代码并查看各个栈帧中的一切对象和数据（也

就是解释器还“留了条生路”的那些）。默认是从最低级开始的（即错误发生的地方）。输入u（或up）和d（或down）即可在栈跟踪的各级别之间切换：

```
ipdb> u
> /home/wesm/book_scripts/ch03/ipython_bug.py(13)calling_things()
     12     works_fine()
---> 13     throws_an_exception()
     14
```

执行%pdb命令可以让IPython在出现异常之后自动调用调试器。很多人都认为这是一个非常实用的功能。

此外，调试器还可以为代码开发工作提供帮助，尤其是当你想要设置断点或对函数/脚本进行单步调试以查看各条语句的执行情况时。实现这个目的的方式有几个。第一，使用带有-d选项的%run命令，这将会在执行脚本文件中的代码之前先打开调试器。必须立即输入s（或step）才能进入脚本：[译注15]

```
In [5]: run -d ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb> s
> g:\ipython_bug.py(1)<module>()
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
```

在此之后，该文件接下来的执行方式就全凭你一句话了。比如说，在上面那个异常中，我们可以在调用works_fine方法的地方设置一个断点，然后输入c（或continue）使脚本一直运行下去直到该断点时为止：

```
ipdb> b 12
ipdb> c
> /home/wesm/book_scripts/ch03/ipython_bug.py(12)calling_things()
     11 def calling_things():
2--> 12     works_fine()
     13     throws_an_exception()
```

这时可以单步进入works_fine()或执行works_fine()（输入n（或next）直接执行到下一行[译注16]）：

```
ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(13)calling_things()
```

译注15：第一，s不一定行，看提示，要用c；第二，这个s实际上是step into。

译注16：也就是step over。

```
2    12     works_fine()
---> 13     throws_an_exception()
     14
```

然后，我们单步进入throws_an_exception并前进到发生错误的那一行，查看在此范围内的变量。注意，调试器命令的优先级高于变量名。这时在变量前面加上感叹号（！）即可查看其内容。

```
ipdb> s
--Call--
> /home/wesm/book_scripts/ch03/ipython_bug.py(6)throws_an_exception()
      5
----> 6 def throws_an_exception():
      7     a = 5

ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(7)throws_an_exception()
      6 def throws_an_exception():
----> 7     a = 5
      8     b = 6

ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(8)throws_an_exception()
      7     a = 5
----> 8     b = 6
      9     assert(a + b == 10)

ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(9)throws_an_exception()
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb> !a
5
ipdb> !b
6
```

要想精通这个交互式调试器，必须经过大量的实践才行。表3-4列出了该调试器的全部命令。如果你习惯了使用某款IDE，刚开始用这种终端型调试器的时候可能会觉得有点麻烦，但慢慢就会习惯了。虽然大部分Python IDE都拥有优秀的GUI调试器，但是在IPython中调试程序却往往会带来更高的生产率。

表3-4：(I)Python调试器命令

命令	功能
h(elp)	显示命令列表
help *command*	显示*command*的文档
c(ontinue)	恢复程序的执行

表3-4：(I)Python调试器命令（续）

命令	功能
q(uit)	退出调试器，不再执行任何代码
b(reak) *number*	在当前文件的第*number*行设置一个断点
b path/to/*file.py:number*	在指定文件的第*number*行设置一个断点
s(tep)	单步进入函数调用
n(ext)	执行当前行，并前进到当前级别的下一行
u(p)/d(own)	在函数调用栈中向上或向下移动
a(rgs)	显示当前函数的参数
debug *statement*	在新的（递归）调试器中调用语句*statement*
l(ist) *statement*	显示当前行，以及当前栈级别上的上下文参考代码
w(here)	打印当前位置的完整栈跟踪（包括上下文参考代码）

调试器的其他使用场景

除上面提到的之外，还有另外几种调用调试器的手段。第一，使用`set_trace`这个特别的函数（以`pdb.set_trace`命名），这差不多可以算作一种“穷人的断点[译注17]”。下面这两个方法可能会在你的日常工作中派上用场（你也可以像我一样直接将其添加到IPython配置中）：

```
def set_trace():
    from IPython.core.debugger import Pdb
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    from IPython.core.debugger import Pdb
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

第一个函数（`set_trace`）非常简单。你可以将其放在代码中任何希望停下来查看一番的地方（比如发生异常的地方）：

```
In [7]: run ch03/ipython_bug.py
> /home/wesm/book_scripts/ch03/ipython_bug.py(16)calling_things()
     15 set_trace()
---> 16 throws_an_exception()
     17
```

按下`c`（或continue）仍然会使代码恢复执行，不受任何影响。

译注17：作者在这里的意思是这种断点比较随便，是硬编码的。

另外那个debug函数使你能够直接在任意函数上使用调试器。假设我们写了如下函数：

```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

现在想对其进行单步调试。f的正常使用方式应该类似于f(1, 2, z=3)这个样子。为了能够单步进入f，将f作为第一个参数传给debug，后面按顺序再跟上各个需要传给f的关键字参数：

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
      1 def f(x, y, z):
----> 2     tmp = x + y
      3     return tmp / z

ipdb>
```

我发现这两个函数虽然简单，但是在日常工作当中却节省了我不少的时间。

此外，这个调试器还可以结合%run使用。通过%run -d执行脚本，你将会直接进入调试器，然后可以设置一些断点并启动脚本：

```
In [1]: %run -d ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb>
```

如果再加上-b和一个行号，则调试器在启动时就会自动设置一个断点：

```
In [2]: %run -d -b2 ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:2
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/book_scripts/ch03/ipython_bug.py(2)works_fine()
      1 def works_fine():
1---> 2     a = 5
      3     b = 6

ipdb>
```

测试代码的执行时间：%time和%timeit

对于规模更大、运行时间更长的数据分析应用程序，你可能会希望测试一下各个部分或函数调用或语句的执行时间。你可能会希望了解某个复杂计算过程中到底是哪些函数占

用的时间最多。幸运的是，在开发和测试代码的过程中，IPython能够让你轻松得到这些信息。使用内置的time模块及其time.clock和time.time函数手工测试代码执行时间是一件令人烦闷的事情，因为你必须编写许多一模一样的了无生趣的公式化代码：

```
import time
start = time.time()
for i in range(iterations):
    # 这里放一些待执行的代码
    elapsed_per = (time.time() - start) / iterations
```

由于这是一个非常常用的功能，所以IPython专门提供了两个魔术函数（%time和%timeit）以便自动完成该过程。%time一次执行一条语句，然后报告总体执行时间。假设我们有一大堆字符串，希望对几个“能够选出具有特殊前缀的字符串”的函数进行比较。下面是一个拥有60万字符串的数组，以及两个不同的“能够选出其中以foo开头的字符串”的方法：

```
# 一个非常大的字符串数组
strings = ['foo', 'foobar', 'baz', 'qux', 'python', 'Guido Van Rossum'] * 100000

method1 = [x for x in strings if x.startswith('foo')]

method2 = [x for x in strings if x[:3] == 'foo']
```

看上去它们的性能表现应该差不多，对吧？我们通过%time来确认一下：

```
In [561]: %time method1 = [x for x in strings if x.startswith('foo')]
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
Wall time: 0.19 s

In [562]: %time method2 = [x for x in strings if x[:3] == 'foo']
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
```

墙上时间（Wall time）是我们最感兴趣的数字。所以，看上去第一个方法耗费了两倍以上的时间，但这并不是一个非常精确的结果。如果你对相同语句多次执行%time的话，就会发现其结果是会变的。为了得到更为精确的结果，需要使用魔术函数%timeit。对于任意语句，它会自动多次执行以产生一个非常精确的平均执行时间。

```
In [563]: %timeit [x for x in strings if x.startswith('foo')]
10 loops, best of 3: 159 ms per loop

In [564]: %timeit [x for x in strings if x[:3] == 'foo']
10 loops, best of 3: 59.3 ms per loop
```

这个貌似平淡无奇的例子正好说明了一个事实：我们非常有必要了解Python标准库、NumPy、pandas以及本书中所用到的其他库的性能特点。在大型数据分析应用程序中，这些不起眼的毫秒数是会不断累积的！

对于那些执行时间非常短（甚至是那些微秒（1e-6秒）或纳秒（1e-9秒）级的）的分析语句和函数而言，%timeit是非常有用的。虽然这些时间值小到几乎可以忽略不计，但同样执行100万次一个20微秒的函数，所用的时间要比一个5微秒的多15秒。在上面那个例子中，我们可以直接对那两个字符串运算进行比较以了解其性能特点：

```
In [565]: x = 'foobar'

In [566]: y = 'foo'

In [567]: %timeit x.startswith(y)
1000000 loops, best of 3: 267 ns per loop

In [568]: %timeit x[:3] == y
10000000 loops, best of 3: 147 ns per loop
```

基本性能分析：%prun和%run -p

代码的性能分析跟代码执行时间密切相关，只不过它关注的是耗费时间的位置。主要的Python性能分析工具是cProfile模块，它不是专为IPython设计的。cProfile在执行一个程序或代码块时，会记录各函数所耗费的时间。

cProfile一般是在命令行上使用的，它将执行整个程序然后输出各函数的执行时间。假设我们有一个简单的脚本：在一个循环中执行一些线性代数计算（计算一个100×100的矩阵的最大本征值绝对值）。

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in xrange(niter):
        mat = np.random.randn(K, K)
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)
    return results
some_results = run_experiment()
print 'Largest one we saw: %s' % np.max(some_results)
```

如果你还不懂NumPy，暂时先别管，后面会讲的。在命令行中输入下列命令即可通过cProfile启动该脚本：

```
python -m cProfile cprof_example.py
```

执行之后，你会发现输出结果是按函数名排序的。这让我们很难发现哪里才是最花时间的地方，因此通常都会再用-s标记指定一个排序规则：

```
$ python -m cProfile -s cumulative cprof_example.py
Largest one we saw: 11.923204422
     15116 function calls (14927 primitive calls) in 0.720 seconds

Ordered by: cumulative time

ncalls  tottime       percall  cumtime  percall filename:lineno(function)
1   0.001    0.001    0.721    0.721 cprof_example.py:1(<module>)
100 0.003    0.000    0.586    0.006 linalg.py:702(eigvals)
200 0.572    0.003    0.572    0.003 {numpy.linalg.lapack_lite.dgeev}
1   0.002    0.002    0.075    0.075 __init__.py:106(<module>)
100 0.059    0.001    0.059    0.001 {method 'randn')
1   0.000    0.000    0.044    0.044 add_newdocs.py:9(<module>)
2   0.001    0.001    0.037    0.019 __init__.py:1(<module>)
2   0.003    0.002    0.030    0.015 __init__.py:2(<module>)
1   0.000    0.000    0.030    0.030 type_check.py:3(<module>)
1   0.001    0.001    0.021    0.021 __init__.py:15(<module>)
1   0.013    0.013    0.013    0.013 numeric.py:1(<module>)
1   0.000    0.000    0.009    0.009 __init__.py:6(<module>)
1   0.001    0.001    0.008    0.008 __init__.py:45(<module>)
262 0.005    0.000    0.007    0.000 function_base.py:3178(add_newdoc)
100 0.003    0.000    0.005    0.000 linalg.py:162(_assertFinite)
...
```

这里只给出了输出结果中的前15行。只需查看cumtime列即可发现各函数所耗费的总时间。注意，如果一个函数调用了别的函数，计时器是不会停下来重新计时的。cProfile记录的是各函数调用的起始和结束时间，并依此计算总时间。

除命令行用法之外，cProfile还可以编程的方式分析任意代码块的性能。IPython为此提供了一个方便的接口，即%prun命令和带-p选项的%run。%prun的格式跟cProfile差不多，但它分析的是Python语句而不是整个.py文件：

```
In [4]: %prun -l 7 -s cumulative run_experiment()
         4203 function calls in 0.643 seconds

Ordered by: cumulative time
List reduced from 32 to 7 due to restriction <7>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.643    0.643 <string>:1(<module>)
     1    0.001    0.001    0.643    0.643 cprof_example.py:4(run_experiment)
   100    0.003    0.000    0.583    0.006 linalg.py:702(eigvals)
   200    0.569    0.003    0.569    0.003 {numpy.linalg.lapack_lite.dgeev}
   100    0.058    0.001    0.058    0.001 {method 'randn'}
   100    0.003    0.000    0.005    0.000 linalg.py:162(_assertFinite)
   200    0.002    0.000    0.002    0.000 {method 'all' of 'numpy.ndarray' objects}
```

执行%run -p -s cumulative cprof_example.py也能达到上面那条系统命令行命令一样的效果，但是却无需退出IPython。

逐行分析函数性能

有些时候，从%prun（或其他基于cProfile的性能分析手段）得到的信息要么不足以说明函数的执行时间，要么就复杂到难以理解（按函数名聚合）。对于这种情况，我们可以使用一个叫做line_profiler的小型库（可以通过PyPI或随便一种包管理工具获取）。其中有一个新的魔术函数%lprun，它可以对一个或多个函数进行逐行性能分析。你可以修改IPython配置（参考IPython文件或本章稍后关于配置的内容）以启用这个扩展，代码如下所示：

```
# A list of dotted module names of IPython extensions to load.
c.TerminalIPythonApp.extensions = ['line_profiler']
```

line_profiler可以通过编程的方式使用（请参阅完整文档），但其最强大的一面却是在IPython中的交互式使用。假设你有一个prof_mod模块，其中有一些用于NumPy数组计算的代码，如下所示：

```
from numpy.random import randn

def add_and_sum(x, y):
    added = x + y
    summed = added.sum(axis=1)
    return summed

def call_function():
    x = randn(1000, 1000)
    y = randn(1000, 1000)
    return add_and_sum(x, y)
```

如果我们想了解add_and_sum函数的性能，%prun会给出如下所示的结果：

```
In [569]: %run prof_mod

In [570]: x = randn(3000, 3000)
In [571]: y = randn(3000, 3000)

In [572]: %prun add_and_sum(x, y)
         4 function calls in 0.049 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1   0.036    0.036   0.046    0.046 prof_mod.py:3(add_and_sum)
     1   0.009    0.009   0.009    0.009 {method 'sum' of 'numpy.ndarray' objects}
     1   0.003    0.003   0.049    0.049 <string>:1(<module>)
     1   0.000    0.000   0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

这个结果并不能说明什么问题。启用line_profiler这个IPython扩展之后，就会出现一个新的魔术命令%lprun。用法上唯一的区别就是：必须为%lprun指明想要测试哪个或哪些函数。%lprun的通用语法为：

```
%lprun -f func1 -f func2 statement_to_profile
```

在本例中，我们想要测试的是add_and_sum，于是执行：

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: book_scripts/prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def add_and_sum(x, y):
     4      1           36510  36510.0     79.5       added = x + y
     5      1            9425   9425.0     20.5       summed = added.sum(axis=1)
     6      1               1      1.0      0.0       return summed
```

这个结果就容易理解多了。这里我们测试的只是add_and_sum这一个函数。上面那个模块中还有一个call_function函数，我们可以结合add_and_sum一起测试，于是最终的测试命令就成了下面这个样子：

```
In [574]: %lprun -f add_and_sum -f call_function call_function()
Timer unit: 1e-06 s
File: book_scripts/prof_mod.py
Function: add_and_sum at line 3
Total time: 0.005526 s
Line #  Hits Time  Per Hit    % Time  Line Contents
==============================================================
    3                                 def add_and_sum(x, y):
    4     1  4375     4375.0   79.2       added = x + y
    5     1  1149     1149.0   20.8       summed = added.sum(axis=1)
    6     1     2        2.0    0.0       return summed
File: book_scripts/prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s
Line #  Hits Time    Per Hit  % Time  Line Contents
==============================================================
    8                                 def call_function():
    9     1   57169    57169.0   47.2       x = randn(1000, 1000)
   10     1   58304    58304.0   48.2       y = randn(1000, 1000)
   11     1    5543     5543.0    4.6       return add_and_sum(x, y)
```

通常，我会用%prun（cProfile）做“宏观的”性能分析，而用%lprun（line_profiler）做“微观的” 性能分析。这两个工具都很有必要了解一下。

注意： 在使用%lprun时，之所以必须显式指明待测试的函数名，是因为“跟踪”每一行代码的执行时间所需的开销很大。对不感兴趣的函数进行跟踪将会对性能分析结果造成显著的影响。

IPython HTML Notebook

2011年，由Brian Granger领导的IPython团队开始开发一种基于Web技术的交互式计算文

档格式，即IPython Notebook（见图3-4）。目前，它已经成为一种非常棒的交互式计算工具，同时还是科研和教学的一种理想媒介。本书中大部分示例都是用它编写的.我强烈建议你也试试。

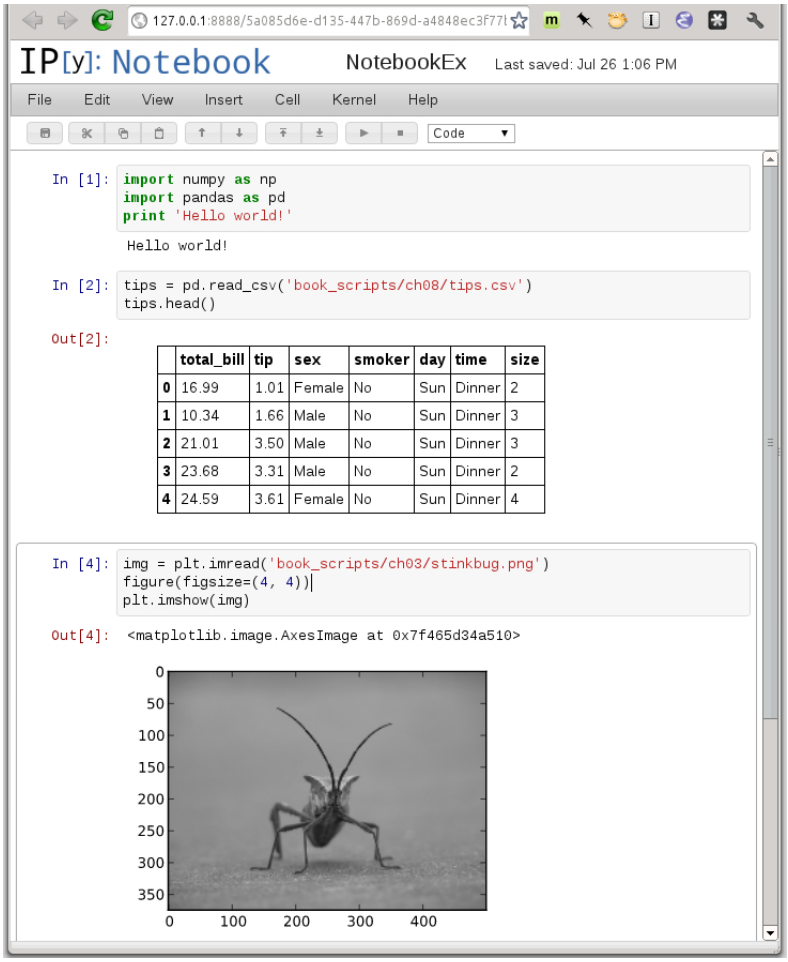

图3-4：IPython Notebook

它有一种基于JSON的文档格式.ipynb，使你可以轻松分享代码、输出结果以及图片等内容。目前在各种Python研讨会上，一种流行的演示手段就是使用IPython Notebook，然后再将.ipynb文件发布到网上以供所有人查阅。

IPython Notebook应用程序是一个运行于命令行上的轻量级服务器进程。执行下面这条命令即可启动：

```
$ ipython notebook --pylab=inline
[NotebookApp] Using existing profile dir: u'/home/wesm/.config/ipython/profile_default'
[NotebookApp] Serving notebooks from /home/wesm/book_scripts
[NotebookApp] The IPython Notebook is running at: http://127.0.0.1:8888/
[NotebookApp] Use Control-C to stop this server and shut down all kernels.
```

在大多数平台上，你的首选Web浏览器会自动打开Notebook的仪表板（dashboard）。有时你可能需要手工打开上面列出的那个URL。你可以在这里创建一个新的记事本并开始研究工作。

由于我们是在一个Web浏览器中使用Notebook的，因此该服务器进程可以运行于任何地方。你甚至可以连接到那些运行在云服务（如Amazon EC2）上的Notebook。直到写作本书时为止，一个新的名为NotebookCloud（*http://notebookcloud.appspot.com*）的项目已经诞生了，它可以轻松地在Amazon EC2上启动记事本。

利用IPython提高代码开发效率的几点提示

为了在IPython中开发、调试代码，并充分发挥其交互优势，许多用户都需要转换一下工作模式。像编码风格以及一些操作细节可能需要做一些调整。

就这点来说，本节的内容更像是艺术而非科学，你需要有一些编程经验才好判断其能否提高你的工作效率。总之，你得让你的代码结构更易于交互且结果更易于查看。我发现通过IPython设计的软件要比独立的命令行应用程序好用。当你执行自己或别人在几个月甚至几年前编写的代码时出现了错误，想找出问题所在时，IPython的交互性就会变得非常重要。

重新加载模块依赖项

在Python中，当你输入`import some_lib`时，`some_lib`中的代码就会被执行，且其中所有的变量、函数和引入项都会被保存在一个新建的`some_lib`模块命名空间中。下次你再输入`import some_lib`时，就会得到这个模块命名空间的一个引用。而这对于IPython的交互式代码开发模式就会有一个问题，比如说，用`%run`执行的某段脚本中牵扯到了某个刚刚做了修改的模块。假设我们有一个`test_script.py`文件，其中有下列代码：

```
import some_lib

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

如果在执行了`%run test_script.py`之后又对`some_lib.py`进行了修改，下次再执行`%run`

test_script.py时将仍然会使用老版的some_lib。其原因就是Python的“一次加载”模块系统。这个行为不同于其他一些数据分析环境（如MATLAB，它会自动应用代码修改[注1]）。为了解决这个问题，你有两个办法可用。第一个办法是使用Python内置的reload函数。将test_script.py修改成下面这个样子：

```
import some_lib
reload(some_lib)

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

这样就保证每次执行test_script.py时都能用上最新版的some_lib了。显然，当依赖变得更强时，就需要在很多地方插入很多的reload。对于这个问题，IPython提供了一个特殊的dreload函数（非魔术函数）来解决模块的“深度”（递归）重加载。如果执行import some_lib之后再输入dreload(some_lib)，则它会尝试重新加载some_lib及其所有的依赖项。遗憾的是，这个办法也不是万灵丹，但是如果真的不行了，重启IPython就行了。

代码设计提示

这个问题不太好讲，但我在日常工作中确实发现了一些高层次的原则。

保留有意义的对象和数据

人们一般不会在命令行上编写下面这样的程序：

```
from my_functions import g

def f(x, y):
    return g(x + y)

def main():
    x = 6
    y = 7.5
    result = x + y

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

如果我们在IPython中执行这段代码的话会出现什么问题？我们在IPython shell中将访问不到任何结果以及main函数中定义的对象。好点的办法是直接在该模块的全局命名空

注1：由于一个模块或包可能会在一个程序中的不同位置多次引入，所以Python会在第一次引入这些模块时对其进行缓存，而不是每次都执行模块中的代码。否则，应用程序的模块化和良好的代码组织等手段就达不到高效的目的了。

间中执行main中的代码（如果你希望该模块是可引入的，也可以将这些代码放在if __name__ == '__main__':块中）。这样，当你%run这段代码时，就能看到main中定义的所有变量了。对这个简单的例子而言，这个原则意义不大，但对本书后面将要介绍的那些针对大数据集的复杂数据分析问题而言就很重要了。

扁平结构要比嵌套结构好

深度嵌套的代码让我想到了洋葱。在测试或调试函数时，你要把这个洋葱剥多少层才能找到感兴趣的代码？“扁平结构要比嵌套结构好”的思想来自“Zen of Python”[译注18]，它对交互式的代码开发模式同样有效。编写函数和类时应尽量注意低耦合和模块化，这样可以使它们更易于测试（如果你编写单元测试的话）、调试和交互式使用。

无惧大文件

如果曾经学过Java（或其他类似的语言），可能会有人告诉你要“尽量保持文件的小型化”。在许多语言中，这都是一个不错的建议。长度太长通常是一种不好的“臭代码”，意味着需要重构或重组。然而在IPython中开发代码时，处理10个小的（但互相关联的）文件（比如都低于100行）可能会让你更为头疼，还不如直接一个大文件或两三个大点的文件来得痛快。更少的文件意味着需要重新加载的模块更少，编辑时需要在各个文件之间的跳转次数也更少。我发现维护更大的（具有高内聚度的）模块会更实用也更具有Python特点。在解决完问题之后，有时将大文件拆分成小文件会更好。

显然，我并不建议将此原则极端化，那可能会让你将所有代码都放到一个巨大的文件里面。对一个大型代码库而言，要找到一种合乎逻辑的模块/包架构需要花点工夫，但这对团队工作非常重要。每个模块都应该具有足够高的内聚度，而且要能足够直观地找到对应各种功能的函数和类。

高级IPython功能

让你的类对IPython更加友好

IPython力求为各种对象呈现一个友好的字符串表示。对于许多对象（如字典、列表和元组等），内置的pprint模块就能给出漂亮的格式。但是对于你自己定义的那些类，就必须自己生成所需的字符串输出。假设我们有下面这个简单的类：

```
class Message:
    def __init__(self, msg):
        self.msg = msg
```

译注18：这是Tim Peters 2004年写的一首“诗”，执行“import this”就能看到。有网民将其翻译成三字经的形式（又名“蛇宗三字经”）。另外，有兴趣的话，可以看看this的源代码。

如果像下面这样写，你就会失望地发现这个类的默认输出形式非常不好看：

```
In [576]: x = Message('I have a secret')

In [577]: x
Out[577]: <__main__.Message instance at 0x60ebbd8>
```

由于IPython会获取__repr__方法返回的字符串（具体办法是output = repr(obj)），并将其显示到控制台上。因此，我们可以为上面那个类添加一个简单的__repr__方法以得到一个更有意义的输出形式：

```
class Message:
    def __init__(self, msg):
        self.msg = msg

    def __repr__(self):
        return 'Message: %s' % self.msg

In [579]: x = Message('I have a secret')

In [580]: x
Out[580]: Message: I have a secret
```

个性化和配置

IPython shell在外观（如颜色、提示符、行间距等）和行为方面的大部分内容都是可以进行配置的。下面是能够通过配置做的部分事情：

- 修改颜色方案。
- 修改输入输出提示符。
- 去掉Out提示符跟下一个In提示符之间的空行。
- 执行任意Python语句。这些语句可以用于引入所有常用的东西，还可以做一些你希望每次启动IPython都发生的事情。
- 启用IPython扩展，如line_profiler中的魔术命令%lprun。
- 定义你自己的魔术命令或系统别名。

所有这些配置选项都定义在一个叫做ipython_config.py的文件中，可以在~/.config/ipython/目录（UNIX）和%HOME%/.ipython/ 目录（Windows）中找到。具体的主目录取决于你的系统。配置信息是基于特定个性化设置的。一般来说，正常启动IPython将会加载默认的个性化设置（位于profile_default目录中）。因此，在我的Linux系统中，默认IPython配置文件的完整路径是：

```
/home/wesm/.config/ipython/profile_default/ipython_config.py
```

这里我就不对该文件的内容作详细介绍了。因为其注释已经说明了各个配置项的功能，各位读者完全可以自己照着做。还有一个很实用的功能是拥有多个个性化设置。假设你想要专门为某个应用程序或项目量身定做一套IPython配置。输入下面这样的命令即可新建一个个性化设置：

```
ipython profile create secret_project
```

然后编辑新建的这个profile_secret_project目录中的配置文件，再用下面这种方式启动IPython：

```
$ ipython --profile=secret_project
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul  3 2011, 15:17:51)
Type "copyright", "credits" or "license" for more information.

IPython 0.13 -- An enhanced Interactive Python.
?   -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help-> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

IPython profile: secret_project

In [1]:
```

同样，有关个性化和配置方面的详细信息，请参考IPython的在线文档。

致谢

本章的部分内容由IPython Development Team整理。我对他们创建了如此神奇的工具而感激涕零。

利用Python进行数据分析

还在苦苦寻觅用Python控制、处理、整理、分析结构化数据的完整课程？本书含有大量的实践案例，你将学会如何利用各种Python库（包括NumPy、pandas、matplotlib以及IPython等）高效地解决各式各样的数据分析问题。

由于作者Wes McKinney是pandas库的主要作者，所以本书也可以作为利用Python实现数据密集型应用的科学计算实践指南。本书适合刚刚接触Python的分析人员以及刚刚接触科学计算的Python程序员。

- 将IPython这个交互式Shell作为你的首要开发环境。
- 学习NumPy（Numerical Python）的基础和高级知识。
- 从pandas库的数据分析工具开始。
- 利用高性能工具对数据进行加载、清理、转换、合并以及重塑。
- 利用matplotlib创建散点图以及静态或交互式的可视化结果。
- 利用pandas的groupby功能对数据集进行切片、切块和汇总操作。
- 处理各种各样的时间序列数据。
- 通过详细的案例学习如何解决Web分析、社会科学、金融学以及经济学等领域的问题。

Wes McKinney 资深数据分析专家，对各种Python库（包括NumPy、pandas、matplotlib以及IPython等）都有深入研究，并在大量的实践中积累了丰富的经验。撰写了大量与Python数据分析相关的经典文章，被各大技术社区争相转载，是Python和开源技术社区公认的权威人物之一。开发了用于数据分析的著名开源Python库——pandas，广获用户好评。在创建Lambda Foundry（一家致力于企业数据分析的公司）之前，他曾是AQR Capital Management的定量分析师。

“科学计算和数据分析社区已经等待这本书很多年了：大量具体的实践建议，以及大量综合应用方法。本书在未来几年里肯定会成为Python领域中技术计算的权威指南。”

——Fernando Pérez
加州大学伯克利分校
研究科学家，
IPython的创始人之一

O'REILLY®
oreilly.com.cn

O'Reilly Media, Inc.授权机械工业出版社出版

客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
投稿热线：(010) 88379604

数字阅读：www.hzmedia.com.cn
华章网站：www.hzbook.com
网上购书：www.china-pub.com

上架指导：计算机/程序设计
ISBN 978-7-111-43673-7
9 787111 436737 >
定价：89.00元