TURING 图灵程序设计丛书

Apress®

Python Data Analytics

Python 数据分析实战

了解数据分析全貌，全面掌握用Python语言
及专业的库进行数据分析，驾驭大数据

【意】 Fabio Nelli 著
杜春晓 译

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Python Data Analytics

Python 数据分析实战

【意】 Fabio Nelli 著
杜春晓 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

Python数据分析实战 / (意) 内利 (Fabio Nelli) 著 ; 杜春晓译. — 北京 : 人民邮电出版社, 2016.8
(图灵程序设计丛书)
ISBN 978-7-115-43220-9

Ⅰ. ①P… Ⅱ. ①内… ②杜… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2016)第180326号

内 容 提 要

Python简单易学，拥有丰富的库，并且具有极强的包容性。本书展示了如何利用Python语言的强大功能，以最小的编程代价进行数据的提取、处理和分析，主要内容包括：数据分析和Python的基本介绍，NumPy库，pandas库，如何使用pandas读写和提取数据，用matplotlib库和scikit-learn库分别实现数据可视化和机器学习，以实例演示如何从原始数据获得信息、D3库嵌入和手写体数字的识别。

本书适合数据分析师等所有需要进行数据采集分析的工作人员。

◆ 著 [意] Fabio Nelli
译 杜春晓
责任编辑 朱 巍
执行编辑 贺子娟 李 敏
责任印制 彭志环

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京市昌平百善印刷厂印刷

◆ 开本：800×1000 1/16
印张：18.75
字数：443千字 2016年8月第1版
印数：1－3 500册 2016年8月北京第1次印刷
著作权合同登记号 图字：01-2016-5330号

定价：59.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第8052号

版权声明

致　　谢

感谢我的友人，尤其要感谢Alberto、Daniele、Roberto和Alex。本书写作历时一年，其间困难重重，挫折不断，感谢他们包容我，并给予我宝贵的精神支持。

向我的母亲致以最深的谢意。

译 者 序

不知不觉，结识Python已有七个年头，掰着指头数完，不禁惊恐于光阴之易逝，叹人生之不能加长，更兼想起宇宙之无穷，免不了慨叹生命之短暂而微茫，此时更觉Python之哲学观“人生短暂，我用Python”虽朴实无华，却更楚楚动人。七年前，国内Python书籍寥寥无几，七年后已能排满一个小书架。见Python这样的好东西为世人所接受，颇感欣慰。Python在Web开发、网络编程、自然语言处理、图像处理以及本书所讲的数据分析等诸多领域都有着广泛的应用。Python简单易学，新手见它大可不必发怵；Python拥有丰富的库，开发者不必重造轮子；Python有极强的包容性，可整合C、C++等语言的代码，弥补其性能上的不足；再加上近年来计算机效率的提高，Python的优势越发凸显出来。即使是其他领域的从业者，为满足业务的需要，若在性能上没有追求极致的嗜好，Python很可能是最适合的编程语言，对于数据分析亦是如此。

互联网的迅猛发展带来了数据量的指数级增长，而数据增长速度仍在加快。无论从规模上还是结构上讲，数据分析工作面对的对象较以往发生了质的改变。各行各业所产生的数据，在过去也许只是被视为副产品，如今也有可能显露它们的价值。但数据规模之大、结构之复杂，纯人工难以胜任，求诸工具是必然。为了满足这一需求，Python数据分析的羽翼随之不停地生长，如今已丰满有力。NumPy、pandas和matplotlib等库提供了矩阵运算、数据读写和处理及绘图等一揽子解决方案。就拿数据可视化来讲，3D、等值线图、地区分布图和玫瑰图等统统不在话下，用IPython Notebook分析完数据，可直接生成各种图表，你甚至还可以拿它来做汇报，连PPT都省了。

诚如作者在书中所言，用Python做数据分析根本不用羡慕其他语言的数据分析工具集。然而，好工具摆在那里，但没有明白人教也是白搭，不过，你手中所托之书刚好能充当这方面的良师益友。它不仅能带你一览数据分析全貌，更力求一招一式地教会你数据分析的十八般武艺。本书示例颇丰，在学习过程中，若能打开IPython Notebook，一点点跟着作者比划，想必新人也能出师，而有一定水平的开发者则可将其作为案头常备的参考书，以便节省不少查阅文档的时间。本书最后，作者举了三个实例，以加深读者对数据分析全过程的理解。第一个例子，用数据分析方法探索海洋对气候的影响，你也许能从中得出足以令当年地理老师心服口服的结论。第二个例子讲解了地区分布图的制作，你会发现原来我们还可以在Python环境中使用JavaScript！第三个例子则是用机器学习方法解决经典的图像识别问题。说了这么多，你是不是像我一样觉得这本书很有趣？在翻译本书的过程中，我安装了Anaconda，从此迷上了IPython Notebook的交互性和直观生动，想必你也会为之所动。

经冬历春，译文乃成，其中半数章节的第一稿是在寒假期间完成的。在家那段时间，有时只

有我跟宝宝在屋里，我很想哄她开心，可为了保证进度，实在脱不开身。她要是哭，我就敲敲桌子，抑或是招招手，她见还有人在就不哭了，可怜的娃。岳父岳母一家人帮着妻子照看孩子不辞辛苦。我在翻译时，他们还会悄悄端过一杯茶来。地脏了，他们就轻轻把地拖了，生怕打扰我。到了饭点，岳母又会张罗出一桌可口的饭菜，如此种种，甚是难忘。没有他们，我哪有闲心码字。

感谢作者Fabio Nelli给我们带来了一顿数据分析的饕餮大餐。感谢图灵公司的朱巍编辑等诸位朋友，本书中文版的顺利出版离不开你们幕后的辛勤付出。此外，邵有生阅读了第1章和第2章译稿，范明武阅读了第3章，研究生同学黄毅阅读了第7章。他们发现了几处错误，并提出了很多非常有价值的修改意见，在此对他们表示诚挚的谢意。感谢在翻译过程中给予我鼓励、支持和帮助的诸位老师、同事和朋友，他们是路本福、都帮森、蔡波、蔡颖、陈健锁、韩旭、李玲玲、秦敏、王海霞、辛欣和王晶，我曾向他们中的几位请教过某些专业问题。感谢我初中时代的地理老师朱怡峰老先生，他激发了我对地理学科的兴趣，以至于多少年后，我在翻译本书第9章气象数据分析时会感到趣味盎然。最后感谢我的父亲和姐姐，他们以我翻译本书为荣。

由于本人学识有限，且时间仓促，书中翻译错误、不当和疏漏之处在所难免，还望读者批评指正。

杜春晓

2016年4月26日

目　录

第1章 数据分析简介

欢迎来到数据分析世界。作为后续章节的铺垫，本章介绍数据分析的主要概念和流程。完成本章的学习，你就能在数据分析的世界中迈出坚实的一步。其余章节会陆续介绍如何借助Python库，把在这里学到的概念和流程转化为Python代码。

1.1 数据分析

当今世界对信息技术的依赖程度日渐加深，每天都会产生和存储海量的数据。数据的来源多种多样——自动检测系统、传感器和科学仪器等。不知你有没有意识到，你每次从银行取钱、买东西、写博客、发微博也会产生新的数据。

什么是数据呢？数据实际上不同于信息，至少在形式上不一样。对于没有任何形式可言的字节流，除了其数量、用词和发送的时间外，其他一无所知，一眼看上去，很难理解其本质。信息实际上是对数据集进行处理，从中提炼出可用于其他场合的结论，也就是说，它是对数据进行处理后得到的结果。从原始数据中抽取信息的这个过程叫作*数据分析*。

数据分析的目的正是抽取不易推断的信息，而一旦理解了这些信息，就能够对产生数据的系统的运行机制进行研究，从而对系统可能的响应和演变作出预测。

数据分析最初用作数据保护，现已发展成为*数据建模*的方法论，从而完成了到一门真正学科的蜕变。模型实际上是指将所研究的系统转化为数学形式。一旦建立数学或逻辑模型，对系统的响应能作出不同精度的预测，我们就可以预测在给定输入的情况下，系统会给出怎样的输出。这样看来，数据分析的目标不止于建模，更重要的是其预测能力。

模型的预测能力不仅取决于建模技术的质量，还取决于选择供分析用的优质数据集的能力。因此*数据搜寻*、*数据提取*和*数据准备*等预处理工作也属于数据分析的范畴，它们对最终结果有重要影响。

到现在为止，我们一直在讲数据、数据的准备及数据处理。在数据分析的各个阶段，还有各种各样的*数据可视化*方法。无论是孤立地看数据，还是将其放到整个数据集来看，理解数据的最好方法莫过于将其做成可视化图形，从而传达出数字中蕴含（有时是隐藏着）的信息。到目前为止，已经有很多可视化模式：类型多样的图表。

数据分析的产出为模型和图形化展示，据此可预测所研究系统的响应；随后进入测试阶段，

用已知输出结果的一个数据集对模型进行测试。这些数据不是用来生成模型的，而是用来检验系统能否重现实际观察到的输出，从而掌握模型的误差，了解其有效性和局限。

拿新模型的测试结果与既有模型进行对比便可知优劣。如新模型胜出，即可进行数据分析的最后一步：部署。部署阶段需要根据模型给出的预测结果，实现相应的决策，同时还要防范模型预测到的潜在风险。

很多工作都离不开数据分析。了解数据分析及实际操作方法，对工作中做出可靠决策大有裨益。有了它，人们可以检验假说，加深对系统的理解。

1.2　数据分析师的知识范畴

数据分析学科研究的问题面很广。数据分析过程要用到多种工具和方法，它们对计算、数学和统计思维要求较高。

因此，一名优秀的数据分析师必须具备多个学科的知识和实际应用能力。这些学科中有的是数据分析方法的基础，熟练掌握它们很有必要。根据应用领域、研究项目的不同，数据分析师可能还需要掌握其他相关学科的知识。总的来说，这些知识可以帮助分析师更好地理解研究对象以及需要什么样的数据。

一般而言，对于大的数据分析项目，最好组建一个由各个相关领域的专家组成的团队，他们要能在各自擅长的领域发挥出最大作用。对于小点的项目，一名优秀的分析师就能胜任，但是他必须善于识别数据分析过程中遇到的问题，知道解决问题需要哪些学科的知识和技能，并能及时学习这些学科，有时甚至需要向相关领域的专家请教。简言之，分析师不仅要知道怎么搜寻数据，更应该懂得怎么寻找处理数据的方法。

1.2.1　计算机科学

不论从事什么领域的数据分析工作，掌握计算机科学知识对分析师来说都是最基本的要求。只有具备良好的计算机科学知识及实际应用经验，才能熟练掌握数据分析必备工具。事实上，数据分析的各个步骤都离不开计算机技术，比如用于计算的软件（IDL、Matlab等）和编程语言（C++、Java、Python等）。

要高效地处理随信息技术迅猛发展而产生的海量数据，就必须用到特定的技能。数据研究和抽取，要求分析师掌握各种常见格式的处理技巧。数据通常以某种结构组织在一起，存储于文件或数据库表中，格式多样。常见的数据存储格式有XML、JSON、XLS、CSV等。很多应用都能处理这些格式的数据文件。从数据库中获取数据要稍微麻烦些，需要掌握SQL数据库查询语言，或使用专门为从某种数据库抽取数据而开发的软件。

此外，一些特定类型的数据研究任务中，分析师所能拿到的不是立刻就能用的干净数据，而是文本文件（文档、日志）或网页。需要的数据则来自这些文件中的图表、测量值、访客量或者HTML表格，而解析文件、抽取数据（数据抓取）需要专业知识。

因此，学习信息技术知识很有必要，只有这样才能掌握在当代计算机科学基础上发展起来的

各种工具，比如软件和编程语言。数据分析和可视化离不开它们。

本书尽可能全面地介绍用Python编程语言及专业的库进行数据分析所需的全部知识。针对数据分析的各个阶段，从数据研究、数据挖掘到预测模型研究结果的部署，Python都有专门的库。

1.2.2 数学和统计学

数据分析涉及大量数学知识，本书全篇都少不了它们的身影。数据处理和分析过程涉及的数学知识可能会很复杂。因此具备扎实的数学功底显得尤为重要，至少要能够理解正在做的事。熟悉常用的统计学概念也很有必要，因为所有对数据进行的分析和解释都以这些概念为基础。如果说计算机科学提供的是数据分析工具，那么统计学提供的就是基础概念。

统计学为分析师提供了很多工具和方法，全部掌握它们需要多年的磨练。数据分析领域最常用到的统计技术有：

- 贝叶斯方法
- 回归
- 聚类

用到这些方法时，会发现其中数学和统计学知识紧密结合，且对两者都有很高的要求。但是在本书中所讲述的Python库的帮助下，读者将有能力驾驭它们。

1.2.3 机器学习和人工智能

数据分析领域最先进的工具之一就是机器学习方法。实际上，尽管数据可视化以及聚类和回归等技术对分析师发现有价值的信息有很大帮助，但在数据分析过程中，分析师经常需要查询数据集中的各种模式，这些步骤专业性很强。

机器学习这门学科所研究的正是如何把一系列步骤和算法结合起来，分析数据，识别数据中存在的模式，找出不同的簇，发现趋势，从数据中抽取有用信息用于数据分析，并实现整个过程的自动化。

机器学习日渐成为数据分析的基础工具，因此了解它（至少也要知道个大概）对数据分析工作的重要性不言而喻。

1.2.4 数据来源领域

数据来源领域（生物、物理、金融、材料试验和人口统计等）的知识也是非常重要的一块。事实上，分析师虽然受过统计学的专业训练，但是他也必须深入到应用领域，记录原始数据，以便更好地理解数据生成过程。此外，数据不仅仅是干巴巴的字符串或数字，还是实际观测参数的表达式，更确切地说是其度量值。因此，对数据来源领域有深入的理解，能够提升解释数据的能力。当然，即使是对乐意学习的分析师来说，学习特定领域的知识也是要下一番工夫的。因此最好能找到相关领域的专家，以便有问题时及时咨询。

1.3　理解数据的性质

数据分析所研究的对象自然是数据。在数据分析的各个阶段，数据都是主要关注对象。要分析、处理的原材料由数据构成。经过处理、分析数据后，最终可能会从中得到有用的信息。这些信息能够增加对研究对象，也就是产生原始数据的系统理解。

1.3.1　数据到信息的转变

数据是对世界万物的记录。任何可以被测量或是分类的事物都能用数据来表示。采集完数据后，可以对其进行研究和分析，以理解事物的性质。人们也常常借助它们进行预测，或者即使做不到预测，至少也能让推测更加有根据。

1.3.2　信息到知识的转变

当信息转化为一组有助于更好地理解特定机制的规则时，就说信息已转化为知识，我们也因而可以用这些知识预测事件的演变。

1.3.3　数据的类型

数据可以分为两个不同的类别：

- 类别型
  - 定类
  - 定序
- 数值型
  - 离散
  - 连续

*类别型数据*指可以被分成不同组或类别的值或观察结果。有两种类别型数据：*定类*（nominal）和*定序*（ordinal）。定类型变量的各类别没有内在的顺序，而定序型变量有预先指定的顺序。

*数值型数据*指通过测量得到的数值或观察结果。有两种不同的数值型数据：*离散型*和*连续型*。离散值的个数是可数的，每个值都与其他值区别开来。相反，连续值产生于结果属于某一确定范围的测量或观察。

1.4　数据分析过程

数据分析过程可以用以下几步来描述：转换和处理原始数据，以可视化方式呈现数据，建模做预测。因此，数据分析无外乎由几步组成，其中每一步所起的作用对后面几步而言都至关重要。因此数据分析几乎可以概括为由以下几个阶段组成的过程链：

- 问题定义

- 数据抽取
- 数据清洗
- 数据转换
- 数据探索
- 预测模型
- 模型评估/测试
- 结果可视化和阐释
- 解决方案部署

图1-1为数据分析各步骤的示意图。

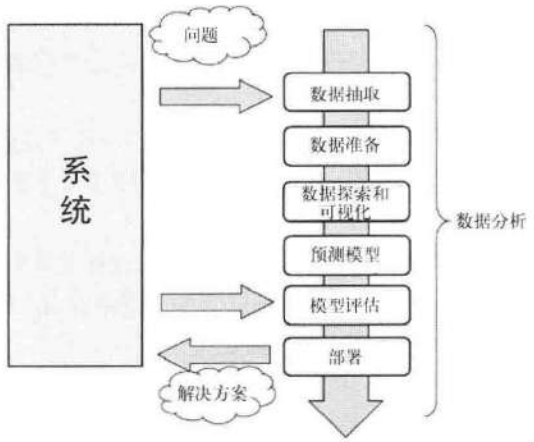


图1-1 数据分析过程

1.4.1 问题定义

采集原始数据前，数据分析过程实际上早已开始。事实上，数据分析总是始于要解决的问题，而这个问题需要事先定义。

只有深入探究作为研究对象的系统之后，才有可能准确定义问题：这个系统可能是一种机制、应用或是一般意义上的过程。一般而言，研究工作是为了更好地理解系统的运行方法，尤其是为了理解其运行规则，因为这些规则有助于我们作出预测或选择（在知情的基础上进行选择）。

问题定义这一步及产生的相关文档（可交付成果），无论是对于科研还是商业问题都很重要，因为这两项能严格保证分析过程是朝着目标结果前进的。事实上，对系统进行全面或详尽的研究有时会很复杂，一开始可能没有足够的信息。因此问题的定义，尤其是问题的规划，将唯一决定整个数据分析项目所遵循的指导方针。

定义好问题并形成文档后，接下来就可以进入数据分析的项目规划环节。该环节要弄清楚高效完成数据分析项目需要哪些专业人士和资源。因此就得考虑解决方案相关领域的一些事项。你

需要寻找各个领域的专家，安装数据分析软件。

因此，在项目规划过程中，应组建起高效的数据分析团队。一般而言，这个团队应该是跨学科的，因为从不同角度研究数据有助于解决问题。因此，一个优秀的团队必然是成功完成数据分析工作的关键因素之一。

1.4.2　数据抽取

问题定义步骤完成之后，在分析数据前，首先要做的就是获取数据。数据的选取一定要本着创建预测模型的目的，数据选取对数据分析的成功起着至关重要的作用。所采集的样本数据必须尽可能多地反映实际情况，也就是能够描述系统对来自现实刺激的反应。事实上，如果原始数据采集不当，即使数据量很大，这些数据描述的情境往往也是与现实相左或存在偏差。

因此，如果对选取不当的数据，或是对不能很好地代表系统的数据集进行数据分析，得到的模型将会偏离作为研究对象的系统。

数据的查找和检索往往要凭借一种直觉，超乎单纯的技术研究和数据抽取。它还要求对数据的内在特点和形式有细致入微的理解，而只有对问题的来源领域有丰富的经验和知识，才能做到这一点。

除了所需数据的质量和数量，另一个问题是查找和正确选择*数据源*（data source）。

如果工作室环境为（技术或科学）实验室，数据源生成的数据是用来做实验的。这种情况下就很容易鉴别数据源的优劣，这时唯一要注意的就是实验过程的设置。

无论是对于哪一个领域的应用，都不可能采用严格的实验方法来重建数据源所属的系统。很多领域的应用需要从周边环境搜寻数据，往往依赖于外部实验数据，甚至常通过采访或调查来收集数据。这种情况下，寻找包含数据分析所需全部信息的数据源难度很大。这时往往需要从多种数据源搜集信息，以弥补缺陷，识别矛盾之处，使数据集尽可能具有普遍性。

当你想找些数据来用时，Web是个不错的起点。但Web中的大多数数据获取起来具有一定难度。事实上，不是所有的数据都是以文件或是数据库形式存在的，有些数据以这样或那样的格式存在于HTML页面中；有的内容很明确，有的则不然。为了获取网页中的内容，人们研究出了Web抓取（Web scraping）方法，通过识别网页中特定的HTML标签采集数据。有些软件就是专门用来抓取网页的。它们找到符合条件的标签，从中抽取目标数据。查找、抽取完成后，就得到了用于数据分析的数据。

1.4.3　数据准备

在数据分析的所有步骤中，数据准备虽然看上去不太可能出问题，但事实上，这一步需要投入更多的资源和时间才能完成。数据往往来自不同的数据源，有着不同的表现形式和格式。因此，在分析数据之前，所有这些不同的数据都要处理成可用的形式。

数据准备阶段关注的是数据获取、清洗和规范化处理，以及把数据转换为优化过的，也就是准备好的形式，通常为表格形式，以便使用在规划阶段就定好的分析方法处理这些数据。

数据中存在的很多问题都必须解决掉，比如存在无效的、模棱两可的数据，值缺失，字段重复以及有些数据超出范围等。

1.4.4 数据探索和可视化

探索数据本质上是指从图形或统计数字中搜寻数据，以发现数据中的模式、联系和关系。数据可视化是突出显示可能的模式的最佳工具。

近年来，数据可视化发展迅猛，已成为一门真正的学科。事实上，专门用来呈现数据的技术有很多，从数据集中抽取最佳信息的可视化技术也不少。

数据探索包括初步检验数据，这对于理解采集到的数据的类型和含义很重要。再结合问题定义阶段所获得的信息，确定数据类型，这决定着选用哪种数据分析方法定义模型最合适。

一般来讲，在这个阶段，除了细致研究用数据可视化方法得到的图表外，可能还包括以下一种或多种活动：

- 总结数据
- 为数据分组
- 探索不同属性之间的关系
- 识别模式和趋势
- 建立回归模型
- 建立分类模型

通常来讲，数据分析需要总结与研究数据相关的各种表述。总结（summarization）过程，在不损失重要信息的情况下，将数据浓缩为对系统的解释。

聚类这种数据分析方法用来找出由共同的属性所组成的组（grouping，分组）。

数据分析的另外一个重要步骤关注的是识别（identification）数据中的关系、趋势和异常现象。为了找到这些信息，需要使用合适的工具，同时还要分析可视化后得到的图像。

其他数据挖掘方法，比如决策树和关联规则挖掘，则是自动从数据中抽取重要的事实或规则。这些方法可以和数据可视化配合使用，以便发现数据之间存在的各种关系。

1.4.5 预测模型

数据分析的预测模型阶段，则要创建或选择合适的统计模型来预测某一个结果的概率。

探索完数据后，你就掌握了用来开发数学模型，为数据中所存在的关系编码的全部信息。这些模型有助于我们理解作为研究对象的系统。具体来说，模型主要有以下两个方面的用途：一是预测系统所产生的数据的值，使用回归模型；二是为新数据分类，使用分类模型或聚类模型。事实上，根据输出结果的类型，模型可分为以下三种。

- 分类模型：模型输出结果为类别型。
- 回归模型：模型输出结果为数值型。
- 聚类模型：模型输出结果为描述型。

生成这些模型的简单方法包括线性回归、逻辑回归、分类、回归树和K-近邻算法。但是分析方法有多种，且每一种都有自己的特点，擅长处理和分析特定类型的数据。每一种方法都能生成一种特定的模型，选取哪种方法跟模型的自身特点有关。

有些模型输出的预测值与系统实际表现一致，这些模型的结构使得它们能够以一种简洁清晰的方式解释我们所研究的系统的某些特点。另外一些模型也能给出正确的预测值，但是它们的结构为“黑箱”，对系统特点的解释能力有限。

1.4.6　模型评估

模型评估阶段也就是测试阶段，对数据分析很重要。在该阶段，我们会验证用先前采集的数据创建的模型是否有效。该阶段之所以重要，是因为直接与真实系统数据比较，可评估模型所生成的数据的有效性。但其实该阶段我们是从整个数据分析过程所使用的初始数据集中取一部分用于验证。

一般来说，用于建模的数据称为训练集，用来验证模型的数据称为验证集。

通过比较模型和实际系统的输出结果，就能评估错误率。使用不同的测试集，就可以得出模型的有效性区间。事实上，预测结果只在一定范围内才有效，或因预测值取值范围而异，预测值和有效性之间存在不同层级的对应关系。

模型评估过程，不仅可以得到模型的确切有效程度（其形式为数值），还可以比较它跟其他模型有什么不同。模型评估技巧有不少，其中最著名的是交叉检验。它的基础操作是把训练集分成不同部分，每一部分轮流作为验证集，同时其余部分用作训练集。通过这种迭代的方式，可以得到最佳模型。

1.4.7　部署

数据分析的最后一步——部署，旨在展示结果，也就是给出数据分析的结论。若应用场景为商业，部署过程将分析结果转换为对购买数据分析服务的客户有益的方案。若应用场景为科技领域，则将成果转换为设计方案或科技出版物。也就是说，部署过程基本上就是把数据分析得到的结果应用到实践中去。

数据分析或挖掘的结果有多种部署方式。通常，数据分析师会在这个阶段为管理层或是客户撰写报告，从概念上描述数据分析结果。报告应上呈经理，以便他们读后好作出相应决策，真正用分析结果指导实践。

数据分析师提交的报告一般应该详细论述以下四点：

- ❑ 分析结果
- ❑ 决策部署
- ❑ 风险分析
- ❑ 商业影响评估

如果项目的产出包括生成预测模型，那么这些模型就可以以单独应用的形式进行部署或集成

到其他软件中。

1.5 定量和定性数据分析

数据分析过程都是以数据为中心，根据数据的特点，其实还可以对数据分析做进一步区分。

如果所分析的数据有着严格的数值型或类别型结构，这种分析称为定量分析；如果数据要用自然语言来描述，则称为定性分析。

由于所处理的对象具有不同的特点，这两种数据分析方法也有所不同。

定量分析所处理的数据具有内在逻辑顺序或者能分成不同的类别。这样，数据就有了不同的结构。顺序、类别和结构可以提供更多信息，从而可以以更加严格的数学形式对数据做进一步处理。用这种数据产生的模型能够作出定量预测，因此分析师也就可以得出更加客观的结论。

而定性分析处理的数据通常没有内在结构，至少结构没那么明显，这些数据既不是数值型也不是类别型。例如，适合定性分析研究的数据包括文本、视频和音频。分析这类数据时，往往需要根据实际情况，开发特殊方法来抽取信息。用这些信息创建的模型能够作出定性预测，而数据分析师给出的结论可能还包括主观解释。从另一方面来讲，定性分析可用来探索更加复杂的系统，而且它所能得到的结论，严格的数学方法可能无法给出。定性分析通常用来研究诸如社会现象或复杂结构等测量难度很大的系统。

图1-2展示了这两种分析方法的不同之处。

图1-2 定量分析和定性分析

1.6 开放数据

为了满足日益增长的数据需求，人们把很多数据资源放到了因特网上。这些被称为开放数据（Open Data）的数据资源对任何有数据需求的人免费开放。

下面是网上的一些开放数据资源站点。更完整、详细的开放数据资源请见附录B。

- DataHub网站（http://datahub.io/dataset）

- 世界卫生组织（http://www.who.int/research/en/）
- Data.gov网站（http://data.gov）
- 欧盟开放数据门户（http://open-data.europa.eu/en/data/）
- 亚马逊AWS开放数据集（http://aws.amazon.com/datasets）
- Facebook Graph（http://developers.facebook.com/docs/graph-api）
- Healthdata.gov网站（http://www.healthdata.gov）
- 谷歌趋势（http://www.google.com/trends/explore）
- 谷歌金融（https://www.google.com/finance）
- Google Books Ngrams项目（http://storage.googleapis.com/books/ngrams/books/datasetsv2.html）
- UCI机器学习数据库（http://archive.ics.uci.edu/ml/）

就开放数据而言，你可以通过LOD云图（http://lod-cloud.net）了解网上都有哪些开放数据资源可用。从云图中你能看到当前网上有哪些开放数据资源，以及这些资源之间的关系（见图1-3）。

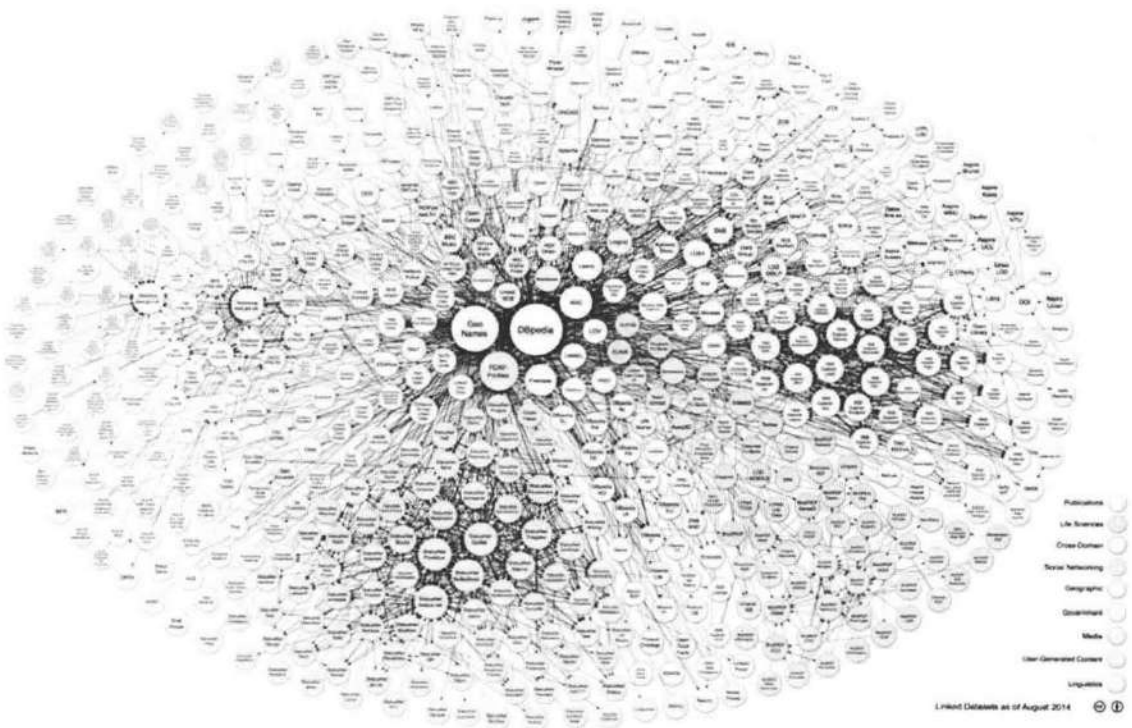


图1-3　相互联系的开放数据云图2014，设计者为Max Schmachtenberg、Christian Bizer、Anja Jentzsch和Richard Cyganiak。来自http://lod-cloud.net/[CC-BY-SA license]

1.7 Python 和数据分析

本书的主要特点是用Python语言介绍数据分析的所有概念。Python大量的库为数据分析和处理提供了完整的工具集，因此它被广泛应用于科学计算领域。

比起R和Matlab等其他主要用于数据分析的编程语言，Python不仅提供数据处理平台，而且还有其他语言和专业应用所没有的特点。Python库一直在增加，算法的实现采用更具创新性的方法，再加上它能跟很多语言（C和Fortran）相对接，这些特点都使得Python在所有可用于数据分析的语言中与众不同。

进一步来说，Python其实不是专用于数据分析的，它还有很多其他方面的用途。比如，它本身是一门通用型编程语言，也可以作脚本来用，还能操作数据库；而且由于Django等框架的问世，Python近些年还用来开发Web应用。因此，使用Python开发的数据分析项目，完全可以跟Web服务器相兼容，也就可以整合到Web应用中。

因此，对于想从事数据分析的读者，Python以及它众多的包，在可以预见的将来会是你的最佳选择。

1.8 结论

通过本章，你了解到了数据分析是怎么回事，更确切地说，它由哪些步骤组成。此外，你也许开始意识到数据在创建预测模型过程中所起的作用，以及甄选数据是数据分析结果准确可靠的基础。

下一章，我们将怀着对Python及其各种库的认可，开始数据分析之旅。

第2章 Python世界简介

Python语言及其周边世界由解释器、工具、编辑器、库和笔记本[1]等组成。近些年来，Python世界急速膨胀，日趋丰富，以至于第一次接触它的开发者有时会感觉它很复杂，有点不知所措。如果你是第一次接触Python编程，面对众多选择，可能会感到迷失了方向，不知从何处入手。

通过本章的学习，你将对Python世界有整体性认识。首先，你将了解到Python语言的基本情况以及它独有的特点。接着，你将会明白从何处入手，了解到什么是解释器，以及如何开始编写第一行Python代码。然后，你将学到如何用IPython和IPython Notebook等工具以更加新颖、高级的交互式形式编写程序。

2.1 Python——编程语言

Python于1991年由Guido van Rossum[2]发明，它是从ABC语言发展而来的，其特点可以概括为以下几个词：

- 解释型
- 可移植
- 面向对象
- 交互式
- 胶水（interfaced）
- 开源
- 便于理解和使用

Python是一种解释型语言，它采用的是伪编译方法。编写完程序后，要有解释器才能运行。要解释并运行源代码，机器上需要安装解释器程序。因而跟C、C++和Java等语言不同，Python程序不需要编译。

Python具有很高的可移植性。用解释器作为接口读取和运行代码的最大优势就是可移植性。事实上，任何现有系统（Linux、Windows和Mac）安装相应版本的解释器后，Python代码无需修改就能在其上运行。正是由于Python具有这个特点，包括树莓派和其他微处理器在内的小型设备

① 这里指IPython Notebook生成的文件。——译者注（后文若无特殊说明，脚注均为“译者注”。）

② Guido的名字经常被弄错，因此他特意在自我介绍中作了说明，详见https://www.python.org/~guido/。

才都把它选作编程语言。

Python属于面向对象的语言。你可以指定表示对象的类，实现继承关系。但是跟C++和Java的区别是，Python没有构造函数或析构函数。在Python中，你可以实现特定的结构来管理异常。然而，由于Python的语言结构很灵活，你可以用函数式编程、向量式编程等方法来实现面向对象方法所能达到的效果。

Python是一种交互式编程语言。由于Python用解释器执行代码，使用环境不同时，Python可呈现出极为不同的特点。事实上，我们可以像编写C++或Java那样，编写大量代码后再运行；或者你也可以输入一行命令后就执行，这样马上就能得到执行结果，然后根据返回结果再决定下一行代码写什么。执行代码的模式具有高度交互性，这使得Python成为像Matlab那样非常适合计算的语言。Python之所以在科学计算领域获得成功，跟这个特点密切相关。

Python可以用作胶水，粘合C/C++和Fortran等其他编程语言，这也是它的一大优点。事实上，Python可以以此弥补执行速度慢这个缺点——这可能是它唯一的缺点。作为一种动态性极高的编程语言，有时执行Python程序所用的时间是用其他语言编写、编译后的静态程序的100倍。因此要解决这类性能问题，可以在Python语言中无缝使用编译好的其他语言的代码。

Python是一门开源的编程语言。Python语言的参考实现CPython完全免费、开源。此外，每一个模块和库都是开源的，它们的代码可以从网上找到。每个月，庞大的开发者社区都会为Python带来很多改进，使Python的库更加丰富，提升它们的性能。CPython由成立于2001年的非盈利机构Python软件基金会管理。该基金会的宗旨是宣传、保护和推动Python编程语言的发展。

Python还是一门易于学习和使用的语言。这可能是Python最为重要的一个特点，因为这是开发者甚至是新手首先就能感受到的。Python代码很直观，读起来很容易，往往会引发用户的无限感慨[1]。久而久之，它就成为了大多数编程新手的首选。然而，简单并不代表它的应用范围窄，相反，Python被广泛用于各个计算领域。此外，比起C++、Java和Fortran等其他编程语言，Python处理起各种任务来更简单，远没有其他编程语言那么复杂。

2.2 Python——解释器

上一节讲过，每次运行python命令，Python解释器就会启动，你将会看到命令提示符>>>。

Python解释器程序无非是读取和解释输入到提示符后面的代码。前面已经提过，解释器既可以一次只接收单条命令，也可以接收整个Python代码文件。不管哪种情况，解释器的处理机制都相同。

每次按下回车键之后，解释器开始以单词为单位逐一（tokenization，单词化）扫描代码（一行或整个文件的所有代码）。这些单词实则是一个个文本片段，解释器把它们组织成为表示程序逻辑结构的树状结构，随后这些代码片段将会被转化为字节码（.pyc或.pyo）。生成的字节码随后将交由Python虚拟机（PVM）执行。解释过程到此结束，请见图2-1。

① 比如“life is short, you need Python”，出处http://sebsauvage.net/python/。

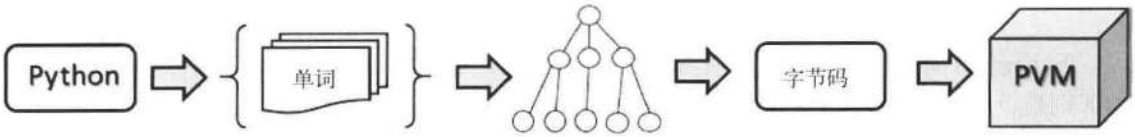


图2-1　Python解释器解释代码的过程

关于这个主题，有一份文档讲得非常好，详见https://www.ics.uci.edu/~pattis/ICS-31/lectures/tokens.pdf。

Python的标准解释器称作Cython，因为它是完全用C语言编写的。除此之外，还有一些用其他语言编写的解释器，比如用Java开发的Jython、用C#开发的（因此只适用于Windows系统）IronPython以及全部用Python开发的PyPy。

2.2.1　Cython

Cython项目以开发能把Python代码转换为等价C代码的编译器为基础。C代码随后在Cython环境中执行。这种编译机制使得在Python代码中嵌入能够提升效率的C代码成为可能。Cython可以被视为一门新的编程语言，它的发明实现了两种编程语言的融合。可以从网上找到大量相关文档。建议你访问链接http://docs.cython.org。

2.2.2　Jython

跟Cython相对应的，还有完全用Java语言开发和编译的Jython。它是由Jim Hugunin 在1997年开发的（http://www.jython.org）。Jython是用Java语言实现的Python编程语言。更进一步来讲，它具有以下特点：Python的扩展和包是用Java类而不是Python模块实现的。

2.2.3　PyPy

PyPy解释器是一种即时（just-in-time，JIT）编译器，它在运行时直接把Python代码转化为机器码。这样做是为了提升代码的执行速度，却因此只使用了所有Python命令中的很少一部分。这个只包含少数Python命令的子集被称作RPython。关于PyPy的更多信息，请访问其官网http://pypy.org。

2.3　Python 2 和 Python 3

目前，Python社区仍处于从系列2解释器到系列3解释器的过渡阶段。你会发现当前这两个版本（2.7版本和3.4版本）都有人在用。两种版本并存的局面，为用户平添几分疑惑，尤其是不知道应该选用哪个版本，或是弄不清楚两者之间到底有什么差异。你一定会问的一个问题是，既然3.x系列更为高级，那为什么2.x系列仍没有停止发版。

当Guido van Rossum（Python之父）决定对Python语言作出重大改进时，他很快就发现这些

改动会使新的Python语言与很多现有代码不兼容。因此，他最终决定创建Python新版本——Python 3.0。为了解决兼容性问题，保证已有代码的顺利运行，他还决定维护跟历史代码相兼容的版本，确切地说，就是2.7版本。

Python 3.0版本于2008年发布，而2.7版本则于2010年发布。Python官方决定在2.7版本之后不再发布大的版本。写作本书时（2014年），Python 3.x系列最新的版本号为3.4。

本书中，我们将使用Python 2.x系列。但是，除去极少数情况，使用3.x系列应该也不会有问题。

2.4 安装 Python

要使用Python开发程序，需要在操作系统中安装它。与Windows不同的是，Linux和Mac OS X系统应该预装了某个版本的Python。如果没有或是你想安装新版本，方法也很简单。虽然Python安装方法因操作系统而异，但是操作起来都很容易。

Debian-Ubuntu Linux系统使用以下命令：

```
apt-get install python
```

支持rpm包的Red Hat 和 Fedora Linux系统则使用以下命令：

```
yum install python
```

如果你用的是Windows或Mac OS X操作系统，可以从Python官网（http://www.python.org)下载喜欢的版本，自动进行安装。

除了上述方法，如今有很多发行版不仅提供Python解释器，还提供很多工具，这些工具简化了Python、所有的库以及相关应用的管理和安装工作。我强烈建议你从网上找一个现成的发行版来用。

2.5 Python 发行版

由于Python语言获得了成功，应用面很广，开发者需要的功能也多种多样。为了满足这些需求，人们开发了大量的包。这么多年下来，包的数量已经多到几乎不可能靠人工来管理了。

鉴于此，很多Python发行版应运而生，它们能够高效地管理成百上千的Python包。事实上，比起单独下载、安装只包含标准库的解释器，然后用到时再逐个安装所需要的库，直接安装Python发行版更加简单。

这些发行版的核心是包管理器，该应用也只不过是能够自动管理、安装、更新、配置和删除作为发行版一部分的Python包。

包管理器非常有用。用户请求所需要的包时（比如要安装一个包），包管理器通常通过因特网分析用户所要安装的包的版本号以及它所依赖的包，如果依赖包不存在的话，会顺便下载。

2.5.1 Anaconda

Anaconda是Continuum Analytics公司（https://store.continuum.io/cshop/anaconda/）开发的免费

的Python包发行版，支持Linux、Windows和Mac OS X操作系统。它不仅提供Python的最新包，还提供搭建Python开发环境所需的大多数工具。

在系统中安装Anaconda后，就可以使用本章所提到的大多数工具和应用，而无需分别安装和管理它们。它所包含的工具有Spyder IDE、IPython QtConsole和IPython Notebook。

整个Anaconda发行版用conda管理所有的包，维护其版本信息，它是Anaconda的包管理器和环境管理器。

```
conda install <package name>
```

该发行版最为有趣的一个特点是，它能够管理Python版本的多种开发环境。安装Anaconda后，默认安装的是Python 2.7版本以及为该版本开发的包。但这没关系，因为通过创建独立于2.7版本的新环境，Anaconda允许开发者同时使用Python的其他版本。例如，你可以创建基于Python 3.4的新环境。

```
conda create -n py34 python=3.4 anaconda
```

以上命令将会创建一个新的Anaconda环境，里面安装的包都指向Python 3.4。在新环境安装所需的包，不会对Python 2.7 环境造成任何影响。创建新环境后，输入下面命令即可激活：

```
source activate py34
```

Windows用户需要输入以下命令：

```
activate py34
C:\Users\Fabio>activate py34
Activating environment "py34"...
[py34] C:\Users\Fabio>
```

你可以创建使用不同Python版本的环境，只需要修改conda create命令中python选项的值即可。如要使用默认的Python版本，输入以下命令即可：

```
source deactivate
```

Windows用户需要输入以下命令：

```
[py34] C:\Users\Fabio>deactivate
Deactivating environment "py34"
C:\Users\Fabio>
```

2.5.2 Enthought Canopy

Enthought公司提供的Canopy发行版跟Anaconda很相似。Enthought公司创立于2001年，其最为知名的是SciPy项目（https://www.enthought.com/products/canopy/）。Canopy发行版支持Linux、Windows和Mac OS X系统，包含大量包、工具和应用，用包管理器进行管理。与conda不同的是，包管理器Canopy完全是图形化的。

遗憾的是，这个发型版只有基础版Canopy Express免费。它除了提供各发行版通常都会提供的包之外，还内置了IPython和Canopy IDE，后者具有其他IDE所没有的特殊功能。嵌入IPython是为了把它的环境用作测试和调试代码的窗口。

2.5.3 Python(x,y)

Python(x,y)是只支持Windows系统的一种免费发行版，下载地址为http://code.google.com/p/pythonxy/。它使用Spyder作为IDE。

2.6 使用 Python

Python语言包容万象，却又不失简洁，用起来还很灵活；不论用它从事哪个领域的开发（数据分析、科学计算和图形界面等），扩展起来都很容易。也正是出于这个原因，Python的用法多种多样，具体怎么用取决于开发者的喜好和能力。这一节，我们介绍本书中所用到的Python的各种用法。各章讨论的主题有所不同，因此所用到的Python方法也会存在差异，我们的原则是为不同的任务选用最合适的方法。

2.6.1 Python shell

走进Python世界最简单的方式莫过于通过Python shell（运行命令行的终端界面）创建一段会话（session）。然后，你就可以输入一条命令，立即测试它是否能正常运行。这种模式阐明了解释器的特性，Python代码所要执行的操作由解释器来决定。解释器能够一次读取一条命令，同时保持先前命令所指定的变量的状态，这一点跟Matlab和其他计算软件相似。

这种模式非常适合第一次接触Python语言的新手。你可以逐条测试命令，无需事先编写、编辑好再来运行可能包含多行代码的完整程序。

这种模式也表明可以逐行对代码进行测试、调试或用来处理计算任务。在终端开启会话模式很简单，只需输入以下命令即可：

```
>>> python
Python 2.7.8 (default, Jul 2 2014, 15:12:11) [MSC v.1500 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

现在，Python shell已经激活，解释器严阵以待，等待接收Python命令。我们来编写最简单也是编程初学者都要编写的经典例子：

```
>>> print "Hello World!"
Hello World!
```

2.6.2 运行完整的 Python 程序

每位程序员最熟悉的方式是编写完整个程序后，再在终端运行。首先使用简单的文本编辑器编写程序。可以使用代码清单2-1作为例子，将它保存为MyFirstProgram.py。

代码清单2-1 MyFirstProgram.py

```
myname = raw_input("What is your name? ")
```

```
print "Hi " + myname + ", I'm glad to say: Hello world!"
```

现在你已编写好自己的第一个Python程序了，可以直接在命令行使用python命令运行它。注意，将包含程序代码的文件名放到python命令的后面。

```
python myFirstProgram.py
What is your name? Fabio Nelli
Hi Fabio Nelli, I'm glad to say: Hello world!
```

2.6.3　使用 IDE 编写代码

比前面更为复杂的方法是使用IDE（Integrated Development Environment，集成开发环境）编写并运行代码。这些编辑器相当复杂，提供了Python开发所需的工作环境。它们提供的多种工具为开发者带来很多便利，尤其非常有助于调试程序。接下来几节将详细介绍几款当前主流的IDE。

2.6.4　跟 Python 交互

即将介绍的最后一种方法——交互式编程，在我看来可能是最有创新性的。前三种方法不论好坏，使用其他语言的开发者都在使用。最后一种方法提供了直接与Python代码交互的机会。

从这个方面讲，IPython的发明极大地丰富了Python世界。功能强大的IPython旨在满足分析师、工程师或研究员等类型的开发者和Python解释器进行交互的需求。稍后会有一节更为详细地介绍IPython以及它的主要特点。

2.7　编写 Python 代码

上节，我们讲了如何编写简单的小程序输出字符串“Hello World”。接下来这一节，我们将从总体上介绍Python语言基础，帮助你熟悉最为重要的基础知识。

本节的目的不是教你如何用Python编写程序，或是讲解Python语言的句法规则，而是让你快速地对Python的基本规则有个总体印象，这样才能继续学习后面的各个主题。

如果你已熟悉Python语言，尽管大胆地跳过Python介绍这一部分内容。相反，如果你不熟悉编程，觉得所讲的概念很难理解，我强烈建议你从网上找找相关文档、教程或课程，自己多补补。

2.7.1　数学运算

前面我们见过print()函数几乎可以输出任何内容。其实，Python不仅是输出工具，还是强大的计算器。在命令行开启一段会话，进行下面的数学运算：

```
>>> 1 + 2
3
>>> (1.045 * 3)/4
0.78375
>>> 4 ** 2
16
```

```
>>> ((4 + 5j) * (2 + 3j))
(-7+22j)
>>> 4 < (2*3)
True
```

Python能够对多种类型的数据进行计算，包括复数和含有布尔值的条件表达式。从上面的计算可以看出，Python解释器直接返回计算结果而不需要使用print()函数输出结果，对于存放在变量中的值也是如此。调用变量，就能看到它里面的内容。

```
>>> a = 12 * 3.4
>>> a
40.8
```

2.7.2 导入新的库和函数

前面讲过，Python的一大特点是通过导入各种现成的包和模块来扩展其功能。导入整个包，需要使用import命令：

```
>>> import math
```

这样，math模块中的所有函数都可以在当前会话中使用，因此你可以直接调用它们。新会话所能使用的函数的标准集也得到了扩展。这些函数的调用方式如下：

```
library_name.function_name()
```

例如，你现在就可以计算变量a所存放的数值的正弦值：

```
>>> math.sin(a)
```

如上所见，调用函数时需带着库的名字。有时你可能会遇到下面这种形式的导入语句：

```
>>> from math import *
```

即使这样做没有问题，也应该避免。事实上，这种导入语句把库中的所有函数都导入进来，在使用时，不用指定库的名称。

```
>>> sin(a)
0.040693257349864856
```

但这种导入方法实际上会带来非常严重的问题，尤其是在导入的库越来越多时。因为分属于不同库的函数可能存在重名的情况，所以如果把它们都导入进来，后导入的函数将覆盖掉先前导入的同名函数。因此程序可能会产生各种错误，甚至出现反常行为。

事实上，这种导入方法一般只用于以下情况：函数数量非常有限，且程序的正常运行又离不开这些函数，同时又完全没有必要导入整个库。

```
>>> from math import sin
```

数据结构

前面的例子中，我们曾用变量存储一个元素。实际上，Python提供了多种极其有用的数据结构，它们能够同时存储多个元素，有时甚至是不同类型的元素。这些数据结构的定义方法因它们

内部所存储的数据的结构而异。

- 列表
- 集合
- 字符串
- 元组
- 字典
- 双队列（deque）
- 堆

这只是可以用Python创建的数据结构中的一小部分。这些数据结构里最常用的是字典和列表。

字典（dictionary）这种数据结构，有时也被称作dict，其中每个元素（值）都有一个与之相关联的被称作键（key）的标签。字典中的数据没有内在顺序，而只是一个个键值对。

```
>>> dict = {'name':'William', 'age':25, 'city':'London'}
```

如果想获取字典里某一特定的值，需要指定它所对应的键的名称。

```
>>> dict["name"]
'William'
```

如果想迭代输出字典里的所有键值对，需要使用for-in结构，还要用到items()函数。

```
>>> for key, value in dict.items():
...     print(key,value)
...
name William
city London
age 25
```

列表（list）这种数据结构包含一系列具有明确顺序的元素，这些元素组成一个序列。它支持新增或删除元素操作。每个元素都有一个叫作索引（index）的数字标识，这个数字也就是该元素在序列中的位次。

```
>>> list = [1,2,3,4]
>>> list
[1, 2, 3, 4]
```

如果想获取单个元素，用方括号指定元素的索引即可（列表第一个元素的索引值为0）；如果想获取列表（或序列）的一部分，用索引i和j[1]指定所需范围的上下界即可。

```
>>> list[2]
3
>>> list[1:3]
[2, 3]
```

用负数作为索引，表示你想从列表的最后一个元素开始，朝第一个元素的方向获取元素。

```
>>> list[-1]
4
```

① 简单一提，以上界j为索引的元素，在进行list[i:j]这种切片操作时是取不到的。

如要扫描列表的每个元素，可使用for-in结构。

```
>>> items = [1,2,3,4,5]
>>> for item in items:
...         item + 1
...
2
3
4
5
6
```

2.7.3 函数式编程

前面例子的for-in结构跟其他编程语言中的非常相似。但实际上，如果你想成为一名真正的Python程序员，就应该避免使用显式循环。Python提供了几种替代方法，指定了诸如*函数式编程*（functional programming，亦即expression-oriented programming，面向表达式的编程）等编程技巧。

Python提供的用于函数式编程开发的函数有：

- map(function, list)，映射函数
- filter(function, list)，过滤函数
- reduce(function, list)，规约函数
- lambda函数
- 列表生成式

前面刚刚讲过的for循环对每个元素执行某一操作，然后把结果汇集起来。其实同样功能可以用map()函数来实现。

```
>>> items = [1,2,3,4,5]
>>> def inc(x): return x+1
...
>>> list(map(inc,items))
[2, 3, 4, 5, 6]
```

上述例子中，我们首先定义了对每一个元素进行操作的函数，随后把这个函数作为map()函数的第一个参数传递进来。Python允许你使用lambda函数直接在第一个参数中定义函数。这样能大幅精简代码，前面的代码结构就可以被浓缩为一行代码。

```
>>> list(map((lambda x: x+1),items))
[2, 3, 4, 5, 6]
```

其他两个函数filter()和reduce()的工作原理与之类似。filter()函数只抽取函数返回结果为True的列表元素。reduce()函数对列表所有元素依次计算后返回唯一结果。使用reduce()前，需要导入functools模块。

```
>>> list(filter((lambda x: x < 4), items))
[1, 2, 3]
>>> from functools import reduce
>>> reduce((lambda x,y: x/y), items)
```

```
0.008333333333333333
```

这两个函数实现了用for循环所能实现的功能。它们取代了这些循环结构及其功能，因为这两者可以表述为简单的函数调用，而函数式编程正是由这样的函数组成的。

函数式编程的最后一个概念叫作列表生成式（list comprehension）。这个概念可用来以非常自然和简单的方式创建列表，而这种列表创建方式跟数学家描述数据集所使用的类似。列表这个序列所包含的元素由特定的函数或运算来指定。

```
>>> S = [x**2 for x in range(5)]
>>> S
[0, 1, 4, 9, 16]
```

2.7.4　缩进

对那些具有其他编程语言背景的人来说，Python中缩进（indentation）所起的作用很奇特。你可能习惯了为了美观和增强代码的可读性而调整缩进，但是对Python而言，缩进是代码实现的一部分，它把代码分为一个个逻辑块。事实上，在Java、C和C++中，每行代码用英文的分号";"跟下一行代码区分开；而在Python中，你不能使用包括标识逻辑块的大括号在内的任何分隔符[1]。其他语言中的分隔符所扮演的角色，在Python中由缩进来扮演。也就是说，解释器根据每行代码的起始位置来决定它是否属于某个逻辑块。

```
>>> a = 4
>>> if a > 3:
...     if a < 5:
...         print("I'm four")
... else:
...     print("I'm a little number")
...
I'm four

>>> if a > 3:
...     if a < 5:
...         print("I'm four")
...     else:
...         print("I'm a big number")
...
I'm four
```

从这个例子可以看到，由于两段代码中else命令使用的缩进不同，其所表示的条件的含义（请见输出的两个字符串[2]）也不同。

① Python命令结尾可以用英文分号，但是没有必要。

② 指的是"I'm a little number"和"I'm a big number"。

2.8 IPython

IPython是在Python的基础上进一步开发的，增加了多种工具：IPython shell，性能有了极大提升的Python终端，功能强大的交互式shell；QtConsole，shell和GUI的混合体，实现在控制台而不是单独的窗口中显示图像；IPython Notebook，集文本、可执行代码、图像和公式的展现于一体的Web界面。

2.8.1 IPython shell

IPython shell看上去像是从命令行运行的Python会话，但实际上它提供了很多其他功能。它比Python自带的shell更加强大，功能更多。在命令行输入ipython命令，即可启动IPython shell。

```
> ipython
Python 2.7.8 (default, Jul 2 2014, 15:12:11) [M
Type "copyright", "credits", or "license" for more information.

IPython 2.4.1 -- 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.

In [1]:
```

如上所示，命令提示符为In [1]这种特殊形式，表示这是第一行输入。IPython命令提示符，无论是输入还是输出缓存都带有编号（索引）。

```
In [1]: print "Hello World!"
Hello World!

In [2]: 3/2
Out[2]: 1

In [3]: 5.0/2
Out[3]: 2.5

In [4]:
```

如上所述，表示输出的提示符也有编号，用Out[1]、Out[2]这类值来表示。IPython把所有输入都存储到变量中。事实上，所有输入都存储在叫作In的列表中。

```
In [4]: In
Out[4]: ['', u'print "Hello World!"', u'3/2', u'5.0/2', u'_i2', u'In']
```

In列表中每个元素的索引恰好是（其所对应的命令的）命令提示符中的数字。因此，指定一个数值，即可获取到先前输入的那一行代码。

```
In [5]: In[3]
Out[5]: u'5.0/2'
```

对于输出，也是如此。

```
{2: 1,
 3: 2.5,
 4: ['',
  u'print "Hello World!"',
  u'3/2',
  u'5.0/2',
  u'_i2',
  u'In',
  u'In[3]',
  u'Out'],
 5: u'5.0/2'}
```

2.8.2　IPython Qt-Console

要从命令行启动这个应用，必须使用以下命令：

```
ipython qtconsole
```

该应用包含一个GUI界面，它囊括了IPython shell的所有功能，请见图2-2。

图2-2　IPython QtConsole

1. IPython Notebook

IPython Notebook是交互式环境IPython的新生代力量（见图2-3）。有了IPython Notebook，可执行代码、文本、公式、图像和动画等内容都能整合到Web文档中，其用途很多，比如可用来做演示、制作教程或是辅助调试程序等。

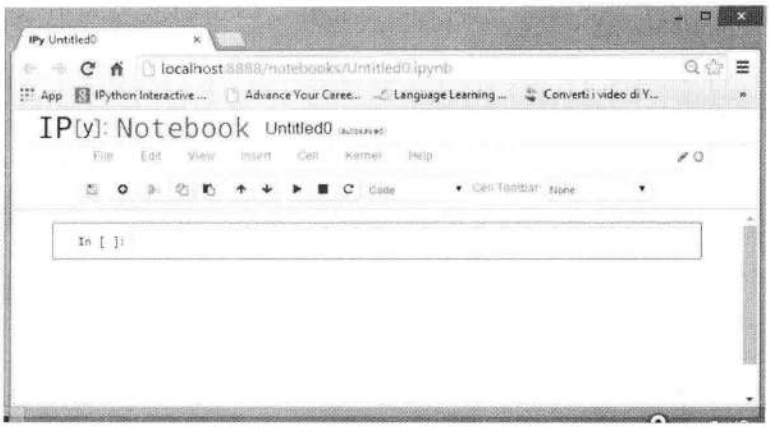


图2-3 IPython Notebook的Web界面

2. Jupyter项目

近来，IPython项目有了长足的发展。自IPython 3.0发布以来，该项目开始迁往新项目Jupyter（https://jupyter.org）。

IPython仍将继续作为Python shell而存在，但IPython项目的Notebook和其他与语言无关的组件将会被迁移，以组建新项目Jupyter，见图2-4。

图2-4 Jupyter项目的图标

2.9 PyPI 仓库——Python 包索引

Python包索引（Python Package Index，PyPI）软件仓库包含Python编程可能会用到的所有软件，例如属于其他Python库的软件。软件仓库直接由各个包的开发者管理，一旦他们的库发布新版本，他们负责将其更新到仓库中。如果你想了解PyPI仓库都有哪些包，请访问PyPI官网https://pypi.python.org/pypi。

至于如何管理这些包，你可以使用PyPI的包管理器pip应用。

从命令行启动pip应用，就可以对单个包进行安装、更新或删除操作。pip会检查这个包是否已安装；如已安装，则检查是否需要更新。同时，它还会检查是否需要安装其他依赖包；如未安装，pip就会下载和安装这个包以及它的依赖包。

```
$ pip install <<package_name>>
$ pip search <<package_name>>
$ pip show <<package_name>>
$ pip unistall <<package_name>>
```

至于如何安装pip，如果你的系统已安装了Python 3.4+（2014年3月发布）或Python 2.7.9（2014年12月发布），pip也随之安装了。但是，如果你使用的是Python的旧版本，则需自行安装，具体方法因操作系统而异。

Linux系统Debian和Ubuntu：

```
$ sudo apt-get install python-pip
```

Linux系统Fedora：

```
$ sudo yum install python-pip
```

Windows系统：

请访问www.pip-installer.org/en/latest/installing.html，下载get-pip.py到你的计算机上。下载完成后，运行如下命令。

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

这样，就能安装好包管理器。记得把C:\Python2.X\Scripts添加到环境变量PATH中去。

2.10　多种 Python IDE

虽然大多数Python开发者习惯了直接在shell（Python或IPython）中编写代码，但除此之外，还可以使用IDE（Interactive Development Environment，交互式开发环境）。IDE除了具有基本的文本编辑功能之外，还提供一系列辅助代码编写和调试的工具。例如，代码自动补全功能、查看命令的相关文档、调试和插入断点等。当然，IDE 提供的工具比这要丰富得多。

2.10.1　IDLE

IDLE（Integrated Development Environment[①]，集成开发环境）是专门为Python开发而编写的IDE。它是Python标准版自带的官方IDE，因此被嵌入到Python标准发行版中（见图2-5）。IDLE软件完全用Python实现。

① 据Python官网介绍，IDLE指的是“Integrated Development and Learning Environment”，详见https://docs.python.org/2/library/idle.html。据维基百科来看，Python之父表示，“IDLE”指的正是本书作者在这里使用的“Integrated Development Environment”，详见https://en.wikipedia.org/wiki/IDLE_(Python)。之所以使用“IDLE”，是为了纪念Monty Python剧团的创始人Eric Idle。最后一点笔者找Guido确认过。

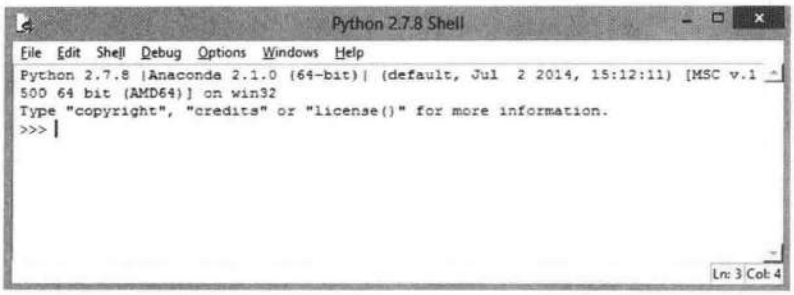


图2-5 IDLE Python shell

2.10.2 Spyder

Spyder（Scientific Python Development Environment，Python科学计算开发环境）IDE跟Matlab IDE有很多相似之处（见图2-6）。它在文本编辑器的基础上，添加了句法高亮和代码分析工具。此外，该IDE可在图像应用中添加立刻可用的控件。

图2-6 Spyder IDE

2.10.3 Eclipse（pyDev）

使用其他编程语言的开发者一定知道Eclipse，它是完全用Java（如要使用，计算机需要安装Java）开发的通用IDE，提供了适用于多种语言的开发环境（见图2-7）。Eclipse有一个版本是专

门用于Python开发的，但是要安装pyDev插件。

图2-7　Eclipse IDE

2.10.4　Sublime

该文本编辑器是Python程序员最喜欢的开发环境之一（见图2-8）。Sublime有多种插件，借助这些插件，用Sublime编写Python程序变得更简单，过程也很享受。

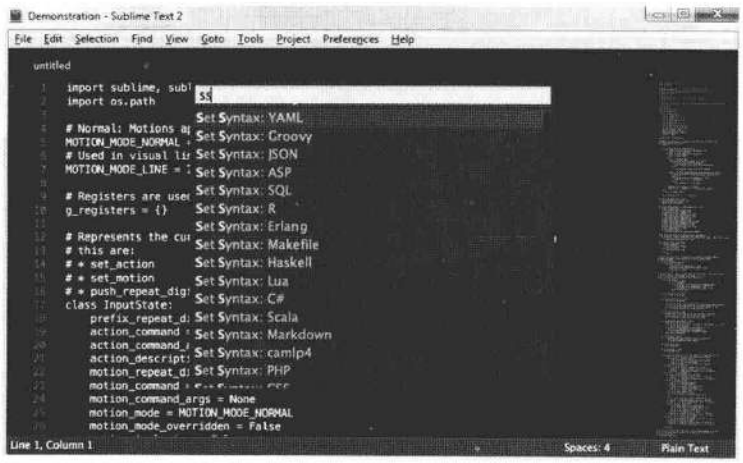


图2-8　Sublime IDE

2.10.5 Liclipse

与Spyder类似，该环境也是专门用于Python开发的（见图2-9）。它与Eclipse IDE十分相似，但是它为Python语言做了充分的适配，因此无需安装pyDev等插件就能使用Python。它的安装和配置比起Eclipse更加简单。

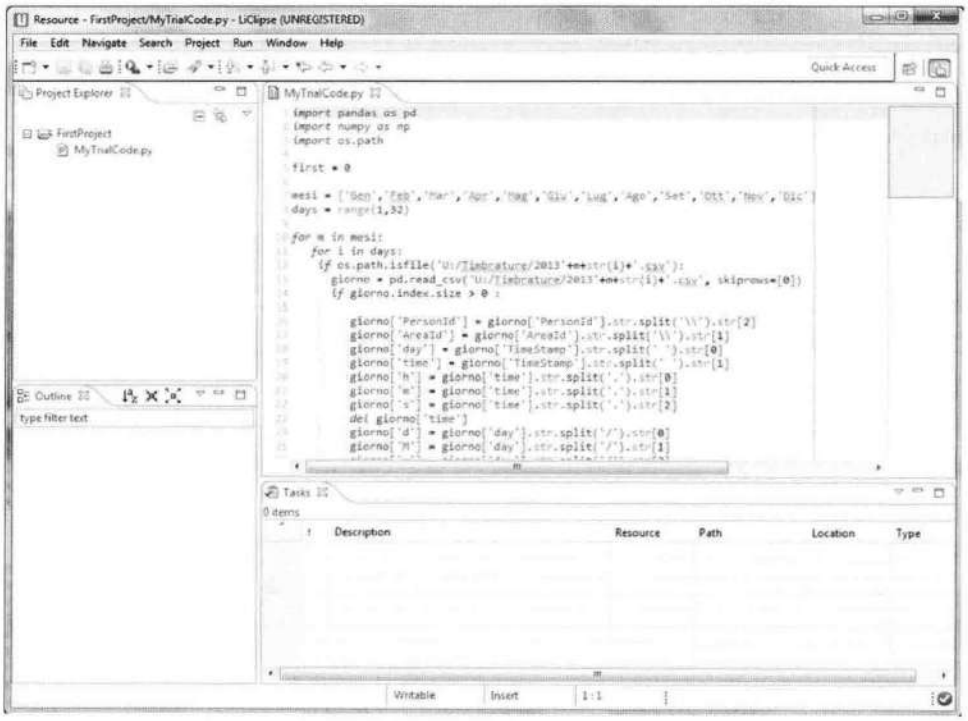


图2-9 Liclipse IDE

2.10.6 NinjaIDE

NinjaIDE[①]（NinjaIDE is "Not Just Another IDE"，NinjaIDE不只是另一个IDE）为首字母缩略词，该词运用了递归方法。这是一款专门用于Python开发的IDE，最近才开发出来。它凝集了很多开发人员的心血，现在看来已是前途无量，很可能会在接下来几年给开发者带来更多惊喜。

2.10.7 Komodo IDE

Komodo IDE功能强大，提供了多种工具，是完备、专业的开发环境。它是用C++开发的付费软件，提供适用于包括Python在内的多种编程语言的开发环境。

① 造词方法跟GNU类似，GNU指的是"GNU is not Unix"。

2.11　SciPy

SciPy（音同“Sigh Pie”）是一组专门用于科学计算的开源Python库。它里面的不少库将是本书很多章节的主角，因为掌握这些库对数据分析很重要。由这些库组成的工具集擅长处理数据计算和可视化，因此用Python做数据分析时，丝毫不用羡慕其他数据计算和分析环境（比如R或Matlab）。其中，下面几个库后续章节会着重讲解：

- NumPy
- matplotlib
- pandas

2.11.1　NumPy

NumPy库其名称的含义是“数值Python”（Numerical Python），很多由它发展而来的Python库都以其为核心。NumPy是用Python进行科学计算的一个基础库，因为它提供了Python基础包所没有提供的数据结构和高性能函数。事实上，正如本书后面将要讲到的，NumPy定义了一种专门用于科学计算的数据结构ndarray——它是一种*N*维数组。

正确使用这个库，能极大提升计算效率，因此在数值运算过程中掌握如何使用该库很重要。由于它具有独一无二的特性，在本书中几乎随处可见，因此很有必要用一章的篇幅（第3章）来介绍它。

NumPy的如下功能将会被添加到Python标准发行版中。

- ndarray：多维数组，比Python基础包提供的速度更快、效率更高。
- 元素级计算（element-wise computation）：一组用于数组或数组之间按照元素级运算的函数。
- 读-写数据集：一组从硬盘中读取数据或往硬盘写入数据的函数。
- 整合C、C++和Fortran等编程语言：整合其他语言编写的代码的工具集。

2.11.2　pandas

该包提供了复杂的数据结构和函数，其目的是降低处理难度，提升速度和效率。它是Python数据分析的核心包。因此，对该包的研究和应用将作为主题贯穿全书（第4、5和6章着重讲解）。详细讲解pandas的各方面知识，尤其是它在数据分析场景中的应用，是本书的主要目标。

该包最为基础的概念为数据框（DataFrame）。它是一个两维表格状数据结构，行和列均有标签。

pandas整合了NumPy库的高性能特性，可处理电子表格或关系型数据库（SQL数据库）中的数据。借助pandas强大的索引方法，对该类数据结构进行变形、切片、聚合和选取子集等操作比较容易。

2.11.3 matplotlib

这个包是目前绘制2D图像最常用的Python包。数据分析少不了可视化工具，而这个包最适合。第7章将详细讲解它的用法，学完之后，你就会知道怎样以最佳方式展现数据分析结果。

2.12 小结

这一章讲述了Python的主要基础内容。我们通过简洁的例子介绍了Python的基础概念，解释了它所引入的新特点，尤其是那些比其他语言更为出色的特点。此外还展示了它的不同使用方法。首先讲解了简单的命令行解释器的使用方法；然后介绍了一系列简单的图形用户界面；最后引入了IDE这类复杂的开发环境，比如Spyder和NinjaIDE。

我们甚至还介绍了极具创新意义的IPython项目，展示了以交互式方式（尤其是IPython Notebook）编写代码的可能性。

Python通过第三方库来扩展标准函数集的功能体现了它的模块化特性，就这一点我们介绍了PyPI在线仓库以及Python的其他发行版，比如Anaconda和Enthought Canopy。

下一章，我们将学习作为Python数值计算基础的第一个库：NumPy。还会讲解ndarray这种数据结构，后续章节中数据分析所使用的更为复杂的数据结构都以它为基础。

第3章

NumPy库

3

NumPy是用Python进行科学计算，尤其是数据分析时，所用到的一个基础库。它是大量Python数学和科学计算包的基础，比如后面要讲到的pandas库就用到了NumPy。pandas库专门用于数据分析，充分借鉴了Python标准库NumPy的相关概念。而Python标准库所提供的内置工具对数据分析方面的大多数计算来说都过于简单或不够用。

为了更好地理解和使用Python所有的科学计算包，尤其是pandas，需要先行掌握NumPy库的用法，这样才能把pandas的用处发挥到极致。pandas是后续章节的主题。

如果你已熟悉NumPy库，可跳过本章直接学习下一章；否则，你可以借此机会复习一下NumPy的基础概念，或者敲敲本章的示例代码，争取尽快使自己再次熟悉起来。

3.1 NumPy 简史

Python语言诞生不久，开发人员就产生了数值计算的需求，更为重要的是，科学社区开始考虑用它进行科学计算。

1995年，Jim Hugunin开发了Numeric，这是第一次尝试用Python进行科学计算。随后又诞生了Numarray包。这两个包都是专门用于数组计算的，但各有各的优势，开发人员只好根据不同的使用场景，从中选择效率更高的包。由于两者之间的区别并不那么明确，开发人员产生了把它们整合为一个包的想法。Travis Oliphant遂着手开发NumPy库，并于2006年发布了它的第一个版本（v 1.0）。

从此之后，NumPy成为Python科学计算的扩展包。如今，在计算多维数组和大型数组方面，它是使用最广的。此外，它还提供多个函数，操作起数组来效率很高，还可用来实现高级数学运算。

当前，NumPy是开源项目，使用BSD许可证。在众多开发者的支持下，这个库的潜力得到了进一步挖掘。

3.2 NumPy 安装

通常，大多数Python发行版都把NumPy作为一个基础包。然而，如果NumPy不是基础包的话，你可以自行安装。

Linux系统（Ubuntu和Debian）：

```
sudo apt-get install python-numpy
```

Linux系统（Fedora）：

```
sudo yum install numpy scipy
```

使用Anaconda发行版的Windows系统：

```
conda install numpy
```

NumPy安装到系统之后，在Python会话中输入以下代码导入它NumPy模块：

```
>>> import numpy as np
```


3.3 ndarray：NumPy 库的心脏

整个NumPy库的基础是ndarray（*N*-dimensional array，*N*维数组）对象。它是一种由同质元素组成的多维数组，元素数量是事先指定好的。同质指的是几乎所有元素的类型和大小都相同。事实上，数据类型由另外一个叫作dtype（data-type，数据类型）的NumPy对象来指定；每个ndarray只有一种dtype类型。

数组的维数和元素数量由数组的型（shape）来确定，数组的型由*N*个正整数组成的元组来指定，元组的每个元素对应每一维的大小。数组的维统称为轴（axes），轴的数量被称作秩（rank）。

NumPy数组的另一个特点是大小固定，也就是说，创建数组时一旦指定好大小，就不会再发生改变。这与Python的列表有所不同，列表的大小是可以改变的。

定义ndarray最简单的方式是使用array()函数，以Python列表作为参数，列表的元素即是ndarray的元素。

```
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
```

检测新创建的对象是否是ndarray很简单，只需要把新声明的变量传递给type()函数即可。

```
>>> type(a)
<type 'numpy.ndarray'>
```

调用变量的dtype属性，即可获知新建的ndarray属于哪种数据类型。

```
>>> a.dtype
dtype('int32')
```

我们刚建的这个数组只有一个轴，因而秩的数量为1，它的型为(3,1)。这些值的获取方法如下：轴数量需要使用ndim属性，数组长度使用size属性，而数组的型要用shape属性。

```
>>> a.ndim
1
>>> a.size
3
>>> a.shape
(3L,)
```

你刚看到的这个数组再简单不过，只有一维。但是数组很容易就能扩展成多维。例如，可以定义一个2×2的二维数组：

```
>>> b = np.array([[1.3, 2.4],[0.3, 4.1]])
>>> b.dtype
dtype('float64')
>>> b.ndim
2
>>> b.size
4
>>> b.shape
(2L, 2L)
```

这个数组有两条轴，所以秩为2，每条轴的长度为2。

ndarray对象拥有另外一个叫作itemsize的重要属性。它定义了数组中每个元素的长度为几个字节。data属性表示的是包含数组实际元素的缓冲区。该属性至今用得并不多，因为要获取数组中的元素，使用接下来几节即将学到的索引方法即可。

```
>>> b.itemsize
8
>>> b.data
<read-write buffer for 0x0000000002D34DF0, size 32, offset 0 at 0x0000000002D5FEA0>
```

3.3.1　创建数组

数组的创建方法有几种，最常用的就是前面你见过的，使用array()函数，参数为单层或嵌套列表。

```
>>> c = np.array([[1, 2, 3],[4, 5, 6]])
>>> c
array([[1, 2, 3],
       [4, 5, 6]])
```

除了列表，array()函数还可以接收嵌套元组或元组列表作为参数。

```
>>> d = np.array(((1, 2, 3),(4, 5, 6)))
>>> d
array([[1, 2, 3],
       [4, 5, 6]])
```

此外，参数可以是由元组或列表组成的列表，其效果相同。

```
>>> e = np.array([(1, 2, 3), [4, 5, 6], (7, 8, 9)])
>>> e
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

3.3.2　数据类型

到目前为止，我们只使用过简单的整型和浮点型数据类型，其实 NumPy数组能够包含多种

数据类型（见表3-1）。例如，可以使用字符串类型：

```
>>> g = np.array([['a', 'b'],['c', 'd']])
>>> g
array([['a', 'b'],
       ['c', 'd']],
      dtype='|S1')
>>> g.dtype
dtype('S1')
>>> g.dtype.name
'string8'
```


表3-1 NumPy所支持的数据类型

数据类型	说　明
bool_	以一个字节形式存储的布尔值（True 或 False）
int_	默认整型（与C中的long相同，通常为int64或int32）
intc	完全等同于C中的int（通常为int32或int64）
intp	表示索引的整型（与C中的size_t相同，通常为int32或int64）
int8	字节（-128~127）
int16	整型（-32768~32767）
int32	整型（-2147483648~2147483647）
int64	整型（-9223372036854775808~9223372036854775807）
uint8	无符号整型（0~255）
uint16	无符号整型（0~65535）
uint32	无符号整型（0~4294967295）
uint64	无符号整型（0~18446744073709551615）
float_	float64的简写形式
float16	半精度浮点型：符号位、5位指数、10位小数部分
float32	单精度浮点型：符号位、8位指数、23位小数部分
float64	双精度浮点型：符号位、11位指数、52位小数部分
complex_	complex128的简写形式
complex64	复数，由两个32位的浮点数来表示（实数部分和虚数部分）
complex128	复数，由两个64位的浮点数来表示（实数部分和虚数部分）

3.3.3 dtype 选项

array()函数可以接收多个参数。每个ndarray()对象都有一个与之相关联的dtype对象，该对象唯一定义了数组中每个元素的数据类型。array()函数默认根据列表或元素序列中各元素的数据类型，为ndarray()对象指定最适合的数据类型。但是，你可以用dtype选项作为函数array()的参数，明确指定dtype的类型。

例如，如要定义一个复数数组，可以像下面这样使用dtype选项：

```
>>> f = np.array([[1, 2, 3],[4, 5, 6]], dtype=complex)
>>> f
```

```
array([[ 1.+0.j, 2.+0.j, 3.+0.j],
       [ 4.+0.j, 5.+0.j, 6.+0.j]])
```

3.3.4　自带的数组创建方法

NumPy库有几个函数能够生成包含初始值的*N*维数组，数组元素因函数而异。在学习本章乃至全书的过程中，你会发现这些函数非常有用。事实上，有了这些函数，仅用一行代码就能生成大量数据。

例如，zeros()函数能够生成由shape参数指定维度信息、元素均为零的数组。举个例子，下述代码会生成一个3 × 3型的二维数组：

```
>>> np.zeros((3, 3))
array([[ 0., 0., 0.],
       [ 0., 0., 0.],
       [ 0., 0., 0.]])
```

ones()函数与上述函数相似，生成一个各元素均为1的数组。

```
>>> np.ones((3, 3))
array([[ 1., 1., 1.],
       [ 1., 1., 1.],
       [ 1., 1., 1.]])
```

这两个函数默认使用float64数据类型创建数组。NumPy arange()函数特别有用。它根据传入的参数，按照特定规则，生成包含一个数值序列的数组。例如，如果要生成一个包含数字0到9的数组，只需传入标识序列结束的数字[1]作为参数即可。

```
>>> np.arange(0, 10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

如果不想以0作为起始值，可自己指定，这时需要使用两个参数：第一个为起始值，第二个为结束值。

```
>>> np.arange(4, 10)
array([4, 5, 6, 7, 8, 9])
```

还可以生成等间隔的序列。如果为arange()函数指定了第三个参数，它表示序列中相邻两个值之间的差距[2]有多大。

```
>>> np.arange(0, 12, 3)
array([0, 3, 6, 9])
```

此外，第三个参数还可以是浮点型[3]。

```
>>> np.arange(0, 6, 0.6)
array([ 0. , 0.6, 1.2, 1.8, 2.4, 3. , 3.6, 4.2, 4.8, 5.4])
```

① 用你想得到的序列的最后一个数字再加1作为参数。下面的例子使用了两个参数，其实如上所述，只传入一个参数即可，序列默认从0开始。

② 也被称作步长。

③ 这点就与Python的range()函数有所不同了，range()函数只可以使用整数作为步长。

到目前为止，你所创建的都是一维数组。如果要生成二维数组，仍然可以使用arange()函数，但是要结合reshape()函数。后者按照指定的形状，把一维数组拆分为不同的部分。

```
>>> np.arange(0, 12).reshape(3, 4)
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
```

另外一个跟arange()函数非常相似的函数是linspace()。它的前两个参数同样是用来指定序列的起始和结尾，但第三个参数不再表示相邻两个数字之间的距离，而是用来指定我们想把由开头和结尾两个数字所指定的范围分成几个部分。

```
>>> np.linspace(0,10,5)
array([  0. ,   2.5,   5. ,   7.5,   10. ])
```

最后，来讲讲另外一个创建包含初始值的数组的方法：使用随机数填充数组。可以使用numpy.random模块的random()函数，数组所包含的元素数量由参数指定。

```
>>> np.random.random(3)
array([ 0.78610272,  0.90630642,  0.80007102])
```

每次用random()函数生成的数组，其元素均会有所不同。若要生成多维数组，只需把数组的大小作为参数传递给它。

```
>>> np.random.random((3,3))
array([[ 0.07878569,  0.7176506 ,  0.05662501],
       [ 0.82919021,  0.80349121,  0.30254079],
       [ 0.93347404,  0.65868278,  0.37379618]])
```

3.4 基本操作

如今你已经知道了新建NumPy数组和定义数组元素的方法。我们该来学习数组的各种运算方法了。

3.4.1 算术运算符

数组的第一类运算是使用算术运算符进行的运算。最显而易见的是为数组加上或乘以一个标量。

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])

>>> a+4
array([4, 5, 6, 7])
>>> a*2
array([0, 2, 4, 6])
```

这些运算符还可以用于两个数组的运算。在NumPy中，这些运算符为*元素级*。也就是说，它们只用于位置相同的元素之间，所得到的运算结果组成一个新的数组。运算结果在新数组中的位

置跟操作数位置相同（见图3-1）。

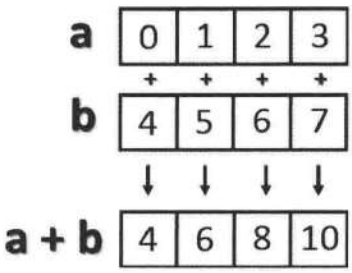


图3-1　元素级加法

```
>>> b = np.arange(4,8)
>>> b
array([4, 5, 6, 7])
>>> a + b
array([ 4, 6, 8, 10])
>>> a - b
array([-4, -4, -4, -4])
>>> a * b
array([ 0, 5, 12, 21])
```

此外，这些运算符还适用于返回值为NumPy数组的函数。例如，你可以用数组a乘上数组b的正弦值或平方根。

```
>>> a * np.sin(b)
array([-0.        , -0.95892427,  -0.558831 ,   1.9709598 ])
>>> a * np.sqrt(b)
array([ 0.        , 2.23606798,     4.89897949,   7.93725393])
```

对于多维数组，这些运算符仍然是元素级。

```
>>> A = np.arange(0, 9).reshape(3, 3)
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> B = np.ones((3, 3))
>>> B
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> A * B
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
```

3.4.2　矩阵积

选择使用*号作为元素级运算符是NumPy库比较奇怪的一点。事实上，在很多其他数据分析

工具中，*在用于两个矩阵之间的运算时指的是矩阵积（mastrix produet）。而NumPy用dot()函数表示这类乘法，注意，它不是元素级的。

```
>>> np.dot(A,B)
array([[ 3.,  3.,  3.],
       [ 12., 12., 12.],
       [ 21., 21., 21.]])
```

所得到的数组中每个元素为，第一个矩阵中与该元素行号相同的元素与第二个矩阵中与该元素列号相同的元素，两两相乘后再求和。图3-2描述的正是矩阵积的计算过程（只给出了矩阵积中两个元素的计算过程）。

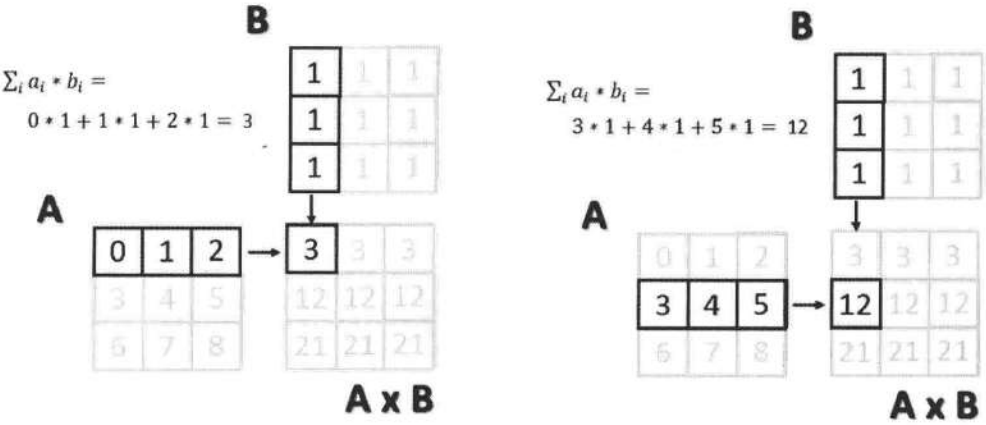


图3-2 矩阵积中元素的计算方法

矩阵积的另外一种写法是把dot()函数当作其中一个矩阵对象的方法。

```
>>> A.dot(B)
array([[ 3.,  3.,  3.],
       [ 12., 12., 12.],
       [ 21., 21., 21.]])
```

由于矩阵积计算不遵循交换律，因此在这里要多说一句，运算对象的顺序很重要。A*B确实不等于B*A。

```
>>> np.dot(B,A)
array([[ 9.,  12.,  15.],
       [ 9.,  12.,  15.],
       [ 9.,  12.,  15.]])
```

3.4.3 自增和自减运算符

Python没有++或--运算符。对变量的值进行自增与自减，需要使用+=或-=运算符。这两个运算符跟前面见过的只有一点不同，运算得到的结果不是赋给一个新数组而是赋给参与运算的数组自身。

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> a += 1
>>> a
array([1, 2, 3, 4])
>>> a -= 1
>>> a
array([0, 1, 2, 3])
```

因此，这类运算符比每次只能加1的自增运算符用途更广。例如，当你想修改数组元素的值而不想生成新数组时，就可以使用它们。

```
array([0, 1, 2, 3])
>>> a += 4
>>> a
array([4, 5, 6, 7])
>>> a *= 2
>>> a
array([ 8, 10, 12, 14])
```

3.4.4　通用函数

通用函数（universal function）通常叫作ufunc，它对数组中的各个元素逐一进行操作。这表明，通用函数分别处理输入数组的每个元素，生成的结果组成一个新的输出数组。输出数组的大小跟输入数组相同。

三角函数等很多数学运算符合通用函数的定义，例如，计算平方根的sqrt()函数、用来取对数的log()函数和求正弦值的sin()函数。

```
>>> a = np.arange(1, 5)
>>> a
array([1, 2, 3, 4])
>>> np.sqrt(a)
array([ 1.        , 1.41421356,  1.73205081,  2.        ])
>>> np.log(a)
array([ 0.        , 0.69314718,  1.09861229,  1.38629436])
>>> np.sin(a)
array([ 0.84147098, 0.90929743,  0.14112001, -0.7568025 ])
```

NumPy实现了很多通用函数。

3.4.5　聚合函数

聚合函数是指对一组值（比如一个数组）进行操作，返回一个单一值作为结果的函数。因而，求数组所有元素之和的函数就是聚合函数。ndarray类实现了多个这样的函数。

```
>>> a = np.array([3.3, 4.5, 1.2, 5.7, 0.3])
>>> a.sum()
15.0
>>> a.min()
```

```
0.29999999999999999
>>> a.max()
5.7000000000000002
>>> a.mean()
3.0
>>> a.std()
2.0079840636817816
```

3.5 索引机制、切片和迭代方法

前几节，我们讲解了数组创建和数组运算。这一节将介绍数组对象的操作方法，以及如何通过索引和切片方法选择元素，以获取数组中某几个元素的视图或者用赋值操作改变元素。最后，我们还会讲解数组的迭代方法。

3.5.1 索引机制

数组索引机制指的是用方括号（[]）加序号的形式引用单个数组元素，它的用处很多，比如抽取元素，选取数组的几个元素，甚至为其赋一个新值。

新建数组的同时，会生成跟数组大小一致的索引（见图3-3）。

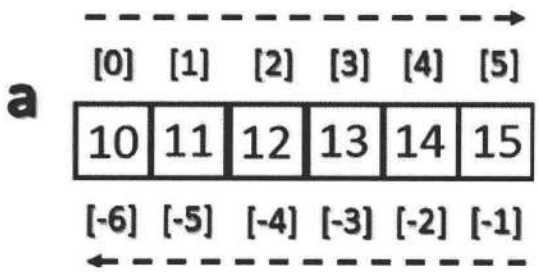


图3-3 一维数组的索引机制

要获取数组的单个元素，指定元素的索引即可。

```
>>> a = np.arange(10, 16)
>>> a
array([10, 11, 12, 13, 14, 15])
>>> a[4]
14
```

NumPy数组还可以使用负数作为索引。这些索引同样为递增序列，只不过从0开始，依次增加-1，但实际表示的是从数组的最后一个元素向数组第一个元素移动。在负数索引机制中，数组第一个元素的索引最小。

```
>>> a[-1]
15
>>> a[-6]
10
```

方括号内传入多个索引值，可以同时选择多个元素。

```
>>> a[[1, 3, 4]]
array([11, 13, 14])
```

再来看下二维数组，它也被称为矩阵。矩阵是由行和列组成的矩形数组，行和列用两条轴来定义，其中轴0用行表示，轴1用列表示。因此，二维数组的索引用一对值来表示：第一个值为行索引，第二个值为列索引。所以，如要获取或选取矩阵中的元素，仍使用方括号，但索引值为两个[行索引，列索引]（见图3-4）。

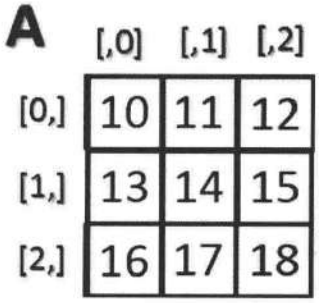


图3-4　二维数组的索引机制

```
>>> A = np.arange(10, 19).reshape((3, 3))
>>> A
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

因此，如想获取第二行第三列的元素，需要使用索引值[1, 2]。

```
>>> A[1, 2]
15
```

3.5.2　切片操作

切片操作是指抽取数组的一部分元素生成新数组。对Python列表进行切片操作得到的数组是原数组的副本，而对NumPy数组进行切片操作得到的数组则是指向相同缓冲区的视图。

如想抽取（或查看）数组的一部分，必须使用切片句法；也就是，把几个用冒号（:）隔开的数字置于方括号里。

如想抽取数组的一部分，例如从第二个到第六个元素这一部分，就需要在方括号里指定起始元素的索引1和结束元素的索引5。

```
>>> a = np.arange(10, 16)
>>> a
array([10, 11, 12, 13, 14, 15])
>>> a[1:5]
array([11, 12, 13, 14])
```

如想从上面那一部分元素中，每隔一定数量的元素抽取一个，可以再用一个数字指定所抽取的两个元素之间的间隔大小。例如，间隔为2，表示每隔一个元素抽取一个。

```
>>> a[1:5:2]
array([11, 13])
```

为了更好地理解切片句法，你还应该了解不明确指明起始和结束位置的情况。如省去第一个数字，NumPy会认为第一个数字是0（对应数组的第一个元素）；如省去第二个数字，NumPy则会认为第二个数字是数组的最大索引值；如省去最后一个数字，它将会被理解为1，也就是抽取所有元素而不再考虑间隔。

```
>>> a[::2]
array([10, 12, 14])
>>> a[:5:2]
array([10, 12, 14])
>>> a[:5:]
array([10, 11, 12, 13, 14])
```

对于二维数组，切分句法依然适用，只不过需要分别指定行和列的索引值。例如，只抽取第一行：

```
>>> A = np.arange(10, 19).reshape((3, 3))
>>> A
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
>>> A[0,:]
array([10, 11, 12])
```

上面代码中，第二个索引处只使用冒号，而没有指定任意数字，这样选择的是所有列。相反，如果想抽取第一列的所有元素，方括号中的两项应该交换位置。

```
>>> A[:,0]
array([10, 13, 16])
```

如要抽取一个小点儿的矩阵，需要明确指定所有的抽取范围。

```
>>> A[0:2, 0:2]
array([[10, 11],
       [13, 14]])
```

如要抽取的行或列的索引不连续，可以把这几个索引放到数组中。

```
>>> A[[0,2], 0:2]
array([[10, 11],
       [16, 17]])
```

3.5.3 数组迭代

Python数组元素的迭代很简单，只需要使用for结构即可。

```
>>> for i in a:
...     print i
```

```
...
10
11
12
13
14
15
```

二维数组当然也可以使用for结构，把两个嵌套在一起即可。第一层循环扫描数组的所有行，第二层循环扫描所有的列。实际上，如果遍历矩阵，你就会发现它总是按照第一条轴对矩阵进行扫描。

```
>>> for row in A:
...     print row
...
[10 11 12]
[13 14 15]
[16 17 18]
```

如果想遍历矩阵的每个元素，可以使用下面结构，用for循环遍历A.flat。

```
>>> for item in A.flat:
...     print item
...
10
11
12
13
14
15
16
17
18
```

除了for循环，NumPy还提供另外一种更为优雅的遍历方法。通常用函数处理行、列或单个元素时，需要用到遍历。如果想用聚合函数处理每一列或行，返回一个数值作为结果，最好用纯NumPy方法处理循环：apply_along_axis()函数。

这个函数接收三个参数：聚合函数、对哪条轴应用迭代操作和数组。如果axis选项的值为0，按列进行迭代操作，处理元素；值为1，则按行操作。例如，可以先求每一列的平均数，再求每一行的平均数。

```
>>> np.apply_along_axis(np.mean, axis=0, arr=A)
array([ 13., 14., 15.])
>>> np.apply_along_axis(np.mean, axis=1, arr=A)
array([ 11., 14., 17.])
```

上述例子使用了NumPy库定义的函数，但是你也可以自己定义这样的函数。上面还使用了聚合函数，然而，用通用函数也未尝不可。下面的例子，先后按行、列进行迭代操作，但两者的最终结果一致。通用函数apply_along_axis()实际上是按照指定的轴逐元素遍历数组。

```
>>> def foo(x):
...     return x/2
```

```
...
>>> np.apply_along_axis(foo, axis=1, arr=A)
array([[5, 5, 6],
       [6, 7, 7],
       [8, 8, 9]])
>>> np.apply_along_axis(foo, axis=0, arr=A)
array([[5, 5, 6],
       [6, 7, 7],
       [8, 8, 9]])
```

如上所见，不论是遍历行还是遍历列，通用函数都将输入数组的每个元素做折半处理。

3.6 条件和布尔数组

到目前为止，你已经尝试过用索引和切片方法从数组中选择或抽取一部分元素。这些方法使用数值形式的索引。另外一种从数组中有选择性地抽取元素的方法是使用条件表达式和布尔运算符。

我们来详细看一下这种方法。例如，你想从由0到1之间的随机数组成的4 × 4型矩阵中选取所有小于0.5的元素。

```
>>> A = np.random.random((4, 4))
>>> A
array([[ 0.03536295,  0.0035115 ,  0.54742404, 0.68960999],
       [ 0.21264709,  0.17121982,  0.81090212, 0.43408927],
       [ 0.77116263,  0.04523647,  0.84632378, 0.54450749],
       [ 0.86964585,  0.6470581 ,  0.42582897, 0.22286282]])
```

创建随机数矩阵后，如果使用表示条件的运算符，比如这里的小于号，你将会得到由布尔值组成的数组。对于原数组中条件满足的元素，布尔数组中处于同等位置（也就是小于0.5的元素所处的位置）的元素为True。

```
>>> A < 0.5
array([[ True,  True, False, False],
       [ True,  True, False,  True],
       [False,  True, False, False],
       [False, False,  True,  True]], dtype=bool)
```

实际上，从数组中选取一部分元素时，隐式地用到了布尔数组。其实，直接把条件表达式置于方括号中，也能抽取所有小于0.5的元素，组成一个新数组。

```
>>> A[A < 0.5]
array([ 0.03536295,  0.0035115 ,  0.21264709,  0.17121982,  0.43408927,
        0.04523647,  0.42582897,  0.22286282])
```

3.7 形状变换

创建二维数组时，你已经见过用reshape()函数把一维数组转换为矩阵。

```
>>> a = np.random.random(12)
>>> a
array([ 0.77841574,  0.39654203,  0.38188665,  0.26704305, 0.27519705,
        0.78115866,  0.96019214,  0.59328414,  0.52008642, 0.10862692,
        0.41894881,  0.73581471])
>>> A = a.reshape(3, 4)
>>> A
array([[ 0.77841574,  0.39654203,  0.38188665,  0.26704305],
       [ 0.27519705,  0.78115866,  0.96019214,  0.59328414],
       [ 0.52008642,  0.10862692,  0.41894881,  0.73581471]])
```

reshape()函数返回一个新数组，因而可用来创建新对象。然而，如果想通过改变数组的形状来改变数组对象，需把表示新形状的元组直接赋给数组的shape属性。

```
>>> a.shape = (3, 4)
>>> a
array([[ 0.77841574,  0.39654203,  0.38188665,  0.26704305],
       [ 0.27519705,  0.78115866,  0.96019214,  0.59328414],
       [ 0.52008642,  0.10862692,  0.41894881,  0.73581471]])
```

由输出结果来看，上述操作改变了原始数组的形状，而没有返回新对象。改变数组形状的操作是可逆的，ravel()函数可以把二维数组再变回一维数组。

```
>>> a = a.ravel()
array([ 0.77841574,  0.39654203,  0.38188665,  0.26704305, 0.27519705,
        0.78115866,  0.96019214,  0.59328414,  0.52008642, 0.10862692,
        0.41894881,  0.73581471])
```

甚至直接改变数组shape属性的值也可以。

```
>>> a.shape = (12)
>>> a
array([ 0.77841574,  0.39654203,  0.38188665,  0.26704305,  0.27519705,
        0.78115866,  0.96019214,  0.59328414,  0.52008642,  0.10862692,
        0.41894881,  0.73581471])
```

另外一种重要的运算是交换行列位置的矩阵转置。NumPy的transpose()函数实现了该功能。

```
>>> A.transpose()
array([[ 0.77841574, 0.27519705,   0.52008642],
       [ 0.39654203,  0.78115866,  0.10862692],
       [ 0.38188665,  0.96019214,  0.41894881],
       [ 0.26704305,  0.59328414,  0.73581471]])
```

3.8　数组操作

往往需要用已有数组创建新数组。本节来看一下如何通过连接或切分已有数组创建新数组。

3.8.1　连接数组

你可以把多个数组整合在一起形成一个包含这些数组的新数组。NumPy使用了栈这个概念，

提供了几个运用栈概念的函数。例如，vstack()函数执行垂直入栈操作，把第二个数组作为行添加到第一个数组，数组朝垂直方向生长。相反，hstack()函数执行水平入栈操作，也就是说把第二个数组作为列添加到第一个数组。

```
>>> A = np.ones((3, 3))
>>> B = np.zeros((3, 3))
>>> np.vstack((A, B))
array([[ 1., 1., 1.],
       [ 1., 1., 1.],
       [ 1., 1., 1.],
       [ 0., 0., 0.],
       [ 0., 0., 0.],
       [ 0., 0., 0.]])
>>> np.hstack((A,B))
array([[ 1., 1., 1., 0., 0., 0.],
       [ 1., 1., 1., 0., 0., 0.],
       [ 1., 1., 1., 0., 0., 0.]])
```

另外两个用于多个数组之间栈操作的函数是column_stack()和row_stack()。这两个函数不同于上面两个。一般来讲，这两个函数把一维数组作为列或行压入栈结构，以形成一个新的二维数组。

```
>>> a = np.array([0, 1, 2])
>>> b = np.array([3, 4, 5])
>>> c = np.array([6, 7, 8])
>>> np.column_stack((a, b, c))
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
>>> np.row_stack((a, b, c))
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

3.8.2 数组切分

上面讲了使用栈操作把多个数组组装到一起的方法。接下来看一下它的逆操作：把一个数组分为几部分。在NumPy中，该操作要用到切分方法。同理，我们也有一组函数，水平切分用hsplit()函数，垂直切分用vsplit()函数。

```
>>> A = np.arange(16).reshape((4, 4))
>>> A
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11],
       [12, 13, 14, 15]])
```

水平切分数组的意思是把数组按照宽度切分为两部分，例如4 × 4矩阵将被切分为两个4 × 2矩阵。

```
>>> [B,C] = np.hsplit(A, 2)
>>> B
array([[ 0, 1],
       [ 4, 5],
       [ 8, 9],
       [12, 13]])
>>> C
array([[ 2, 3],
       [ 6, 7],
       [10, 11],
       [14, 15]])
```

反之，垂直切分指的是把数组按照高度分为两部分，如4×4矩阵将被切为两个2×4矩阵。

```
>>> [B,C] = np.vsplit(A, 2)
>>> B
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> C
array([[ 8, 9, 10, 11],
       [12, 13, 14, 15]])
```

split()函数更为复杂，可以把数组分为几个不对称的部分。此外，除了传入数组作为参数外，还得指定被切分部分的索引。如果指定axis=1项，索引为列索引；如果axis=0，索引为行索引。

例如，要把矩阵切分为三部分，第一部分为第一列，第二部分为第二列、第三列，而第三部分为最后一列。你需要像下面这样指定索引值。

```
>>> [A1,A2,A3] = np.split(A,[1,3],axis=1)
>>> A1
array([[ 0],
       [ 4],
       [ 8],
       [12]])
>>> A2
array([[ 1, 2],
       [ 5, 6],
       [ 9, 10],
       [13, 14]])
>>> A3
array([[ 3],
       [ 7],
       [11],
       [15]])
```

你也可以按行切分，方法相同。

```
>>> [A1,A2,A3] = np.split(A,[1,3],axis=0)
>>> A1
array([[0, 1, 2, 3]])
>>> A2
array([[ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
```

```
>>> A3
array([[12, 13, 14, 15]])
```

split()函数还具有vsplit()和hsplit()函数的功能。

3.9 常用概念

这一节将介绍NumPy库的几个常用概念。我们会讲解副本和视图的区别，其中着重讲解两者返回值的不同点。我们还会介绍NumPy函数的很多事务（transaction）隐式使用的广播机制（broadcasting）。

3.9.1 对象的副本或视图

你可能已经注意到，NumPy中，尤其是在做数组运算或数组操作时，返回结果不是数组的副本就是视图。NumPy中，所有赋值运算不会为数组和数组中的任何元素创建副本。

```
>>> a = np.array([1, 2, 3, 4])
>>> b = a
>>> b
array([1, 2, 3, 4])
>>> a[2] = 0
>>> b
array([1, 2, 0, 4])
```

把数组a赋给数组b，实际上不是为a创建副本，b只不过是调用数组a的另外一种方式。事实上，修改a的第三个元素，同样会修改b的第三个元素。数组切片操作返回的对象只是原数组的视图。[①]

```
>>> c = a[0:2]
>>> c
array([1, 2])
>>> a[0] = 0
>>> c
array([0, 2])
```

如上所见，即使是切片操作得到的结果，实际上仍指向相同的对象。如果想为原数组生成一份完整的副本，从而得到一个不同的数组，使用copy()函数即可。

```
>>> a = np.array([1, 2, 3, 4])
>>> c = a.copy()
>>> c
array([1, 2, 3, 4])
>>> a[0] = 0
>>> c
array([1, 2, 3, 4])
```

上面的例子中，即使改动数组a的元素，数组c仍保持不变。

① 注意与Python列表切片操作区别开来。列表操作得到的是副本。

3.9.2　向量化

向量化和广播这两个概念是NumPy内部实现的基础。有了向量化，编写代码时无需使用显式循环。这些循环实际上不能省略，只不过是在内部实现，被代码中的其他结构代替。向量化的应用使得代码更简洁，可读性更强，你可以说使用了向量化方法的代码看上去更“Pythonic”。向量化使得很多运算看上去更像是数学表达式，例如，NumPy中两个数组相乘可以表示为：

```
a * b
```

甚至两个矩阵相乘也可以这么表示：

```
A * B
```

其他语言的上述运算要用到多重for结构。例如，计算数组相乘：

```
for (i = 0; i < rows; i++){
  c[i] = a[i]*b[i];
}
```

计算矩阵相乘：

```
for( i=0; i < rows; i++){
   for(j=0; j < columns; j++){
      c[i][j] = a[i][j]*b[i][j];
   }
}
```

由上可见，使用NumPy时，代码的可读性更强，其表达式更像是数学表达式。

3.9.3　广播机制

广播机制这一操作实现了对两个或以上数组进行运算或用函数处理，即使这些数组形状并不完全相同。并不是所有的维度都要彼此兼容才符合广播机制的要求，但它们必须要满足一定的条件。

前面讲过，在NumPy中，如何通过用表示数组各维度长度的元组（也就是数组的型）把数组转换成多维数组。

因此，若两个数组的各维度兼容，也就是两个数组的每一维等长，或其中一个数组为一维，那么广播机制就适用。如果这两个条件都不能满足，NumPy就会抛出异常，说两个数组不兼容。

```
>>> A = np.arange(16).reshape(4, 4)
>>> b = np.arange(4)
>>> A
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11],
       [12, 13, 14, 15]])
>>> b
array([0, 1, 2, 3])
```

执行完上述代码后。我们就得到两个数组：

```
4 x 4
4
```

广播机制有两条规则。第一条是为缺失的维度补上个1。如果这时满足兼容性条件，就可以应用广播机制，再来看第二条规则。

```
4 x 4
4 x 1
```

兼容性规则满足之后，再来看一下广播机制的第二条规则。这一规则解释的是如何扩展最小的数组，使得它跟最大的数组大小相同，以便使用元素级的函数或运算符。

第二条规则假定缺失元素（一维）都用已有值进行了填充（见图3-5）。

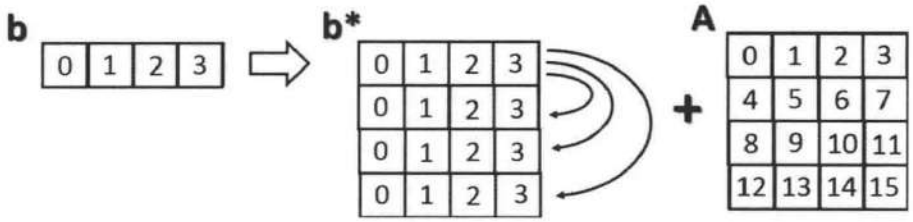


图3-5 应用广播机制的第二条规则

既然两个数组维度相同，它们里面的值就可以相加。

```
>>> A + b
array([[ 0, 2, 4, 6],
       [ 4, 6, 8, 10],
       [ 8, 10, 12, 14],
       [12, 14, 16, 18]])
```

上例这种情况比较简单，一个数组较另一个小。还有更复杂的情况，即两个数组形状不同、维度不同、互有长短。

```
>>> m = np.arange(6).reshape(3, 1, 2)
>>> n = np.arange(6).reshape(3, 2, 1)
>>> m
array([[[0, 1]],

       [[2, 3]],

       [[4, 5]]])
>>> n
array([[[0],
        [1]],

       [[2],
        [3]],

       [[4],
        [5]]])
```

即使是这种复杂情况，分析两个数组的形状，你也会发现它们相互兼容，因此广播规则仍然适用。

```
3 x 1 x 2
3 x 2 x 1
```

这种情况下，两个数组都要扩展维度（进行广播）。

```
m* = [[[0,1],          n* = [[[0,0],
       [0,1]],                [1,1]],
      [[2,3],                 [[2,2],
       [2,3]],                 [3,3]],
      [[4,5],                 [[4,4],
       [4,5]]]                 [5,5]]]
```

然后，就可以对两个数组进行诸如加法这样的元素级运算。

```
>>> m + n
array([[[ 0,  1],
        [ 1,  2]],

       [[ 4,  5],
        [ 5,  6]],

       [[ 8,  9],
        [ 9, 10]]])
```

3.10　结构化数组

通过前面几节的多个例子，我们讲了一维数组和二维数组。在NumPy中，不仅可以创建规模更为复杂的数组，还可以创建结构更为复杂的数组，后者叫作*结构化数组*（structured array），它包含的是结构或记录而不是独立的元素。

例如，你可以创建一个简单的结构化数组，其中元素为结构体。你可以用dtype选项，指定一系列用逗号隔开的说明符，指明组成结构体的元素及它们的数据类型和顺序。

```
bytes                b1
int                  i1, i2, i4, i8
unsigned ints        u1, u2, u4, u8
floats               f2, f4, f8
complex              c8, c16
fixed length strings a<n>
```

例如，你想指定由一个整数、一个长度为6的字符串和一个布尔值组成的结构体，就要在dtype选项中按顺序指定各自的说明符。

```
>>> structured = np.array([(1, 'First', 0.5, 1+2j),(2, 'Second', 1.3, 2-2j),
(3, 'Third', 0.8, 1+3j)],dtype=('i2, a6, f4, c8'))
>>> structured
array([(1, 'First', 0.5, (1+2j)),
       (2, 'Second', 1.2999999523162842, (2-2j)),
       (3, 'Third', 0.800000011920929, (1+3j))],
      dtype=[('f0', '<i2'), ('f1', 'S6'), ('f2', '<f4'), ('f3', '<c8')])
```

你还可以在数据类型（dtype）选项中明确指定每个元素的类型，如int8、uint8、float16、

complex64等。

```
>>> structured = np.array([(1, 'First', 0.5, 1+2j),(2, 'Second', 1.3,2-2j),
(3, 'Third', 0.8, 1+3j)],dtype=('
int16, a6, float32, complex64'))
>>> structured
array([(1, 'First', 0.5, (1+2j)),
       (2, 'Second', 1.2999999523162842, (2-2j)),
       (3, 'Third', 0.800000011920929, (1+3j))],
      dtype=[('f0', '<i2'), ('f1', 'S6'), ('f2', '<f4'), ('f3', '<c8')])
```

然而，上述两种做法结果相同。生成的数组中，dtype序列包含结构体各项的名称及相应的数据类型。

使用索引值，就能获取到包含相应结构体的行。

```
>>> structured[1]
(2, 'Second', 1.2999999523162842, (2-2j))
```

自动赋给结构体每个元素的名称可以看成数组列的名称。用它们作为结构化索引，就能引用类型相同或是位于同列的元素。

```
>>> structured['f1']
array(['First', 'Second', 'Third'],
      dtype='|S6')
```

如上所见，自动分配的名称的第一个字符为f（field，字段），后面紧跟的是表示它在序列中位置的整数。其实，用更有意义的内容作为名字，用处更大。在创建数组时，可以指定各字段的名称：

```
>>> structured = np.array([(1,'First',0.5,1+2j),(2,'Second',1.3,2-2j),(3,'Third',0.8,1+3j)],
dtype=[('id','i2'),('position','a6'),('value','f4'),('complex','c8')])
>>> structured
array([(1, 'First', 0.5, (1+2j)),
       (2, 'Second', 1.2999999523162842, (2-2j)),
       (3, 'Third', 0.800000011920929, (1+3j))],
      dtype=[('id', '<i2'), ('position', 'S6'), ('value', '<f4'), ('complex', '<c8')])
```

或在创建完成后，重新定义结构化数组的dtype属性，在元组中指定各字段的名称。

```
>>> structured.dtype.names = ('id','order','value','complex')
```

现在，你可以使用更有意义的字段名来获取数组的某一列。

```
>>> structured['order']
array(['First', 'Second', 'Third'],
      dtype='|S6')
```

3.11 数组数据文件的读写

我们还没有讲如何读取文件中的数据。NumPy这方面的内容很重要，用处很大，尤其是在处理数组中包含大量数据的情况时。这在数据分析中很常见，因为要分析的数据集通常都很大，所

以由人工来管理这类事务的执行，以及接下来的从一台计算机或计算过程的一段会话读取数据到另一台计算机或另一段会话，是不可取甚至是不可能的。

鉴于此，NumPy提供了几个函数，数据分析师可用其把结果保存到文本或二进制文件中。类似地，NumPy还提供了从文件中读取数据并将其转换为数组的方法。

3.11.1　二进制文件的读写

NumPy的save()方法以二进制格式保存数据，load()方法则从二进制文件中读取数据。

假如你有一个数组要保存，例如数据分析过程产生的结果，调用save()函数即可，参数有两个：要保存到的文件名和要保存的数组，其中文件名中的.npy扩展名系统会自动添加。

```
>>> data
array([[ 0.86466285,  0.76943895,  0.22678279],
       [ 0.12452825,  0.54751384,  0.06499123],
       [ 0.06216566,  0.85045125,  0.92093862],
       [ 0.58401239,  0.93455057,  0.28972379]])
>>> np.save('saved_data',data)
```

若要恢复存储在.npy文件中的数据，可以使用load()函数，用文件名作为参数，这次记得添加.npy扩展名。

```
>>> loaded_data = np.load('saved_data.npy')
>>> loaded_data
array([[ 0.86466285,  0.76943895,  0.22678279],
       [ 0.12452825,  0.54751384,  0.06499123],
       [ 0.06216566,  0.85045125,  0.92093862],
       [ 0.58401239,  0.93455057,  0.28972379]])
```

3.11.2　读取文件中的列表形式数据

很多时候，你要读写文本格式的数据（如TXT或CSV）。当你使用NumPy或其他应用时，考虑到文本格式的文件不必使用这些应用也能处理，因此一般都会将数据存储为文本格式而不是二进制格式。拿几行CSV（Comma-Separated Values，用逗号分割的值）格式的数据为例。这种格式为列表形式，每两个值之间用逗号隔开（见代码清单3-1）。

代码清单3-1　data.csv

```
id,value1,value2,value3
1,123,1.4,23
2,110,0.5,18
3,164,2.1,19
```

NumPy的genfromtxt()函数可以从文本文件中读取数据并将其插入数组中。通常而言，这个函数接收三个参数：存放数据的文件名、用于分割值的字符和是否含有列标题。在接下来这个例子中，分隔符为逗号。

```
>>> data = np.genfromtxt('data.csv', delimiter=',', names=True)
```

```
>>> data
array([(1.0,  123.0, 1.4, 23.0), (2.0, 110.0, 0.5, 18.0),
       (3.0,  164.0, 2.1, 19.0)],
      dtype=[('id', '<f8'), ('value1', '<f8'), ('value2', '<f8'), ('value3', '<f8')])
```

从输出结果可以看到，我们得到了一个结构化数组，各列的标题变为各字段的名称。

这个函数其实包含两层隐式循环：第一层循环每次读取一行；第二层循环将每一行的多个值分开后，再对这些值进行转化，依次插入所创建的元素。这个函数的优点是它能处理文件中的缺失数据。

以上面的文件为例（见代码清单3-2），我们从中删除几个元素后，将其另存为data2.csv。

代码清单3-2 data2.csv

```
id,value1,value2,value3
1,123,1.4,23
2,110,,18
3,,2.1,19
```

运行下述命令，观察genfromtxt()是怎样把内容为空的项填充为nan值的。

```
>>> data2 = np.genfromtxt('data2.csv', delimiter=',', names=True)
>>> data2
array([(1.0, 123.0, 1.4, 23.0), (2.0, 110.0, nan, 18.0),
       (3.0, nan, 2.1, 19.0)],
      dtype=[('id', '<f8'), ('value1', '<f8'), ('value2', '<f8'), ('value3', '<f8')])
```

输出结果中，数组的下面为文件的列标题。可以将这些标题看成能够充当索引的标签，用它们就能按列抽取元素。

```
>>> data2['id']
array([ 1., 2., 3.])
```

而按照传统方法，使用数值索引则是按行抽取数据。

```
>>> data2[0]
(1.0, 123.0, 1.4, 23.0)
```

3.12 小结

本章介绍了NumPy库所有的主要内容。通过一系列例子，你熟悉了NumPy的多种功能，它们是书中其他内容的基础。事实上，后续多个概念来自其他更为专业的科学计算库，但是这些库的结构参考了NumPy，并且是以NumPy库为基础进行开发的。

你还从本章学到了ndarray如何扩展了Python的功能，从而使其适用于科学计算，尤其是数据分析。

对任何想从事数据分析的人来说，NumPy都是一项至关重要的技能。

下一章，我们将介绍一个新库pandas。它以NumPy为基础，吸收了本章讲到的所有基础概念，并对它们进行了扩展，以使其更适合数据分析。

第4章

pandas库简介

你终于要走进本书的心脏了：pandas库。这个库是用Python语言分析数据的好工具。

你首先将学习这个库的基础知识、安装方法，最后将熟悉Series（序列）和DataFrame（数据框）这两种数据结构。在学习本章的过程中，你将学着使用pandas库的几个基础函数处理最常见的数据分析任务。熟悉这些操作对本书后续内容的学习起着至关重要的作用。因此，很有必要多重复几遍从本章学到的所有技能，直到熟练掌握。

我们还将通过多个例子讲解pandas库采用的新概念：它的数据结构所使用的索引机制。至于如何充分利用索引机制处理数据，这一章和接下来几章会作讲解。

本章最后，我们来看一下如何将索引机制这个概念同时扩展到多层：层级索引。

4.1 pandas：Python 数据分析库

pandas是一个专门用于数据分析的开源Python库。目前，所有使用Python语言研究和分析数据集的专业人士，在做相关统计分析和决策时，pandas都是他们的基础工具。

2008年，Wes McKinney一人挑起了pandas库的设计和开发工作；2012年，他的同事Sien Chang加入开发。他俩一起开发出了Python社区最为有用的库之一——pandas。

数据分析工作需要一个专门的库，它能够以最简单的方式提供数据处理、数据抽取和数据操作所需的全部工具，开发pandas正是为了满足这个需求。

Wes McKinney选择以NumPy库作为Python库pandas的基础进行设计。可以说，该选择对于pandas的成功和它的迅速扩展起着至关重要的作用。事实上，选择以NumPy为基础，不仅使pandas能和其他大多数模块相兼容，而且还能借力NumPy模块在计算方面性能高的优势。

另外一个意义深远的决定是为数据分析专门设计了两种数据结构。实际情况是，pandas没有使用Python已有的内置数据结构，也没有使用其他库的数据结构，而是开发了两种新型的数据结构。

这两种数据结构的设计初衷是用于关系型或带标签的数据。用它们管理与SQL关系数据库和Excel工作表具有类似特征的数据很方便。

本书会讲到一系列数据分析的基础操作，操作对象通常为数据库表或工作表。pandas提供多

个函数和方法用于数据分析，在很多情况下，它们是执行这些操作的最佳方法。

因此，pandas的主要目的是为每一位数据分析人士提供所有的基础工具。

4.2 安装

安装pandas库最简单和最常用的方法是先安装一个发行版，例如，先安装Anaconda或Enthought，再用发行版安装pandas。

4.2.1 用 Anaconda 安装

对于选用Anaconda发行版的读者，安装pandas很简单。首先，我们来看一下pandas是否已经安装，安装的版本号是多少。为此，在终端输入以下命令：

```
conda list pandas
```

因为我事先在我的个人计算机（Windows系统）上安装了pandas库，所以得到如下输出结果：

```
# packages in environment at C:\Users\Fabio\Anaconda:
#
pandas                    0.14.1               np19py27_0
```

如果你运行上述命令时的输出结果不是这样，就需要安装pandas库。请输入以下命令：

```
conda install pandas
```

Anaconda立即检查它所有的依赖库，管理其他模块的安装，为你省心不少。

如果想更新pandas库，命令也很简单直接：

```
conda update pandas
```

Anaconda会检查pandas以及所有依赖模块的版本，如有要更新的，即予以提示，然后询问你是否想更新。

```
Fetching package metadata: ....
Solving package specifications: .
Package plan for installation in environment C:\Users\Fabio\Anaconda:
```

Anaconda将会下载下面这些包。

```
Package                 |            build
------------------------|-----------------------------------
pytz-2014.9             |          py27_0         169 KB
requests-2.5.3          |          py27_0         588 KB
six-1.9.0               |          py27_0          16 KB
conda-3.9.1             |          py27_0         206 KB
pip-6.0.8               |          py27_0         1.6 MB
scipy-0.15.1            |        np19py27_0       71.3 MB
pandas-0.15.2           |        np19py27_0       4.3 MB
------------------------------------------------------------
                                   Total:        78.1 MB
```

Anaconda将会更新下面这些包。

```
conda:    3.9.0-py27_0      --> 3.9.1-py27_0
pandas:   0.14.1-np19py27_0 --> 0.15.2-np19py27_0
pip:      1.5.6-py27_0      --> 6.0.8-py27_0
pytz:     2014.7-py27_0     --> 2014.9-py27_0
requests: 2.5.1-py27_0      --> 2.5.3-py27_0
scipy:    0.14.0-np19py27_0 --> 0.15.1-np19py27_0
six:      1.8.0-py27_0      --> 1.9.0-py27_0

Proceed ([y]/n)?
```

敲入y键后，Anaconda开始从网上下载所有新的模块。因此，如要执行这一步，计算机需联网。

```
Fetching packages ...
scipy-0.15.1-n 100% |################################| Time: 0:01:11   1.05 MB/s
Extracting packages ...
[      COMPLETE      ] |##################################################| 100%
Unlinking packages ...
[      COMPLETE      ] |##################################################| 100%
Linking packages ...
[      COMPLETE      ] |##################################################| 100%
```

4.2.2　用 PyPI 安装

还可以从PyPI安装pandas：

```
pip install pandas
```

4.2.3　在 Linux 系统的安装方法

如果你用的是某一Linux发行版，且不打算使用打包好的Python发行版，可以像安装其他包那样安装pandas。

Debian和Ubuntu Linux系统：

```
sudo apt-get install python-pandas
```

OpenSuse和Fedora Linux系统，则需要使用以下命令：

```
zypper in python-pandas
```

4.2.4　用源代码安装

如果你想通过编译源代码来安装pandas，请参考以下GitHub链接：http://github.com/pydata/pandas。

```
git clone git://github.com/pydata/pandas.git
cd pandas
python setup.py install
```

编译前，确保已经安装Cython。更多信息请参考包括官方文档（http://pandas.pydata.org/

pandas-docs/stable/install.html）在内的在线文档。

4.2.5 Windows 模块仓库

Windows系统用户如果喜欢自己管理模块，以便总是使用最新模块，可以从网上模块仓库下载很多第三方模块：Christoph Gohlke的Windows系统Python扩展包仓库（www.lfd.uci.edu/~gohlke/pythonlibs/）。每个模块都提供32位和64位WHL（wheel）格式的安装包。如果要安装模块，需要使用pip命令（参见第2章）。

```
pip install SomePackege-1.0.whl
```

选择模块时，注意选择与Python版本和计算机系统相兼容的版本。此外，虽然NumPy不依赖其他包，但是pandas依赖多个包。要确保安装所有的依赖包，不过安装顺序并不重要。

这种方法的缺点是，每个包要单独安装，没有包管理器来帮你管理版本和依赖；但优点是，对这些模块及其版本有更大的控制权，你不用使用Anaconda等发行版提供的包，而是使用最新的包。

4.3 测试 pandas 是否安装成功

pandas库还提供一项功能，安装完成后，可运行测试，检查内部命令是否能够执行（官方文档表示，所有内部代码的测试覆盖率高达97%）。

首先确保Python发型版安装了nose模块（请见下面“nose模块”的介绍）。如已安装，输入以下命令开始测试：

```
nosetests pandas
```

测试任务需要花费几分钟时间，测试完成后，将显示问题列表。

nose模块

nose模块是用来在项目开发阶段，特别是Python模块的开发阶段，测试Python代码的。这个模块扩展了unittest模块的功能，Python的unittest模块是用来测试代码的。与其相比，nose模块简化了测试代码，降低了它的编写难度。

我建议你读读这篇文章：http://pythontesting.net/framework/nose/nose-introduction/。

4.4 开始 pandas 之旅

本章的主题是解释数据结构和用来处理这些数据结构的相关函数/方法，所以不需要编写大段代码。

因此，我认为学习本章的好方法是打开Python shell，逐条输入命令。这样，你就有机会熟悉本章依次讲解的单个函数和数据结构。

此外，本章前面例子中定义的数据和函数在后面仍然有效，无需每次重复定义。每个例子结束后，你最好重复练习各条命令，可做些适当的修改，在操作过程中留意如何操纵数据结构中的数据。这种方法非常适合用来熟悉本章讲解的几项主要内容，你可以尽情地以交互式方式探索命令的作用，从而避免机械地编写和执行代码。

注意　本章假定你多少熟悉Python和NumPy。如遇到任何困难，请阅读前面的第2章和第3章。

首先，在Python shell打开一段会话，导入pandas库。pandas的常用导入方法如下：

```
>>> import pandas as pd
>>> import numpy as np
```

因此，本书中从此往后，再见到pd和np时，它们分别指的是与pandas和NumPy这两个库相关的对象或方法，即使你可能经常想使用下面这种方法导入pandas模块：

```
>>> from pandas import *
```

这样，你就无需用pd指定函数、对象或方法。然而，通常来说，这种方法在Python社区看来是不太好的。

4.5　pandas数据结构简介

pandas的核心为两大数据结构，数据分析相关的所有事务都是围绕着这两种结构进行的：

- Series
- DataFrame

后面也会讲，Series这类数据结构用于存储一个序列这样的一维数据，而DataFrame作为更复杂的数据结构，则用于存储多维数据。

虽然这些数据结构不能解决所有问题，但它们为大多数应用提供了有效和强大的工具。就简洁性而言，它们理解和使用起来都很简单。此外，很多更为复杂的数据结构都可以追溯到这两种结构。

然而，两者的奇特之处是将Index（索引）对象和标签整合到自己的结构中。后面你将会感受到，该特点使得这两种数据结构具有很强的可操作性。

4.5.1　Series对象

pandas库的Series对象用来表示一维数据结构，跟数组类似，但多了一些额外的功能。它的内部结构很简单（见图4-1），由两个相互关联的数组组成，其中主数组用来存放数据（NumPy任意类型数据）。主数组的每个元素都有一个与之相关联的标签，这些标签存储在另外一个叫作Index的数组中。

Series	
index	value
0	12
1	-4
2	7
3	9

图4-1 Series对象的结构

1. 声明Series对象

调用Series()构造函数，把要存放在Series对象中的数据以数组形式传入，就能创建一个如图4-1所示的Series对象。

```
>>> s = pd.Series([12,-4,7,9])
>>> s
0    12
1    -4
2     7
3     9
dtype: int64
```

从Series的输出可以看到，左侧Index是一列标签，右侧是标签对应的元素。

声明Series时，若不指定标签，pandas默认使用从0开始依次递增的数值作为标签。这种情况下，标签与Series对象中元素的索引（在数组中的位置）一致。

然而，最好使用有意义的标签，用以区分和识别每个元素，而不用考虑元素插入到Series中的顺序。

因此，调用构造函数时，就需要指定index选项，把存放有标签的数组赋给它，其中标签为字符串类型。

```
>>> s = pd.Series([12,-4,7,9], index=['a','b','c','d'])
>>> s
a    12
b    -4
c     7
d     9
dtype: int64
```

如果想分别查看组成Series对象的两个数组，可像下面这样调用它的两个属性：index（索引）和values（元素）。

```
>>> s.values
array([12, -4, 7, 9], dtype=int64)
>>> s.index
Index([u'a', u'b', u'c', u'd'], dtype='object')
```

2. 选择内部元素

若想获取Series对象内部的元素，把它作为普通的NumPy数组，指定键即可。

```
>>> s[2]
7
```

或者，指定位于索引位置处的标签。

```
>>> s['b']
-4
```

跟从NumPy数组选择多个元素的方法相同，可像下面这样选取多项：

```
>>> s[0:2]
A   12
b   -4
dtype: int64
```

或者，这种情况甚至可以使用元素对应的标签，只不过要把标签放到数组中：

```
>>> s[['b','c']]
b  -4
c   7
dtype: int64
```

3. 为元素赋值

既然你已理解单个元素的选取方法，赋值方法也就不言自明。可以用索引或标签选取元素后进行赋值。

```
>>> s[1] = 0
>>> s
a  12
b   0
c   7
d   9
dtype: int64
>>> s['b'] = 1
>>> s
a   12
b    1
c    7
d    9
dtype: int64
```

4. 用NumPy数组或其他Series对象定义新Series对象

你可以用NumPy数组或现有的Series对象定义新的Series对象。

```
>>> arr = np.array([1,2,3,4])
>>> s3 = pd.Series(arr)
>>> s3
0   1
1   2
2   3
3   4
dtype: int32
```

```
>>> s4 = pd.Series(s)
>>> s4
a    12
b     4
c     7
d     9
dtype: int64
```

然而，这样做时不要忘记新Series对象中的元素不是原NumPy数组或Series对象元素的副本，而是对它们的引用。也就是说，这些对象是动态插入到新Series对象中。如改变原有对象元素的值，新Series对象中这些元素也会发生改变。

```
>>> s3
0    1
1    2
2    3
3    4
dtype: int32

>>> arr[2] = -2
>>> s3
0    1
1    2
2   -2
3    4
dtype: int32
```

上述例子，改动arr数组第三个元素的值，同时也会修改Series对象s3中相应的元素。

5. 筛选元素

pandas库的开发是以NumPy库为基础的，因此就数据结构而言，NumPy数组的多种操作方法得以扩展到Series对象中，其中就有根据条件筛选数据结构中的元素这一方法。

如要获取Series对象中所有大于8的元素，可使用以下代码：

```
>>> s[s > 8]
a    12
d     9
dtype: int64
```

6. Series对象运算和数学函数

适用于NumPy数组的运算符（+、-、*、/）或其他数学函数，也适用于Series对象。

至于运算符，直接用来编写算术表达式即可。

```
>>> s / 2
a    6.0
b   -2.0
c    3.5
d    4.5
dtype: float64
```

然而，至于NumPy库的数学函数，必须指定它们的出处np，并把Series实例作为参数传入。

```
>>> np.log(s)
a    2.484907
b         NaN
c    1.945910
d    2.197225
dtype: float64
```

7. Series对象的组成元素

Series对象往往包含重复的元素，你很可能想知道里面都包含哪些元素，统计元素重复出现的次数或判断一个元素是否在Series中。

我们来声明一个包含多个重复元素的Series对象。

```
>>> serd = pd.Series([1,0,2,1,2,3], index=['white','white','blue','green','green','yellow'])
>>> serd
white     1
white     0
blue      2
green     1
green     2
yellow    3
dtype: int64
```

要弄清楚Series对象包含多少个不同的元素，可使用unique()函数。其返回结果为一个数组，包含Series去重后的元素，但顺序看上去很随意。

```
>>> serd.unique()
array([1, 0, 2, 3], dtype=int64)
```

跟unique()函数相似的另外一个函数是value_counts()函数，它不仅返回各个不同的元素，还计算每个元素在Series中的出现次数。

```
>>> serd.value_counts()
2    2
1    2
3    1
0    1
dtype: int64
```

最后，isin()函数用来判断所属关系，也就是判断给定的一列元素是否包含在数据结构之中。isin()函数返回布尔值，可用于筛选Series或DataFrame列中的数据。

```
>>> serd.isin([0,3])
white     False
white      True
blue      False
green     False
green     False
yellow     True
dtype:     bool
>>> serd[serd.isin([0,3])]
white     0
yellow    3
dtype: int64
```

8. NaN

在前面的一个例子中，我们求负数的对数，得到的返回结果为NaN（Not a Number，非数值）。数据结构中若字段为空或者不符合数字的定义时，用这个特定的值来表示。

一般来讲，NaN值表示数据有问题，必须对其进行处理，尤其是在数据分析时。从某些数据源抽取数据时遇到了问题，甚至是数据源缺失数据，往往就会产生这类数据。进一步来讲，计算负数的对数，执行计算或函数时抛出异常等特定情况，也可能产生这类数据。后续章节会讲解NaN值的几种不同处理方法。

尽管NaN值是数据有问题才产生的，然而在pandas中是可以定义这种类型的数据，并把它添加到Series等数据结构中的。创建数据结构时，可为数组中元素缺失的项输入np.NaN。

```
>>> s2 = pd.Series([5,-3,np.NaN,14])
>>> s2
0    5
1   -3
2  NaN
3   14
```

isnull()和lnotnull()函数用来识别没有对应元素的索引时非常好用。

```
>>> s2.isnull( )
0    False
1    False
2     True
3    False
dtype: bool
>>> s2.notnull( )
0     True
1     True
2    False
3     True
dtype: bool
```

上述两个函数返回两个由布尔值组成的Series对象，其元素值是True还是False取决于原Series对象的元素是否为NaN。如果是NaN，isnull()函数返回值为True；反之，如果不是NaN，notnull()函数返回值为True。这两个函数可用作筛选条件。

```
>>> s2[s2.notnull( )]
0     5
1    -3
3    14
dtype: float64
>>> s2[s2.isnull( )]
2   NaN
dtype: float64
```

9. Series用作字典

我们还可以把Series对象当作字典（dict，dictionary）对象来用。定义Series对象时，就可以利用这种相似性。事实上，我们可以用事先定义好的字典来创建Series对象。

```
>>> mydict = {'red': 2000, 'blue': 1000, 'yellow': 500, 'orange': 1000}
```

```
>>> myseries = pd.Series(mydict)
blue      1000
orange    1000
red       2000
yellow     500
dtype: int64
```

上述例子中，索引数组用字典的键来填充，每个索引所对应的元素为用作索引的键在字典中对应的值。你还可以单独指定索引，pandas会控制字典的键和数组索引标签之间的相关性。如遇缺失值处，pandas就会为其添加NaN。

```
>>> colors = ['red','yellow','orange','blue','green']
>>> myseries = pd.Series(mydict, index=colors)
red       2000
yellow     500
orange    1000
blue      1000
green      NaN
dtype: float64
```

10. Series对象之间的运算

我们前面见识过Series对象和标量之间的数学运算，Series对象之间也可以进行这类运算，甚至标签也可以参与运算。

事实上，Series这种数据结构在运算时有一大优点，它能够通过识别标签对齐不一致的数据。在下面这个例子中，我们来求只有部分元素标签相同的两个Series对象之和。

```
>>> mydict2 = {'red':400,'yellow':1000,'black':700}
>>> myseries2 = pd.Series(mydict2)
>>> myseries + myseries2
black      NaN
blue       NaN
orange     NaN
green      NaN
red       2400
yellow    1500
dtype: float64
```

上述运算得到一个新Series对象，其中只对标签相同的元素求和。其他只属于任何一个Series对象的标签也被添加到新对象中，只不过它们的值均为NaN。

4.5.2　DataFrame 对象

DataFrame这种列表式数据结构跟工作表（最常见的是Excel工作表）极为相似，其设计初衷是将Series的使用场景由一维扩展到多维。DataFrame由按一定顺序排列的多列数据（见图4-2）组成，各列的数据类型可以有所不同（数值、字符串或布尔值等）。

DataFrame			
	columns		
index	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

图4-2 DataFrame数据结构

Series对象的Index数组存放有每个元素的标签，而DataFrame对象则有所不同，它有两个索引数组。第一个数组与行相关，它与Series的索引数组极为相似。每个标签与标签所在行的所有元素相关联。而第二个数组包含一系列列标签，每个标签与一列数据相关联。

DataFrame还可以理解为一个由Series组成的字典，其中每一列的名称为字典的键，形成DataFrame的列的Series作为字典的值。进一步来说，每个Series的所有元素映射到叫作Index的标签数组。

1. 定义DataFrame对象

新建DataFrame对象的最常用方法是传递一个dict对象给DataFrame()构造函数。dict对象以每一列的名称作为键，每个键都有一个数组作为值。

```
>>> data = {'color' : ['blue','green','yellow','red','white'],
                    'object' : ['ball','pen','pencil','paper','mug'],
                    'price' : [1.2,1.0,0.6,0.9,1.7]}

>>>frame = pd.DataFrame(data)
>>> frame
    color object price
0    blue   ball   1.2
1   green    pen   1.0
2  yellow pencil   0.6
3     red  paper   0.9
4   white    mug   1.7
```

如果用来创建DataFrame对象的dict对象包含一些用不到的数据，你可以只选择自己感兴趣的。在DataFrame构造函数中，用columns选项指定需要的列即可。新建的DataFrame各列顺序与你指定的列顺序一致，而与它们在字典中的顺序无关。

```
>>> frame2 = pd.DataFrame(data, columns=['object','price'])
>>> frame2
   object price
0    ball   1.2
1     pen   1.0
2  pencil   0.6
```

```
3   paper   0.9
4     mug   1.7
```

DataFrame对象跟Series一样，如果Index数组没有明确指定标签，pandas也会自动为其添加一列从0开始的数值作为索引。如果想用标签作为DataFrame的索引，则要把标签放到数组中，赋给index选项。

```
>>> frame2 = pd.DataFrame(data, index=['one','two','three','four','five'])
>>> frame2
        color object price
one      blue   ball   1.2
two     green    pen   1.0
three  yellow pencil   0.6
four      red  paper   0.9
five    white    mug   1.7
```

既已引入两个新选项index和columns，还可以想出一种定义DataFrame的新方法。我们不再使用dict对象，而是定义一个构造函数，指定三个参数，参数顺序如下：数据矩阵、index选项和columns选项。记得将存放有标签的数组赋给index选项，将存放有列名称的数组赋给columns选项。

从书中后续很多例子中，你会看到，要方便、快捷地创建包含数据的数组，可以使用np.arange(16).reshape((4,4))生成一个4×4型、包含数字0~15的矩阵。

```
>>> frame3 = pd.DataFrame(np.arange(16).reshape((4,4)),
...                       index=['red','blue','yellow','white'],
...                       columns=['ball','pen','pencil','paper'])
>>> frame3
        ball pen pencil paper
red        0   1      2     3
blue       4   5      6     7
yellow     8   9     10    11
white     12  13     14    15
```

2. 选取元素

如想知道DataFrame对象所有列的名称，在它上面调用columns属性即可。

```
>>> frame.columns
Index([u'colors', u'object', u'price'], dtype='object')
```

类似地，要获取索引列表，调用index属性即可。

```
>>> frame.index
Int64Index([0, 1, 2, 3, 4], dtype='int64')
```

如果要获取存储在数据结构中的元素，可以使用values属性获取所有元素。

```
>>> frame.values
array([['blue', 'ball', 1.2],
       ['green', 'pen', 1.0],
       ['yellow', 'pencil', 0.6],
       ['red', 'paper', 0.9],
       ['white', 'mug', 1.7]], dtype=object)
```

或者，如果想选择一列内容，把这一列的名称作为索引即可。

```
>>> frame['price']
0    1.2
1    1.0
2    0.6
3    0.9
4    1.7
Name: price, dtype: float64
```

如上所见，返回值为Series对象。另外一种方法是用列名称作为DataFrame实例的属性。

```
>>> frame.price
0    1.2
1    1.0
2    0.6
3    0.9
4    1.7
Name: price, dtype: float64
```

至于DataFrame中的行，用ix属性和行的索引值就能获取到。

```
>>> frame.ix[2]
color     yellow
object    pencil
price        0.6
Name: 2, dtype: object
```

返回结果同样是一个Series对象，其中列的名称已经变为索引数组的标签，而列中的元素变为Series的数据部分。

用一个数组指定多个索引值就能选取多行：

```
>>> frame.ix[[2,4]]
    color object price
2 yellow  pencil   0.6
4  white     mug   1.7
```

若要从DataFrame抽取一部分，可以用索引值选择你想要的行。事实上，你可以把一行看作DataFrame的一部分，通过指定索引范围来选取，其中这一行的索引作为起始索引值（下面例子中的0），下一行的索引作为结束索引（下面例子中的1）。

```
>>> frame[0:1]
  color object price
0  blue    ball  1.2
```

如上，返回结果为只包含一行数据的DataFrame对象。如需多行，必须扩展选择范围。

```
>>> frame[1:3]
    color object price
1   green    pen   1.0
2 yellow  pencil   0.6
```

最后，如要获取存储在DataFrame中的一个元素，需要依次指定元素所在的列名称、行的索引值或标签。

```
>>> frame['object'][3]
'paper'
```

3. 赋值

一旦你理解了组成DataFrame的各元素的获取方法，依照相同的逻辑就能增加或修改元素。

例如，前面讲过用index属性指定DataFrame结构中的索引数组，用columns属性指定包含列名称的行。你还可以用name属性为这两个二级结构指定标签，便于识别。

```
>>> frame.index.name = 'id'; frame.columns.name = 'item'
>>> frame
item  color object price
id
0      blue   ball   1.2
1     green    pen   1.0
2    yellow pencil   0.6
3       red  paper   0.9
4  white    mug   1.7
```

灵活程度非常高是pandas数据结构的一大优点。事实上，你可以在任何层级修改它们的内部结构。例如，执行添加一列新元素这类常见的操作。

添加列的方法很简单，指定DataFrame实例新列的名称，为其赋值即可。

```
>>> frame['new'] = 12
>>> frame
   colors object price new
0    blue   ball   1.2  12
1   green    pen   1.0  12
2  yellow pencil   0.6  12
3     red  paper   0.9  12
4   white    mug   1.7  12
```

从结果可以看出，DataFrame新增了名称为“new”的一列，它的各个元素均为12。

然而，如果想更新一列的内容，需要把一个数组赋给这一列。

```
>>> frame['new'] = [3.0,1.3,2.2,0.8,1.1]
>>> frame
    color object price  new
0    blue   ball   1.2  3.0
1   green    pen   1.0  1.3
2  yellow pencil   0.6  2.2
3     red  paper   0.9  0.8
4   white    mug   1.7  1.1
```

如果想更新某一列的全部数据，方法类似。例如借助np.arange()函数预先定义一个序列，用它更新某一列的所有元素。

为DataFrame的各列赋一个Series对象也可以创建DataFrame，例如使用np.arange()函数生成一个递增序列。

```
>>> ser = pd.Series(np.arange(5))
>>> ser
0    0
1    1
2    2
3    3
```

```
4    4
dtype: int32
>>> frame['new'] = ser
>>> frame
    color object price  new
0    blue   ball   1.2    0
1   green    pen   1.0    1
2  yellow pencil   0.6    2
3     red  paper   0.9    3
4   white    mug   1.7    4
```

最后，我们来看一下修改单个元素的方法：选择元素，为其赋新值即可。

```
>>> frame['price'][2] = 3.3
```

4. 元素的所属关系

前面你已经见过用isin()函数判断一组元素是否属于Series对象，其实该函数对DataFrame对象也适用。

```
>>> frame.isin([1.0,'pen'])
   color object  price
0  False  False  False
1  False   True   True
2  False  False  False
3  False  False  False
4  False  False  False
```

你得到了一个只包含布尔值的DataFrame对象，其中只有满足从属关系之处元素为True。如把上述返回结果作为条件，将得到一个新DataFrame，其中只包含满足条件的元素。

```
>>> frame[frame.isin([1.0,'pen'])]
  color object price
0   NaN    NaN   NaN
1   NaN    pen     1
2   NaN    NaN   NaN
3   NaN    NaN   NaN
4   NaN    NaN   NaN
```

5. 删除一列

如想删除一整列的所有数据，使用del命令。

```
>>> del frame['new']
>>> frame
   colors object price
0    blue   ball   1.2
1   green    pen   1.0
2  yellow pencil   0.6
3     red  paper   0.9
4   white    mug   1.7
```

6. 筛选

对于DataFrame对象，也可以通过指定条件筛选元素。例如，你想获取所有小于指定数字（比如12）的元素。

```
>>> frame[frame < 12]
        ball  pen pencil paper
red        0    1      2     3
blue       4    5      6     7
yellow     8    9     10    11
white    NaN  NaN    NaN   NaN
```

返回的DataFrame对象只包含所有小于12的数字，各元素的位置保持不变。其他不符合条件的元素被替换为NaN。

7. 用嵌套字典生成DataFrame对象

嵌套字典是Python广泛使用的数据结构，示例如下：

```
nestdict = { 'red': { 2012: 22, 2013: 33 },
                    'white': { 2011: 13, 2012: 22; 2013: 16},
                    'blue': {2011: 17, 2012: 27; 2013: 18}}
```

直接将这种数据结构作为参数传递给DataFrame()构造函数，pandas就会将外部的键解释成列名称，将内部的键解释为用作索引的标签。

解释嵌套结构时，可能并非所有位置都有相应的元素存在。pandas会用NaN填补缺失的元素。

```
>>> nestdict = {'red':{2012: 22, 2013: 33},
...             'white':{2011: 13, 2012: 22, 2013: 16},
...             'blue': {2011: 17, 2012: 27, 2013: 18}}
>>> frame2 = pd.DataFrame(nestdict)
>>> frame2
      blue red white
2011    17 NaN    13
2012    27  22    22
2013    18  33    16
```

8. DataFrame转置

处理列表数据时可能会用到转置操作（列变为行，行变为列）。pandas提供了一种很简单的转置方法。调用T属性就能得到DataFrame对象的转置形式。

```
>>> frame2.T
       2011  2012  2013
blue     17    27    18
red     NaN    22    33
white    13    22    16
```

4.5.3　Index对象

现在，你知道了Series、DataFrame对象以及它们的结构形式，对这些数据结构的特性也一定有所了解。事实上，它们在数据分析方面的大多数优秀特性都取决于完全整合到这些数据结构中的Index对象。

轴标签或其他用作轴名称的元数据就存储为Index对象。前面讲过如何把存储多个标签的数组转化为Index对象：指定构造函数的index选项。

```
>>> ser = pd.Series([5,0,3,8,4], index=['red','blue','yellow','white','green'])
>>> ser.index
Index([u'red', u'blue', u'yellow', u'white', u'green'], dtype='object')
```

跟pandas数据结构（Series和DataFrame）中其他元素不同的是，Index对象不可改变。声明后，它不能改变。不同数据结构共用Index对象时，该特性能够保证它的安全。

每个Index对象都有很多方法和属性，当你需要知道它们所包含的值时，这些方法和属性非常有用。

1. Index对象的方法

Index对象提供了几种方法，可用来获取数据结构索引的相关信息。例如，idmin()和idmax()函数分别返回索引值最小和最大的元素。

```
>>> ser.idxmin()
'red'
>>> ser.idxmax()
'green'
```

2. 含有重复标签的Index

到目前为止，我们见过的所有索引都是位于一个单独的数据结构中，且所有标签都是唯一的。虽然只有满足这个条件，很多函数才能运行，但是对pandas数据结构而言，这个条件并不是必需的。

我们来举一个例子，定义一个含有重复标签的Series。

```
>>> serd = pd.Series(range(6), index=['white','white','blue','green','green','yellow'])
>>> serd
white     0
white     1
blue      2
green     3
green     4
yellow    5
dtype: int64
```

从数据结构中选取元素时，如果一个标签对应多个元素，我们得到的将是一个Series对象而不是单个元素。

```
>>> serd['white']
white    0
white    1
dtype: int64
```

以上逻辑适用于索引中存在重复项的DataFrame，其返回结果为DataFrame对象。

数据结构很小时，识别索引的重复项很容易，但随着数据结构逐渐增大以后，难度也在增加。pandas的Index对象还有is_unique属性。调用该属性，就可以知道数据结构（Series和DataFrame）中是否存在重复的索引项。

```
>>> serd.index.is_unique
False
>>> frame.index.is_unique
True
```

4.6　索引对象的其他功能

与Python常用数据结构相比，pandas不仅利用了NumPy数组的高性能优势，还整合了索引机制。

最终事实证明，这样做颇有几分成效。事实上，虽然已有的动态数据结构极为灵活，但在结构中增加诸如标签这样的内部索引机制，为接下来及下一章要讲解的一系列必要操作，提供了更为简单、直接的执行方法。

这一节，我们来详细分析几种使用索引机制实现的基础操作。

- 更换索引
- 删除
- 对齐

4.6.1　更换索引

前面讲过，数据结构一旦声明，Index对象就不能改变。这么说一点也没错，但是执行更换索引操作就可以解决这个问题。

事实上，重新定义索引之后，我们就能够用现有的数据结构生成一个新的数据结构。

```
>>> ser = pd.Series([2,5,7,4], index=['one','two','three','four'])
>>> ser
one      2
two      5
three    7
four     4
dtype: int64
```

pandas的reindex()函数可更换Series对象的索引。它根据新标签序列，重新调整原Series的元素，生成一个新的Series对象。

更换索引时，可以调整索引序列中各标签的顺序，删除或增加新标签。如增加新标签，pandas会添加NaN作为其元素。

```
>>> ser.reindex(['three','four','five','one'])
three     7
four      4
five    NaN
one       2
dtype: float64
```

从返回结果可以看到，标签顺序全部调整过。删除了标签two及其元素，增加了新标签five。

然而，重新编制索引，定义所有的标签序列可能会很麻烦，对大型DataFrame来说更是如此。但是我们可以使用自动填充或插值方法。

为了更好地理解自动编制索引功能，我们先来定义以下Series对象。

```
>>> ser3 = pd.Series([1,5,6,3],index=[0,3,5,6])
>>> ser3
```

```
0    1
3    5
5    6
6    3
dtype: int64
```

刚定义的Series对象，其索引列并不完整而是缺失了几个值（1、2和4）。常见的需求为插值，以得到一个完整的序列。方法是用reindex()函数，method选项的值为ffill。此外，还需要指定索引值的范围。要指定一列0~5的值，参数为range(6)。

```
>>> ser3.reindex(range(6),method='ffill')
0    1
1    1
2    1
3    5
4    5
5    6
dtype: int64
```

由结果可见，新Series对象添加了原Series对象缺失的索引项。新插入的索引项，其元素为前面索引编号比它小的那一项的元素。所以我们看到索引项1、2的值为1，也就是索引项0的值。

如果你想用新插入索引后面的元素，需要使用bfill方法。

```
>>> ser3.reindex(range(6),method='bfill')
0    1
1    5
2    5
3    5
4    6
5    6
dtype: int64
```

用这种方法，索引项1和2的元素则为5，也就是索引项3的元素。

更换索引的概念可以由Series扩展到DataFrame，你不仅可以更换索引（行），还可以更换列，甚至更换两者。如前所述，我们可以增加行或列，但pandas用NaN弥补原数据结构中缺失的元素。

```
>>> frame.reindex(range(5), method='ffill',columns=['colors','price','new','object'])
   colors price  new  object
0    blue   1.2  NaN    ballpand
1   green   1.0  NaN     pen
2  yellow   0.6  NaN  pencil
3     red   0.9  NaN   paper
4   white   1.7  NaN     mug
```

4.6.2 删除

另外一种跟Index对象相关的操作是删除。因为索引和列名称有了标签作为标识，所以删除操作变得很简单。

pandas专门提供了一个用于删除操作的函数：drop()，它返回不包含已删除索引及其元素的

新对象。

举例来说，我们想从Series对象中删除一项。为此，我们先来定义一个含有四个元素的Series对象，其中各元素标签均不相同。

```
>>> ser = Series(np.arange(4.), index=['red','blue','yellow','white'])
>>> ser
red       0
blue      1
yellow    2
white     3
dtype: float64
```

假如我们想删除标签为yellow的这一项。用标签作为drop()函数的参数，就可以删除这一项。

```
>>> ser.drop('yellow')
red      0
blue     1
white    3
dtype: float64
```

传入一个由多个标签组成的数组，可以删除多项。

```
>>> ser.drop(['blue','white'])
red       0
yellow    2
dtype: float64
```

要删除DataFrame中的元素，需要指定元素两个轴的轴标签。我们通过具体的例子来看一下，首先声明一个DataFrame对象。

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                      index=['red','blue','yellow','white'],
...                      columns=['ball','pen','pencil','paper'])
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

传入行的索引可删除行。

```
>>> frame.drop(['blue','yellow'])
       ball  pen  pencil paper
red       0    1       2     3
white    12   13      14    15
```

要删除列，需要指定列的索引，但是还必须用axis选项指定从哪个轴删除元素。如按照列的方向删除，axis的值为1。

```
>>> frame.drop(['pen','pencil'],axis=1)
        Ball  paper
red        0      3
blue       4      7
yellow     8     11
```

```
white     12     15
```

4.6.3 算术和数据对齐

pandas能够将两个数据结构的索引对齐，这可能是与pandas数据结构索引对象有关的最强大的功能。这一点尤其体现在数据结构之间的算术运算上。参与运算的两个数据结构，其索引项顺序可能不一致，而且有的索引项可能只存在于一个数据结构中。

从下面几个例子中，你就会发现做算术运算时，pandas很擅长对齐不同数据结构的索引项。我们来举个例子，先来定义两个Series对象，分别指定两个不完全一致的标签数组。

```
>>> s1 = pd.Series([3,2,5,1],['white','yellow','green','blue'])
>>> s2 = pd.Series([1,4,7,2,1],['white','yellow','black','blue','brown'])
```

算术运算种类很多，我们考虑一下最简单的求和运算。刚定义的两个Series对象，有些标签两者都有，有些只属于其中一个对象。如果一个标签，两个Series对象都有，就把它们的元素相加，反之，标签也会显示在结果（新Series对象）中，只不过元素为NaN。

```
>>> s1 + s2
black    NaN
blue       3
brown    NaN
green    NaN
white      4
yellow     6
dtype: float64
```

DataFrame对象之间的运算，虽然看起来可能更复杂，但对齐规则相同，只不过行和列都要执行对齐操作。

```
>>> frame1 = pd.DataFrame(np.arange(16).reshape((4,4)),
...                       index=['red','blue','yellow','white'],
...                       columns=['ball','pen','pencil','paper'])
>>> frame2 = pd.DataFrame(np.arange(12).reshape((4,3)),
...                       index=['blue','green','white','yellow'],
...                       columns=['mug','pen','ball'])
>>> frame1
        Ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> frame2
        mug  pen  ball
blue      0    1     2
green     3    4     5
white     6    7     8
yellow    9   10    11
>>> frame1 + frame2
        ball  mug  paper pen pencil
blue       6  NaN    NaN   6    NaN
```

```
green    NaN  NaN    NaN NaN    NaN
red      NaN  NaN    NaN NaN    NaN
white     20  NaN    NaN  20    NaN
yellow    19  NaN    NaN  19    NaN
```

4.7　数据结构之间的运算

既然你已经熟悉了Series和DataFrame等数据结构，以及针对它们的多种基础操作，我们接下来讲解两种及以上数据结构之间的运算。

例如，上一节我们曾学过两个对象之间的算术运算。这一节，我们则对两种数据结构之间的运算进行深入探讨。

4.7.1　灵活的算术运算方法

前面刚学过可直接在pandas数据结构之间使用算术运算符。相同的运算还可以借助灵活的算术运算方法（flexible arithmetic methods）来完成。

- add()
- sub()
- div()
- mul()

这些函数的调用方法与数学运算符的使用方法不同。例如，两个DataFrame对象的求和运算，不再使用“frame1+frame2”这种格式，而是使用下面这种格式：

```
>>> frame1.add(frame2)
        ball  mug  paper  pen pencil
blue       6  NaN    NaN    6    NaN
green    NaN  NaN    NaN  NaN    NaN
red      NaN  NaN    NaN  NaN    NaN
white     20  NaN    NaN   20    NaN
yellow    19  NaN    NaN   19    NaN
```

如上所见，结果跟使用+运算符所得到的相同。你可能还注意到，如果两个DataFrame对象的索引和列名称差别很大，新得到的DataFrame对象将有很多元素为NaN。本章后面会讲到这类数据的处理方法。

4.7.2　DataFrame 和 Series 对象之间的运算

再次回到算术运算符，pandas允许参与运算的对象为不同的数据结构，比如DataFrame和Series。举例之前，先来定义两个不同的数据结构。

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                      index=['red','blue','yellow','white'],
...                      columns=['ball','pen','pencil','paper'])
>>> frame
```

```
        ball  pen pencil paper
red        0    1      2     3
blue       4    5      6     7
yellow     8    9     10    11
white     12   13     14    15
>>> ser = pd.Series(np.arange(4), index=['ball','pen','pencil','paper'])
>>> ser
ball      0
pen       1
pencil    2
paper     3
dtype: int32
```

定义数据结构时，我们特意让Series对象的索引和DataFrame对象的列名称保持一致。这样就可以直接对它们进行运算。

```
>>> frame - ser
        ball  pen pencil paper
red        0    0      0     0
blue       4    4      4     4
yellow     8    8      8     8
white     12   12     12    12
```

如上所见，DataFrame对象各元素分别减去了Series对象中索引与之相同的元素。DataFrame对象每一列的所有元素，无论对应哪一个索引项，都执行了减法操作。

如果一个索引项只存在于其中一个数据结构之中，则运算结果中会为该索引项生成一列，只不过该列的所有元素都是NaN。

```
>>> ser['mug'] = 9
>>> ser
ball      0
pen       1
pencil    2
paper     3
mug       9
dtype: int64
>>> frame - ser
        ball  mug paper pen pencil
red        0  NaN     0   0      0
blue       4  NaN     4   4      4
yellow     8  NaN     8   8      8
white     12  NaN    12  12     12
```

4.8 函数应用和映射

这一节将讲解pandas库函数。

4.8.1 操作元素的函数

pandas库以NumPy为基础，并对它的很多功能进行了扩展，以用来操作新数据结构Series和

DataFrame。通用函数（ufunc，universal function）就是经过扩展得到的功能，这类函数能够对数据结构中的元素进行操作，因此特别有用。

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                      index=['red','blue','yellow','white'],
...                      columns=['ball','pen','pencil','paper'])
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

例如，使用NumPy的np.sqrt()函数就能计算DataFrame对象每个元素的平方根。

```
>>> np.sqrt(frame)
             ball       pen    pencil     paper
red      0.000000  1.000000  1.414214  1.732051
blue     2.000000  2.236068  2.449490  2.645751
yellow   2.828427  3.000000  3.162278  3.316625
white    3.464102  3.605551  3.741657  3.872983
```

4.8.2　按行或列执行操作的函数

除了通用函数，用户还可以自己定义函数。需要注意的是这些函数对一维数组进行运算，返回结果为一个数值。例如，我们可以定义一个计算数组元素取值范围的lambda函数。

```
>>> f = lambda x: x.max() - x.min()
```

还可以用下面这种形式定义函数：

```
>>> def f(x):
...     return x.max() - x.min()
...
```

用apply()函数可以在DataFrame对象上调用刚定义的函数。

```
>>> frame.apply(f)
ball      12
pen       12
pencil    12
paper     12
dtype: int64
```

然而，每一列的运算结果为一个数值。如果你想用函数处理行而不是列，需将axis选项设置为1。

```
>>> frame.apply(f, axis=1)
red       3
blue      3
yellow    3
white     3
dtype: int64
```

apply()函数并不是一定要返回一个标量，它还可以返回Series对象，因而可以借助它同时执行多个函数。每调用一次函数，就会有两个或两个以上的返回结果。我们可以像下面这样指定一个函数。

```
>>> def f(x):
...     return pd.Series([x.min(), x.max()], index=['min','max'])
...
```

像之前一样，应用这个函数，但是返回结果不再是Series而是DataFrame对象，并且DataFrame对象的行数跟函数返回值的数量相等。

```
>>> frame.apply(f)
     ball  pen  pencil  paper
min     0    1       2      3
max    12   13      14     15
```


4.8.3 统计函数

数组的大多数统计函数对DataFrame对象依旧有效，因此没有必要使用apply()函数。例如，sum()和mean()函数分别用来计算DataFrame对象元素之和及它们的均值。

```
>>> frame.sum()
ball      24
pen       28
pencil    32
paper     36
dtype: int64
>>> frame.mean()
ball      6
pen       7
pencil    8
paper     9
dtype: float64
```

describe()函数能够计算多个统计量。

```
>>> frame.describe()
            ball        pen     pencil      paper
count   4.000000   4.000000   4.000000   4.000000
mean    6.000000   7.000000   8.000000   9.000000
std     5.163978   5.163978   5.163978   5.163978
min     0.000000   1.000000   2.000000   3.000000
25%     3.000000   4.000000   5.000000   6.000000
50%     6.000000   7.000000   8.000000   9.000000
75%     9.000000  10.000000  11.000000  12.000000
Max    12.000000  13.000000  14.000000  15.000000
```

4.9 排序和排位次

另外一种使用索引机制的基础操作是排序（sorting）。对数据进行排序通常为必要操作，因

此简化它的实现非常重要。pandas的sort_index()函数返回一个跟原对象元素相同但顺序不同的新对象。

首先看一下Series对象各项的排序方法。要排序的索引只有一列，因此操作很简单。

```
>>> ser = pd.Series([5,0,3,8,4], index=['red','blue','yellow','white','green'])
>>> ser
red       5
blue      0
yellow    3
white     8
green     4
dtype: int64fra
>>> ser.sort_index()
blue      0
green     4
red       5
white     8
yellow    3
dtype: int64
```

输出结果中，各元素按照以字母表顺序升序排列（A～Z）的标签进行排序。这是默认的排序方法，但若指定ascending选项，将其值置为False，则可按照降序排列。

```
>>> ser.sort_index(ascending=False)
yellow    3
white     8
red       5
green     4
blue      0
dtype: int64
```

对于DataFrame对象，可分别对两条轴中的任意一条进行排序。如果要根据索引对行进行排序，可依旧使用sort_index()函数，不用指定参数，前面已经讲过；如果要按列进行排序，则需要指定axis选项，其值为1。

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                      index=['red','blue','yellow','white'],
...                      columns=['ball','pen','pencil','paper'])
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> frame.sort_index()
        ball  pen  pencil  paper
blue       4    5       6      7
red        0    1       2      3
white     12   13      14     15
yellow     8    9      10     11
>>> frame.sort_index(axis=1)
        ball  paper  pen  pencil
```

```
red        0     3    1     2
blue       4     7    5     6
yellow     8    11    9    10
white     12    15   13    14
```

至此，我们已经讲解了根据索引进行排序的方法。但你往往还需要对数据结构中的元素进行排序，对于这个问题，Series和DataFrame对象有所不同，要区别对待。

对Series对象排序，使用order()函数。

```
>>> ser.order()
blue      0
yellow    3
green     4
red       5
white     8
dtype: int64
```

对DataFrame对象排序，使用前面用过的sort_index()函数，只不过要用by选项指定根据哪一列进行排序。

```
>>> frame.sort_index(by='pen')
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

如果要基于两列或更多的列进行排序，则把这些列的名称放到数组中，赋给by选项。

```
>>> frame.sort_index(by=['pen','pencil'])
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

排位次操作（ranking）跟排序操作紧密相关，该操作为序列的每个元素安排一个位次（初始值为1，依次加1），位次越靠前，所使用的数值越小。

```
>>> ser.rank()
red       4
blue      1
yellow    2
white     5
green     3
dtype: float64
```

我们还可以把数据在数据结构中的顺序（没有进行排序操作）作为它的位次。只要使用method选项把first赋给它即可。

```
>>> ser.rank(method='first')
red       4
blue      1
yellow    2
```

```
white     5
green     3
dtype: float64
```

默认位次使用升序。要按照降序排列，就把ascending选项的值置为False。

```
>>> ser.rank(ascending=False)
red       2
blue      5
yellow    4
white     1
green     3
dtype: float64
```

4.10　相关性和协方差

相关性（correlation）和协方差（covariance）是两个重要的统计量，pandas计算这两个量的函数分别是corr()和cov()。这两个量的计算通常涉及两个Series对象。

```
>>> seq2 = pd.Seri
es([3,4,3,4,5,4,3,2],['2006','2007','2008','2009','2010','2011','2012','2013'])
>>> seq = pd.Seri
es([1,2,3,4,4,3,2,1],['2006','2007','2008','2009','2010','2011','2012','2013'])
>>> seq.corr(seq2)
0.77459666924148329
>>> seq.cov(seq2)
0.8571428571428571
```

另外一种情况是，计算单个DataFrame对象的相关性和协方差，返回两个新DataFrame对象形式的矩阵。

```
>>> frame2 = DataFrame([[1,4,3,6],[4,5,6,1],[3,3,1,5],[4,1,6,4]],
...                    index=['red','blue','yellow','white'],
...                    columns=['ball','pen','pencil','paper'])
>>> frame2
        ball  pen  pencil  paper
red        1    4       3      6
blue       4    5       6      1
yellow     3    3       1      5
white      4    1       6      4

>>> frame2.corr()
            ball       pen     pencil      paper
ball    1.000000 -0.276026   0.577350 -0.763763
pen    -0.276026  1.000000 -0.079682 -0.361403
pencil  0.577350 -0.079682  1.000000 -0.692935
paper  -0.763763 -0.361403 -0.692935  1.000000
>>> frame2.cov()
            ball       pen     pencil      paper
ball    2.000000 -0.666667   2.000000 -2.333333
pen    -0.666667  2.916667 -0.333333 -1.333333
pencil  2.000000 -0.333333   6.000000 -3.666667
```

```
paper  -2.333333  -1.333333  -3.666667   4.666667
```

用corrwith()方法可以计算DataFrame对象的列或行与Series对象或其他DataFrame对象元素两两之间的相关性。

```
>>> serred       0
blue     1
yellow   2
white    3
green    9
dtype: float64
>>> frame2.corrwith(ser)
ball      0.730297
pen      -0.831522
pencil    0.210819
paper    -0.119523
dtype: float64
>>> frame2.corrwith(frame)
ball      0.730297
pen      -0.831522
pencil    0.210819
paper    -0.119523
dtype: float64
```

4.11 NaN数据

由前几节可知，补上缺失的数值很容易，它们在数据结构中用NaN来表示，以便于识别。在数据分析过程中，有些元素在某个数据结构中没有定义，这种情况很常见。

pandas意在更好地管理这种可能出现的情况。事实上，这一节我们将讲解缺失值的处理方法，这样许多问题就可以避免。比如，pandas库在计算各种描述性统计量时，其实没有考虑NaN值。

4.11.1 为元素赋NaN值

有时需要为数据结构中的元素赋NaN值，这时用NumPy的np.NaN（或np.nan）即可。

```
>>> ser = pd.Series([0,1,2,np.NaN,9], index=['red','blue','yellow','white','green'])
>>> ser
red        0
blue       1
yellow     2
white    NaN
green      9
dtype: float64
>>> ser['white'] = None
>>> ser
red        0
blue       1
yellow     2
white    NaN
green      9
dtype: float64
```

4.11.2　过滤NaN

数据分析过程中，有几种去除NaN的方法。然而，若要人工逐一删除NaN元素很麻烦，也很不安全，因为无法确保删除了所有的NaN。而dropna()函数可以帮你解决这个问题。

```
>>> ser.dropna()
red       0
blue      1
yellow    2
green     9
dtype: float64
```

另外一种方法是，用notnull()函数作为选取元素的条件，实现直接过滤。

```
>>> ser[ser.notnull()]
red       0
blue      1
yellow    2
green     9
dtype: float64
```

DataFrame处理起来要稍微复杂点。如果对这类对象使用dropna()方法，只要行或列有一个NaN元素，该行或列的全部元素都会被删除。

```
>>> frame3 = pd.DataFrame([[6,np.nan,6],[np.nan,np.nan,np.nan],[2,np.nan,5]],
...                       index = ['blue','green','red'],
...                       columns = ['ball','mug','pen'])

>>> frame3
       ball  mug  pen
blue      6  NaN    6
green   NaN  NaN  NaN
red       2  NaN    5
>>> frame3.dropna()
Empty DataFrame
Columns: [ball, mug, pen]
Index: []
```

因此，为了避免删除整行或整列，需使用how选项，指定其值为all，告知dropna()函数只删除所有元素均为NaN的行或列。

```
>>> frame3.dropna(how='all')
      ball  mug  pen
blue     6  NaN    6
red      2  NaN    5
```

4.11.3　为NaN元素填充其他值

删除NaN元素，可能会删除跟数据分析相关的其他数据，所以与其冒着风险去过滤NaN元素，不如用其他数值替代NaN。fillna()函数能够满足大多数需要。这个函数以替换NaN的元素作为参数。所有NaN可以替换为同一个元素，如下所示：

```
>>> frame3.fillna(0)
       ball  mug  pen
blue      6    0    6
green     0    0    0
red       2    0    5
```

或者，若要将不同列的NaN替换为不同的元素，依次指定列名称及要替换成的元素即可。

```
>>> frame3.fillna({'ball':1,'mug':0,'pen':99})
       ball  mug  pen
blue      6    0    6
green     1    0   99
red       2    0    5
```

4.12 等级索引和分级

等级索引（hierarchical indexing）是pandas的一个重要功能，单条轴可以有多级索引。你可以像操作两维结构那样处理多维数据。

举个简单的例子：创建包含两列索引的Series对象，也就是说，创建一个包含两层的数据结构。

```
>>> mser = pd.Series(np.random.rand(8),
...         index=[['white','white','white','blue','blue','red','red','red'],
...         ['up','down','right','up','down','up','down','left']])
>>> mser
white  up       0.461689
       down     0.643121
       right    0.956163
blue   up       0.728021
       down     0.813079
red    up       0.536433
       down     0.606161
       left     0.996686
dtype: float64

>>> mser.index
MultiIndex(levels=[[u'blue', u'red', u'white'], [u'down', u'left', u'right', u'up']],
           labels=[[2, 2, 2, 0, 0, 1, 1, 1], [3, 0, 2, 3, 0, 3, 0, 1]])ù
```

通过指定等级索引，二级元素的选取操作得以简化。

事实上，你可以选取第一列索引中某一索引项的元素，使用最经典的做法即可：

```
>>> mser['white']
up       0.461689
down     0.643121
right    0.956163
dtype: float64
```

或者像下面这样，选取第二列索引中某一索引项的元素：

```
>>> mser[:,'up']
white    0.461689
```

```
blue     0.728021
red      0.536433
dtype: float64
```

若要选取某一特定的元素，指定两个索引即可，很直观吧？

```
>>> mser['white','up']
0.46168915430531676
```

等级索引在调整数据形状和进行基于组的操作（比如创建数据透视表）方面起着非常重要的作用。例如，可以使用unstack()函数调整DataFrame中的数据。这个函数把使用的等级索引Series对象转换为一个简单的DataFrame对象，其中把第二列索引转换为相应的列。

```
>>> mser.unstack()
          down      left     right        up
blue   0.813079       NaN       NaN  0.728021
red    0.606161  0.996686       NaN  0.536433
white  0.643121       NaN  0.956163  0.461689
```

如果想进行逆操作，把DataFrame对象转换为Series对象，可使用stack()函数。

```
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> frame.stack()
red     ball       0
        pen        1
        pencil     2
        paper      3
blue    ball       4
        pen        5
        pencil     6
        paper      7
yellow  ball       8
        pen        9
        pencil    10
        paper     11
white   ball      12
        pen       13
        pencil    14
        paper     15
dtype: int32
```

对于DataFrame对象，可以为它的行和列都定义等级索引。声明DataFrame对象时，为index选项和columns选项分别指定一个元素为数组的数组。

```
>>> mframe = pd.DataFrame(np.random.randn(16).reshape(4,4),
...       index=[['white','white','red','red'], ['up','down','up','down']],
...       columns=[['pen','pen','paper','paper'],[1,2,1,2]])
>>> mframe
                pen                paper
                  1         2         1         2
```

```
white up   -1.964055  1.312100 -0.914750 -0.941930
      down -1.886825  1.700858 -1.060846 -0.197669
red   up   -1.561761  1.225509 -0.244772  0.345843
      down  2.668155  0.528971 -1.633708  0.921735
```

4.12.1 重新调整顺序和为层级排序

有时，你需要调整某一条轴上各层级的顺序或者调整某一层中各元素的顺序。

swaplevel()函数以要互换位置的两个层级的名称为参数，返回交换位置后的一个新对象，其中各元素的顺序保持不变。

```
>>> mframe.columns.names = ['objects','id']
>>> mframe.index.names = ['colors','status']
>>> mframe
objects             pen               paper
id                    1         2         1         2
colors status
white  up     -1.964055  1.312100 -0.914750 -0.941930
       down   -1.886825  1.700858 -1.060846 -0.197669
red    up     -1.561761  1.225509 -0.244772  0.345843
       down    2.668155  0.528971 -1.633708  0.921735

>>> mframe.swaplevel('colors','status')
objects             pen               paper
id                    1         2         1         2
status colors
up     white  -1.964055  1.312100 -0.914750 -0.941930
down   white  -1.886825  1.700858 -1.060846 -0.197669
up     red    -1.561761  1.225509 -0.244772  0.345843
down   red     2.668155  0.528971 -1.633708  0.921735
```

而sortlevel()函数只根据一个层级对数据排序。

```
>>> mframe.sortlevel('colors')
objects             pen               paper
id                    1         2         1         2
colors status
red    down    2.668155  0.528971 -1.633708  0.921735
       up     -1.561761  1.225509 -0.244772  0.345843
white  down   -1.886825  1.700858 -1.060846 -0.197669
       up     -1.964055  1.312100 -0.914750 -0.941930
```

4.12.2 按层级统计数据

DataFrame或Series对象的很多描述性和概括统计量都有level选项，可用它指定要获取哪个层级的描述性和概括统计量。

例如，你想对行一层级进行统计，把层级的名称赋给level选项即可。

```
>>> mframe.sum(level='colors')
objects       pen               paper
```

```
id              1         2         1         2
colors
red      1.106394  1.754480 -1.878480  1.267578
white   -3.850881  3.012959 -1.975596 -1.139599
```

若想对某一层级的列进行统计，例如id，则需要把axis选项的值设置为1，把第二条轴作为参数。

```
>>> mframe.sum(level='id', axis=1)
id                    1         2
colors status
white  up     -2.878806  0.370170
       down   -2.947672  1.503189
red    up     -1.806532  1.571352
       down    1.034447  1.450706
```

4.13　小结

本章介绍了pandas库，你从中学到了它的安装方法，并且对它的特点有了全面的认识。

想必你对它的两种基础数据结构Series和DataFrame，以及它们的操作方法和主要特点，也有了较为详细的了解。尤其是，你已经知道了这两种结构索引机制的重要性以及它们的最佳操作方法。最后，你还学习了通过创建层级索引扩展这两种数据结构，以按照不同的次级层级分别处理其中的数据。

下一章将学习从文件等外部数据源获取数据，以及把分析结果写入文件的方法。

第5章 pandas：数据读写

通过上一章的学习，你熟悉了pandas库以及它所提供的用于数据分析的基础功能，也知道了DataFrame和Series是这个库的核心，数据处理、计算和分析都是围绕它们展开的。

本章将学习pandas从多种存储媒介（比如文件和数据库）读取数据的工具，还将学到直接将不同的数据结构写入不同格式文件的方法，而无需过多考虑所使用的技术。

本章的主要内容为pandas的多种I/O API函数，它们为把大多数常用格式的数据作为DataFrame对象进行读写提供了很大便利。你首先会学到文本文件的读写，随后再逐步过渡到更加复杂的二进制文件。

本章最后将讲解SQL和NoSQL常用数据库的连接方法，我们用几个例子来说明如何直接把DataFrame中的数据存储到数据库中。同时，我们还会介绍如何从数据库读取数据，存储为DataFrame对象，并对其进行检索。

5.1 I/O API 工具

pandas是数据分析专用库，因此如你所料，它主要关注的是数据计算和处理。此外，从外部文件读写数据也被视作数据处理的一部分。实际上正如后面会讲到的，即使在这个阶段，你也可以对数据做一定的处理，以为接下来对数据做进一步分析做准备。

因此，数据读写对数据分析很重要，于是pandas库必须得为此提供专门的工具——一组被称为I/O API的函数。这些函数分为完全对称的两大类：*读取函数和写入函数*。

读取函数	写入函数
read_csv	to_csv
read_excel	to_excel
read_hdf	to_hdf
read_sql	to_sql
read_json	to_json
read_html	to_html
read_stata	to_stata
read_clipboard	to_clipboard
read_pickle	to_pickle

（续）

读取函数	写入函数
read_msgpack	to_msgpack（带实验性质）
read_gbq	to_gbq（带实验性质）

5.2　CSV和文本文件

多年以来，人们已习惯于文本文件的读写，特别是列表形式的数据。如果文件每一行的多个元素是用逗号隔开的，则这种格式叫作CSV，这可能是最广为人知和最受欢迎的格式。

其他由空格或制表符分隔的列表数据通常存储在各种类型的文本文件中（扩展名一般为.txt）。

因此这种文件类型是最常见的数据源，它易于转录和解释。pandas的下列函数专门用来处理这种文件类型：

- read_csv
- read_table
- to_csv

5.3　读取CSV或文本文件中的数据

根据一般经验，对数据分析人员来说，最常执行的操作是从CSV文件或其他类型的文本文件中读取数据。

为了弄清楚pandas处理这类数据的方法，我们在工作目录下创建一个短小的CSV文件，将其保存为myCSV_01.csv，如代码清单5-1所示。

代码清单5-1　myCSV_01.csv

```
white,red,blue,green,animal
1,5,2,3,cat
2,7,8,5,dog
3,3,6,7,horse
2,2,8,3,duck
4,4,2,1,mouse
```

这个文件以逗号作为分隔符，因此可以使用read_csv()函数读取它的内容，同时将其转换为DataFrame对象。

```
>>> csvframe = read_csv('myCSV_01.csv')
>>> csvframe
   white  red  blue  green animal
0      1    5     2      3    cat
1      2    7     8      5    dog
2      3    3     6      7  horse
3      2    2     8      3   duck
4      4    4     2      1  mouse
```

如你所见，读取CSV文件中的数据很简单。CSV文件中的数据为列表数据，位于不同列的元素用逗号隔开。但是既然CSV文件被视作文本文件，你还可以使用read_table()函数，但是得指定分隔符。

```
>>> read_table('ch05_01.csv',sep=',')
   white  red  blue  green  animal
0      1    5     2      3     cat
1      2    7     8      5     dog
2      3    3     6      7   horse
3      2    2     8      3    duck
4      4    4     2      1   mouse
```

从上述例子可知，标识各列名称的表头位于CSV文件的第一行，但一般情况并非如此，往往CSV文件的第一行就是列表数据（见代码清单5-2）。

代码清单5-2 myCSV_02.csv

```
1,5,2,3,cat
2,7,8,5,dog
3,3,6,7,horse
2,2,8,3,duck
4,4,2,1,mouse
>>> read_csv('ch05_02.csv')

   1  5  2  3    cat
0  2  7  8  5    dog
1  3  3  6  7  horse
2  2  2  8  3   duck
3  4  4  2  1  mouse
```

对于没有表头这种情况，使用header选项，将其值置为None，pandas会为其添加默认表头。

```
>>> read_csv('ch05_02.csv', header=None)
   0  1  2  3      4
0  1  5  2  3    cat
1  2  7  8  5    dog
2  3  3  6  7  horse
3  2  2  8  3   duck
4  4  4  2  1  mouse
```

此外，还可以使用names选项指定表头，直接把存有各列名称的数组赋给它即可。

```
>>> read_csv('ch05_02.csv', names=['white','red','blue','green','animal'])
   white  red  blue  green animal
0      1    5     2      3    cat
1      2    7     8      5    dog
2      3    3     6      7  horse
3      2    2     8      3   duck
4      4    4     2      1  mouse
```

再来说一下更为复杂的情况，假如你想读取CSV文件，创建一个具有等级结构的DataFrame对象。为此，可以添加index_col选项，扩展read_csv()函数的功能，把所有想转换为索引的列名称赋给index_col。

为了更好地理解这种可能性，新建一个CSV文件，其中有两列将用作等级索引。然后，将其保存到工作目录，文件名为myCSV_03.csv（见代码清单5-3）。

代码清单5-3　myCSV_03.csv

```
color,status,item1,item2,item3
black,up,3,4,6
black,down,2,6,7
white,up,5,5,5
white,down,3,3,2
white,left,1,2,1
red,up,2,2,2
red,down,1,1,4

>>> read_csv('ch05_03.csv', index_col=['color','status'])
              item1  item2  item3
color status
black up          3      4      6
      down        2      6      7
white up          5      5      5
      down        3      3      2
      left        1      2      1
red   up          2      2      2
      down        1      1      4
```

5.3.1　用 RegExp 解析 TXT 文件

有时要解析的数据文件不是以逗号或分号分隔的。对于这种情况，正则表达式就能派上用场。可以使用sep选项指定正则表达式，在read_table()函数内使用。

为了更好地理解正则表达式的用法，以及用它分隔多个元素的方法，我们可以从一个简单的例子入手。假如你有一个文件，比如是TXT文件，它里面的元素是以空格或制表符分隔的，且没有规律可言。在这种情况下，只有用正则表达式才能兼顾两种分隔符。可以使用通配符\s*。\s匹配空格或制表符（若只匹配制表符，可使用\t），星号表示这些字符可能有多个（其他最常用的通配符请见表5-1），也就是说，相邻的元素是由多个空格或制表符隔开的。

表5-1　元字符

.	换行符以外的单个字符
\d	数字
\D	非数字字符
\s	空白字符
\S	非空白字符
\n	换行符
\t	制表符
\uxxxx	用十六进制数字xxxx表示的Unicode字符

我们来举一个稍微极端点的例子，所有元素随机以制表符或空格分隔，顺序随机（见代码清单5-4）。

代码清单5-4 ch05_04.txt

```
white red blue green
    1   5    2     3
    2   7    8     5
    3   3    6     7

>>> read_table('ch05_04.txt',sep='\s*')
   white  red  blue  green
0      1    5     2      3
1      2    7     8      5
2      3    3     6      7
```

如上所见，我们得到了一个完美的DataFrame对象，其中所有元素均处在正确的位置。

接下来这个例子看起来有点奇怪或不寻常，但其实并不罕见。该例子非常有助于理解正则表达式潜在的强大功能。你通常把逗号、空格和制表符等看作分隔符，但实际应用中，字母数字组合或者整数均可用作分隔符，比如数字0。

在接下来这个例子中，TXT文件中数字和字母杂糅在一起，你需要从中抽取数字部分。

若TXT文件中的数据无表头，记得将header选项置为None（见代码清单 5-5）。

代码清单5-5 ch05_05.txt

```
000END123AAA122
001END124BBB321
002END125CCC333

>>> read_table('ch05_05.txt',sep='\D*',header=None)
   0    1    2
0  0  123  122
1  1  124  321
2  2  125  333
```

另一种很常见的情况是，解析数据时把空行排除在外。文件中的表头或没有必要的注释，有时用不到（见代码清单5-6）。使用skiprows选项，可以排除多余的行。把要排除的行的行号放到数组中，赋给该选项即可。

使用该选项时，需要注意一点。如要排除前五行，需要这样写：skiprows = 5；如只是排除第五行，写作skiprows = [5]。

代码清单5-6 ch05_06.txt

```
############ LOG FILE #############
This file has been generated by automatic system
white,red,blue,green,animal
12-Feb-2015: Counting of animals inside the house
1,5,2,3,cat
2,7,8,5,dog
```

```
13-Feb-2015: Counting of animals outside the house
3,3,6,7,horse
2,2,8,3,duck
4,4,2,1,mouse

>>> read_table('ch05_06.txt',sep=',',skiprows=[0,1,3,6])
   white  red blue green animal
0      1    5    2     3    cat
1      2    7    8     5    dog
2      3    3    6     7  horse
3      2    2    8     3   duck
4      4    4    2     1  mouse
```

5.3.2　从 TXT 文件读取部分数据

处理大文件或是只对文件部分数据感兴趣时，往往需要按照部分（块）读取文件，因为只需要部分数据。这两种情况都得使用迭代。

举例来说，假如只想读取文件的一部分，可明确指定要解析的行号，这时要用到nrows和skiprows选项。你可以指定起始行n（n = SkipRows）和从起始行往后读多少行（nrows = i）。

```
>>> read_csv('ch05_02.csv',skiprows=[2],nrows=3,header=None)
   0  1  2  3      4
0  1  5  2  3    cat
1  2  7  8  5    dog
2  2  2  8  3   duck
```

另外一项既有趣又很常用的操作是切分想要解析的文本，然后遍历各个部分，逐一对其执行某一特定操作。

例如，对于一列数字，每隔两行取一个累加起来，最后把和插入到Series对象中。这个小例子理解起来很简单，也没有实际应用价值，但是一旦领会了其原理，你就能将其用到更加复杂的情况。

```
>>> out = Series()
>>> i = 0
>>> pieces = read_csv('ch05_01.csv',chunksize=3)
>>> for piece in pieces:
...     out.set_value(i,piece['white'].sum())
...     i = i + 1
...
0    6
dtype: int64
0    6
1    6
dtype: int64
>>> out
0    6
1    6
dtype: int64
```

5.3.3 往CSV文件写入数据

从文件读取数据很常用，把计算结果或数据结构所包含的数据写入数据文件也是常用的必要操作。

例如，你可能想把DataFrame中的数据写入CSV文件。在写入过程中，就要用到to_csv()函数，其参数为即将生成的文件名（见代码清单5-7）。

```
>>> frame2
ball  pen  pencil  paper
0        1       2       3
4        5       6       7
8        9      10      11
12      13      14      15
>>> frame2.to_csv('ch05_07.csv')
```

代码清单5-7 ch05_07.csv

```
ball,pen,pencil,paper
0,1,2,3
4,5,6,7
8,9,10,11
12,13,14,15
```

由上述例子可知，把DataFrame写入文件时，索引和列名称连同数据一起写入。使用index和header选项，把它们的值设置为False，可取消这一默认行为（见代码清单5-8）。

```
>>> frame2.to_csv('ch05_07b.csv', index=False, header=False)
```

代码清单5-8 ch05_08.csv

```
1,2,3
5,6,7
9,10,11
13,14,15
```

需要注意的是，数据结构中的NaN写入文件后，显示为空字段（见代码清单5-9）。

```
>>> frame3
        ball  mug  paper  pen  pencil
blue       6  NaN    NaN    6     NaN
green    NaN  NaN    NaN  NaN     NaN
red      NaN  NaN    NaN  NaN     NaN
white     20  NaN    NaN   20     NaN
yellow    19  NaN    NaN   19     NaN
>>> frame3.to_csv('ch05_08.csv')
```

代码清单5-9 ch05_09.csv

```
,ball,mug,paper,pen,pencil
blue,6.0,,,6.0,
green,,,,,
red,,,,,
```

```
white,20.0,,,20.0,
yellow,19.0,,,19.0,
```

但是你可以用to_csv()函数的na_rep选项把空字段替换为你需要的值。常用值有NULL、0和NaN（见代码清单5-10）。

```
>>> frame3.to_csv('ch05_09.csv', na_rep ='NaN')
```

代码清单5-10　ch05_10.csv

```
,ball,mug,paper,pen,pencil
blue,6.0,NaN,NaN,6.0,NaN
green,NaN,NaN,NaN,NaN,NaN
red,NaN,NaN,NaN,NaN,NaN
white,20.0,NaN,NaN,20.0,NaN
yellow,19.0,NaN,NaN,19.0,NaN
```

注意　在上述几个例子中，DataFrame一直是我们讨论的主题，因为通常需要将这种数据结构写入文件。但是，所有这些函数和选项也适用于Series。

5.4　读写 HTML 文件

pandas提供以下I/O API函数用于读写HTML格式的文件：

- read_html()
- to_html()

这两个函数非常有用。把DataFrame等复杂的数据结构转换为HTML表格很简单，无需编写一长串HTML代码就能实现。pandas这方面的能力很强大，如果你从事Web开发，该功能将给你带来很多便捷。

逆操作也很有用，因为如今主要的数据源为因特网。网上的很多数据并不总是拿来就能用的，它们不是存储在TXT或CSV文件中。这些数据是网页文本的一部分，因此实现一个读取网页数据的函数非常有必要。

读取网页数据这种操作被称为网页抓取，应用极广。它逐渐演变成数据分析过程中的一项基础操作，被整合到了数据分析的第一步——数据挖掘和数据准备。

注意　如今很多网站为避免模块缺失和错误信息，已采用HTML5格式。我强烈建议你安装html5lib模块。若使用Anaconda，安装命令为：

```
conda install html5lib
```

5.4.1 写入数据到 HTML 文件

现在我们来学习把DataFrame转换为HTML表格的方法。DataFrame的内部结构被自动转换为嵌入在表格中的<TH>、<TR>、<TD>标签，保留所有内部层级结构。使用该函数，无需了解HTML知识。

因为有时DataFrame等数据结构很复杂，规模很大，所以对需要开发网页的人来说，往HTML文件中写入数据的函数用处很大。

为了更好地理解它的功能，我们来举个例子。先定义一个简单的DataFrame对象。

to_html()函数可以直接把DataFrame转换为HTML表格。

```
>>> frame = pd.DataFrame(np.arange(4).reshape(2,2))
```

I/O API函数是在pandas数据结构内部定义的，因此可以直接在DataFrame实例上调用to_html()函数。

```
>>> print(frame.to_html())
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
</thead>

  <tbody>
    <tr>
      <th>0</th>
      <td> 0</td>
      <td> 1</td>
    </tr>
    <tr>
      <th>1</th>
      <td> 2</td>
      <td> 3</td>
    </tr>
  </tbody>
</table>
```

如上所见，该函数按照DataFrame的内部结构，正确生成了创建HTML表格所需的HTML标签。

下面的例子演示如何在HTML文件中自动生成表格。我们创建一个比上面更加复杂、具有索引和列名称的DataFrame对象。

```
>>> frame = pd.DataFrame( np.random.random((4,4)),
...                       index = ['white','black','red','blue'],
...                       columns = ['up','down','right','left'])
>>> frame
             up      down     right      left
white  0.292434  0.457176  0.905139  0.737622
black  0.794233  0.949371  0.540191  0.367835
```

```
red    0.204529  0.981573  0.118329  0.761552
blue   0.628790  0.585922  0.039153  0.461598
```

现在，请把注意力放到如何生成一个字符串并把它写入到HTML页面上。这个例子虽然短小，但是可以帮助你直接在Web浏览器中理解和测试pandas的功能。

首先创建一个包含HTML页面代码的字符串：

```
>>> s = ['<HTML>']
>>> s.append('<HEAD><TITLE>My DataFrame</TITLE></HEAD>')
>>> s.append('<BODY>')
>>> s.append(frame.to_html())
>>> s.append('</BODY></HTML>')
>>> html = ''.join(s)
```

既然HTML页面的所有内容都存储在变量html中，你可以直接把它写入到myFrame.html文件中：

```
>>> html_file = open('myFrame.html','w')
>>> html_file.write(html)
>>> html_file.close()
```

现在，工作目录中多了myFrame.html文件。双击直接用浏览器打开它，将会看到HTML表格显示在网页的左上方，如图5-1所示。

	up	down	right	left
white	0.292434	0.457176	0.905139	0.737622
black	0.794233	0.949371	0.540191	0.367835
red	0.204529	0.981573	0.118329	0.761552
blue	0.628790	0.585922	0.039153	0.461598

图5-1　DataFrame转换为网页的HTML表格

5.4.2　从 HTML 文件读取数据

如上所见，pandas可以直接用DataFrame生成HTML表格。逆操作也很简单；read_html()函数解析HTML页面，寻找HTML表格。如果找到，就将其转换为可以直接用于数据分析的DataFrame对象。

更精确地讲，即使只有一个表格，read_html()函数也会返回一个DataFrame列表。至于要解析的数据源，可以是多种类型。例如，你可能需要读取任意目录中的HTML文件。比如，解析上个例子中创建的HTML文件。

```
>>> web_frames = pd.read_html('myFrame.html')
>>> web_frames[0]
  Unnamed: 0        up      down     right      left
0      white  0.292434  0.457176  0.905139  0.737622
1      black  0.794233  0.949371  0.540191  0.367835
2        red  0.204529  0.981573  0.118329  0.761552
3       blue  0.628790  0.585922  0.039153  0.461598
```

如上所见，所有跟HTML表格无关的标签都没有考虑在内。进一步讲，web_frames是一个元素为DataFrame的列表，虽然在这个例子中，你只抽取了一个表格。要从列表中选择我们想使用的DataFrame，可用传统的索引方法。由于这里只有一个元素，因此索引为0。

然而，read_html()函数最常用的模式是以网址作为参数，直接解析并抽取网页中的表格。

举个例子，下面的网址所指向页面的HTML表格为一排行榜，包含用户名字和得分两项。你可以直接以这个地址为参数进行处理。

```
>>> ranking = pd.read_html('http://www.meccanismocomplesso.org/en/
meccanismo-complesso-sito-2/classifica-punteggio/')
>>> ranking[0]
    Member        points  levels  Unnamed: 3
0        1   BrunoOrsini    1075         NaN
1        2     Berserker     700         NaN
2        3  albertosallu     275         NaN
3        4          Mr.Y     180         NaN
4        5           Jon     170         NaN
5        6  michele sisi     120         NaN
6        7   STEFANO GUST     120         NaN
7        8   Davide Alois     105         NaN
8        9   Cecilia Lala     105         NaN
...
```

上述操作适用于解析任意一个包含一个或多个表格的网页。

5.5 从 XML 读取数据

pandas的所有I/O API函数中，没有专门用来处理XML（可扩展标记语言）格式的。虽然没有，但这种格式其实很重要，因为很多结构化数据都是以XML格式存储的。pandas没有专门的处理函数也没关系，因为Python有很多读写XML格式数据的库（除了pandas）。

其中一个库叫作lxml，它在大文件处理方面性能优异，因而从众多同类库之中脱颖而出。这一节将介绍如何用它处理XML文件，以及如何把它和pandas整合起来，以最终从XML文件中获取到所需数据并将其转换为DataFrame对象。要想获得关于这个库的更多信息，我强烈建议你访问lxml的官方网站：http://lxml.de/index.html。

以代码清单5-11的XML文件为例。新建books.xml文件，写入下述代码，并将其保存到工作目录下。

代码清单5-11 books.xml

```
<?xml version="1.0"?>
<Catalog>
   <Book id="ISBN9872122367564">
      272103_1_EnRoss, Mark</Author>
      <Title>XML Cookbook</Title>
      <Genre>Computer</Genre>
      <Price>23.56</Price>
      <PublishDate>2014-22-01</PublishDate>
```

```
    </Book>
    <Book id="ISBN9872122367564">
        272103_1_EnBracket, Barbara</Author>
        <Title>XML for Dummies</Title>
        <Genre>Computer</Genre>
        <Price>35.95</Price>
        <PublishDate>2014-12-16</PublishDate>
    </Book>
</Catalog>
```

在这个例子中，你需要直接把XML中的数据结构转换为DataFrame对象。要完成该操作，首先要用到lxml库的二级模块objectify，导入方法如下：

```
>>> from lxml import objectify
```

现在就可以用parse()函数解析XML文件了。

```
>>> xml = objectify.parse('books.xml')
>>> xml
<lxml.etree._ElementTree object at 0x0000000009734E08>
You got an object tree, which is an internal data structure of the module lxml.
Look in more detail at this type of object. To navigate in this tree structure, so as to select
element by element, you must first define the root. You can do this with the getroot() function.
>>> root = xml.getroot()
```

既然已定义了根结构，你就可以获取到树结构的各个节点，每个节点与原始的XML文件中的标签相对应。节点的名称跟标签名称相同。因此，要选择节点，只需依次指定几个标签。注意各标签之间用点号分隔，标签的次序反应的正是树中节点的层级顺序。

```
>>> root.Book.Author
'Ross, Mark'
>>> root.Book.PublishDate
'2014-22-01'
```

这样，你可以获取单个节点。若要同时获取多个元素，可以使用getchildren()函数，它能获取某个元素的所有子节点。

```
>>> root.getchildren()
[<Element Book at 0x9c66688>, <Element Book at 0x9c66e08>]
```

再用tag属性，就能获取到子节点tag属性的名称。

```
>>> [child.tag for child in root.Book.getchildren()]
['Author', 'Title', 'Genre', 'Price', 'PublishDate']
```

用text属性，可获取到位于标签之间的内容。

```
>>> [child.text for child in root.Book.getchildren()]
['Ross, Mark', 'XML Cookbook', 'Computer', '23.56', '2014-22-01']
```

你知道我们能够遍历lxml.etree树结构就行，接下来需要做的是把树结构转换为DataFrame对象。定义以下函数，分析eTree的所有内容，逐行填充DataFrame对象。

```
>>> def etree2df(root):
...     column_names = []
```

```
...     for i in range(0,len(root.getchildren()[0].getchildren())):
...         column_names.append(root.getchildren()[0].getchildren()[i].tag)
...     xml:frame = pd.DataFrame(columns=column_names)
...     for j in range(0, len(root.getchildren())):
...         obj = root.getchildren()[j].getchildren()
...         texts = []
...         for k in range(0, len(column_names)):
...             texts.append(obj[k].text)
...         row = dict(zip(column_names, texts))

...         row_s = pd.Series(row)
...         row_s.name = j
...         xml:frame = xml:frame.append(row_s)
...     return xml:frame
...
>>> etree2df(root)
             Author           Title    Genre Price PublishDate
0        Ross, Mark    XML Cookbook Computer 23.56  2014-22-01
1  Bracket, Barbara XML for Dummies Computer 35.95  2014-12-16
```

5.6 读写 Microsoft Excel 文件

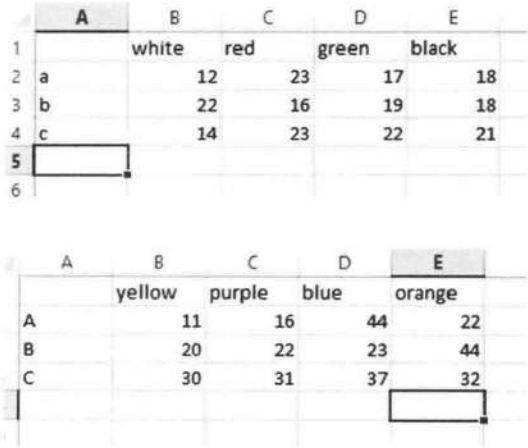
由上节可知，从CSV文件读取数据的操作很简单。除了CSV文件，用Excel工作表存放列表形式的数据也很常见。

pandas专门定义了几个函数来处理这种格式。前面给出的I/O API函数中，有两个是专门用于Excel文件的：

- to_excel()
- read_excel()

read_excel()函数能够读取Excel 2003（.xls）和Excel 2007（.xlsx）两种类型的文件。该函数之所以能够读取Excel，是因为它整合了xlrd模块。

首先，打开一个Excel文件，在sheet1和sheet2中输入图5-2中的数据。然后将其保存为data.xls。



	A	B	C	D	E
1		white	red	green	black
2	a	12	23	17	18
3	b	22	16	19	18
4	c	14	23	22	21
5					
6					

	A	B	C	D	E
		yellow	purple	blue	orange
A		11	16	44	22
B		20	22	23	44
C		30	31	37	32

图5-2　Excel文件sheet1和sheet2中的两组数据

要读取XLS文件中的数据，并将其转换为DataFrame对象，只需要使用read_excel()函数。

```
>>> pd.read_excel('data.xls')
   white  red  green  black
a     12   23     17     18
b     22   16     19     18
c     14   23     22     21
```

如上所见，读取Excel时，默认返回的DataFrame对象包含第一个工作表中的数据。若要读取第二个工作表中的数据，需要用第二个参数指定工作表的名称或工作表的序号（索引）。

```
>>> pd.read_excel('data.xls','Sheet2')
   yellow  purple  blue  orange
A      11      16    44      22
B      20      22    23      44
C      30      31    37      32
>>> pd.read_excel('data.xls',1)
   yellow  purple  blue  orange
A      11      16    44      22
B      20      22    23      44
C      30      31    37      32
```

上述操作也适用于Excel写操作。因此要将DataFrame对象转换为Excel，代码如下：

```
>>> frame = pd.DataFrame(np.random.random((4,4)),
...                      index = ['exp1','exp2','exp3','exp4'],
...                      columns = ['Jan2015','Fab2015','Mar2015','Apr2005'])
>>> frame
      Jan2015   Fab2015   Mar2015   Apr2005
exp1  0.030083  0.065339  0.960494  0.510847
exp2  0.531885  0.706945  0.964943  0.085642
exp3  0.981325  0.868894  0.947871  0.387600
exp4  0.832527  0.357885  0.538138  0.357990
>>> frame.to_excel('data2.xlsx')
```

工作目录中会生成一个包含数据的新Excel文件，如图5-3所示。

	A	B	C	D	E
1		Jan2015	Fab2015	Mar2015	Apr2005
2	exp1	0,030083	0,065339	0,960494	0,510847
3	exp2	0,531885	0,706945	0,964943	0,085642
4	exp3	0,981325	0,868894	0,947871	0,3876
5	exp4	0,832527	0,357885	0,538138	0,35799
6					

图5-3 Excel文件中的DataFrame数据

5.7 JSON 数据

JSON（JavaScript Object Notation，JavaScript对象标记）已成为最常用的标准数据格式之一，特别是在Web数据的传输方面。因此，如果要使用Web数据，通常要处理这类数据格式。

这种格式很灵活，尽管数据结构跟你很熟悉的列表形式差别很大。

这一节，你将学习使用I/O API函数中的read_json()和to_json()函数。本节第二部分将给出一个例子，要求你处理JSON格式的结构化数据，这跟真实应用场景更为接近。

我觉得用于检测JSON格式是否正确的一个很好用的在线应用是JSONViewer，其网址是http://jsonviewer.stack.hu/。输入和复制JSON数据到这个Web应用中，就可以检测其格式是否合法。它还能以树状结构显示数据，方便你理解数据的结构（如图5-4所示）。

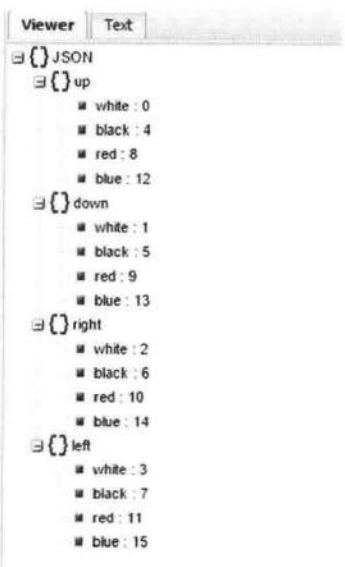


图5-4 JSONViewer

我们看看这个更为有用的使用场景吧。你有一个DataFrame，需要将其转换为JSON文件。因

此，定义一个DataFrame对象，然后调用它的to_json()函数，传入你要创建的文件名作为参数。

```
>>> frame = pd.DataFrame(np.arange(16).reshape(4,4),
...                      index=['white','black','red','blue'],
...                      columns=['up','down','right','left'])
>>> frame.to_json('frame.json')
```

工作目录中新增一个JSON文件（见代码清单5-12），它包含JSON格式的DataFrame数据。

代码清单5-12　frame.json

```
{"up":{"white":0,"black":4,"red":8,"blue":12},"down":{"white":1,"black":5,"red":9,"blue":13},
"right":{"white":2,"black":6,"red":10,"blue":14},"left":{"white":3,"black":7,"red":11,"blue":15}}
```

写入操作的逆操作——读取JSON文件也很简单，用read_json()函数，传入文件名作为参数即可。

```
>>> pd.read_json('frame.json')
       down  left  right  up
black     5     7      6   4
blue     13    15     14  12
red       9    11     10   8
white     1     3      2   0
```

上述例子相当简单，其中的JSON数据为列表形式（因为frame.json文件是由DataFrame对象转换而来的）。然而，JSON文件中的数据通常不是列表形式。因此，你需要将字典结构的文件转换为列表形式。这个过程称为规范化（normalization）。

pandas库的json_normalize()函数能够将字典或列表转换为表格。使用前，首先需要导入这个函数：

```
>>> from pandas.io.json import json_normalize
```

然后，用任意文本编辑器编写如代码清单5-13所示的JSON文件，将其保存到工作目录下，文件名为books.json。

代码清单5-13　books.json

```
[{"writer": "Mark Ross",
 "nationality": "USA",
 "books": [
        {"title": "XML Cookbook", "price": 23.56},
        {"title": "Python Fundamentals", "price": 50.70},
        {"title": "The NumPy library", "price": 12.30}
            ]
},

{"writer": "Barbara Bracket",
 "nationality": "UK",
 "books": [
        {"title": "Java Enterprise", "price": 28.60},
        {"title": "HTML5", "price": 31.35},
        {"title": "Python for Dummies", "price": 28.00}
```

```
            ]
}]
```

如上所见，文件结构不再是列表形式，而是一种更为复杂的形式。因此无法再使用read_json()函数来处理。正如你将从这个例子中学到的，我们仍然可以从这个数据结构中获取到列表形式的数据。首先，加载JSON文件的内容，并将其转换为一个字符串。

```
>>> file = open('books.json','r')
>>> text = file.read()
>>> text = json.loads(text)
```

然后就可以调用json_normalize()函数。快速浏览JSON文件中的数据后，举个例子，你可能想得到一个包含所有图书信息的表格，这种情况下只要把键books作为第二个参数即可。

```
>>> json_normalize(text,'books')
   price                title
0  23.56          XML Cookbook
1  50.70  Python Fundamentals
2  12.30     The NumPy library
3  28.60       Java Enterprise
4  31.35                HTML5
5  28.00   Python for Dummies
```

该函数会读取所有以books作为键的元素的值。元素中的所有属性将会转换为嵌套的列名称，而属性值将会转换为DataFrame的元素。该函数使用一串递增的数字作为索引。

然而，你得到的DataFrame对象只包含一部分内部信息。增加跟books位于同一级的其他键的值可能会有用处，把存储键名的列表作为第三个参数传入即可。

```
>>> json_normalize(text2,'books',['writer','nationality'])
   price                title nationality           writer
0  23.56          XML Cookbook         USA        Mark Ross
1  50.70  Python Fundamentals         USA        Mark Ross
2  12.30     The NumPy library         USA        Mark Ross
3  28.60       Java Enterprise          UK  Barbara Bracket
4  31.35                HTML5          UK  Barbara Bracket
5  28.00   Python for Dummies          UK  Barbara Bracket
```

我们用原来的树结构生成了一个DataFrame对象。

5.8 HDF5格式

至此，你已学习了文本格式的读写。若要分析大量数据，最好使用二进制格式。Python有多种二进制数据处理工具。HDF5库在这个方面取得了一定的成功。

HDF代表等级数据格式（hierarchical data format）。HDF5库关注的是HDF5文件的读写，这种文件的数据结构由节点组成，能够存储大量数据集。

该库全部用C语言开发，提供了Python、Matlab和Java语言接口。它的迅速扩展得益于开发人员的广泛使用，还得益于它的效率，尤其是使用这种格式存储大量数据，其效率很高。比起其他处理起二进制数据更为简单的格式，HDF5支持实时压缩，因而能够利用数据结构中的重复模式

压缩文件。

目前，Python提供两种操纵HDF5格式数据的方法：PyTables和h5py。这两种方法有几点不同，选用哪一种很大程度上取决于具体需求。

h5py为HDF5的高级API提供接口。PyTables封装了很多HDF5细节，提供更加灵活的数据容器、索引表、搜索功能和其他计算相关的介质。

pandas还有一个叫作HDFStore、类似于dict的类，它用PyTables存储pandas对象。使用HDF5格式之前，必须导入HDFStore类。

```
>>> from pandas.io.pytables import HDFStore
```

现在就可以把DataFrame中的数据存储到.h5文件中了。首先创建DataFrame对象。

```
>>> frame = pd.DataFrame(np.arange(16).reshape(4,4),
...                      index=['white','black','red','blue'],
...                      columns=['up','down','right','left'])
```

接着，创建一个叫mydata.h5的HDF5文件，把DataFrame中的数据存储到它里面。

```
>>> store = HDFStore('mydata.h5')
>>> store['obj1'] = frame
From here, you can guess how you can store multiple data structures within the same HDF5
file, specifying for each of them a label.
>>> frame2
       up  down  right  left
white   0   0.5      1   1.5
black   2   2.5      3   3.5
red     4   4.5      5   5.5
blue    6   6.5      7   7.5
>>> store['obj2'] = frame2
```

你可以把多种数据结构存储到一个HDF5文件中，比如store变量表示的这个文件。

```
>>> store
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
/obj1            frame        (shape->[4,4])
```

逆操作也很简单。我们来考虑一下包含多种数据结构的HDF5文件，可以像下面这样获取里面的对象。

```
>>> store['obj2']
       Up  down  right  left
white   0   0.5      1   1.5
black   2   2.5      3   3.5
red     4   4.5      5   5.5
blue    6   6.5      7   7.5
```

5.9　pickle——Python 对象序列化

pickle模块实现了一个强大的算法，能够对用Python实现的数据结构进行序列化（pickling）和反序列化操作。序列化是指把对象的层级结构转换为字节流的过程。

序列化便于对象的传输、存储和重建，仅用接收器就能重建对象，还能保留它的所有原始特征。

Python的序列化操作由pickle模块实现。写作本书时，有一个cPickle模块，它对pickle模块做了大量优化（用C语言实现）。在很多情况下，这个模块甚至要比pickle模块快1000倍。然而，用哪个模块都可以，这两个模块的接口几乎一致。

详细讲解pandas操作这类格式的I/O函数之前，我们先来更为详细地讲解cPickle模块以及它的用法。

5.9.1 用 cPickle 实现 Python 对象序列化

pickle模块（或cPickle）使用的数据格式是Python独有的，默认使用ASCII表达式，以增强可读性。用文本编辑器打开pickle文件，你就会发现它里面的内容是能够读懂的。使用模块前，需要先导入它

```
>>> import cPickle as pickle
```

然后创建具有内部结构的足够复杂的对象，例如字典对象。

```
>>> data = { 'color': ['white','red'], 'value': [5, 7]}
```

接着，用cPickle模块的dumps()函数对data对象执行序列化操作。

```
>>> pickled_data = pickle.dumps(data)
```

输出pickled_data变量的值，查看dict对象序列化的结果。

```
>>> print pickled_data
(dp1
S'color'
p2
(lp3
S'white'
p4
aS'red'
p5
asS'value'
p6
(lp7
I5
aI7
as.
```

数据序列化后，再写入文件或用套接字、管道等发送都很简单。

传输结束后，用cPickle模块的loads()函数能够重建被序列化的对象（反序列化）。

```
>>> nframe = pickle.loads(pickled_data)
>>> nframe
{'color': ['white', 'red'], 'value': [5, 7]}
```

5.9.2 用 pandas 实现对象序列化

用pandas库实现对象序列化（反序列化）很方便，所有工具都是现成的，无需在Python会话

中导入cPickle模块，所有的操作都是隐式进行的。

pandas的序列化格式并不是完全使用ASCII编码。

```
>>> frame = pd.DataFrame(np.arange(16).reshape(4,4), index = ['up','down','left','right'])
>>> frame.to_pickle('frame.pkl')
```

工作目录中将生成新文件frame.pkl，其包含frame中的所有信息。

使用以下命令，就能打开PKL文件，读取里面的内容。

```
>>> pd.read_pickle('frame.pkl')
       0   1   2   3
up     0   1   2   3
down   4   5   6   7
left   8   9  10  11
right 12  13  14  15
```

如上所见，pandas的所有序列化和反序列化操作都在后台运行，用户根本看不到。这使得这两项操作对数据分析人员而言尽可能简单和易于理解。

注意　使用这种格式时，要确保打开的文件的安全性。pickle格式无法规避错误和恶意数据。

5.10　对接数据库

在很多应用中，所使用的数据来自于文本文件的很少，因为文本文件不是存储数据最有效的方式。

数据往往存储于SQL类关系型数据库，作为补充，NoSQL数据库近来也已流行开来。

从SQL数据库加载数据，将其转换为DataFrame对象很简单。pandas提供的几个函数简化了该过程。

pandas.io.sql模块提供独立于数据库、叫作sqlalchemy的统一接口。该接口简化了连接模式，不管对于什么类型的数据库，操作命令都只有一套。连接数据库使用create_engine()函数，你可以用它配置驱动器所需的用户名、密码、端口和数据库实例等所有属性。

下面是各种数据库的连接方法。

```
>>> from sqlalchemy import create_engine
```

PostgreSQL：

```
>>> engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')
```

MySQL：

```
>>> engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')
```

Oracle：

```
>>> engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')
```

MSSQL：

```
>>> engine = create_engine('mssql+pyodbc://mydsn')
```

SQLite：

```
>>> engine = create_engine('sqlite:///foo.db')
```

5.10.1 SQLite3 数据读写

作为第一个例子，你将学习使用Python内置的SQLite数据库sqlite3。SQLite3工具实现了简单、轻量级的DBMS SQL，因此可以内置于用Python语言实现的任何应用。它很实用，你可以在单个文件中创建一个嵌入式数据库。

若想使用数据库的所有功能而又不想安装真正的数据库，这个工具就是最佳选择。若想在使用真正的数据库之前练习数据库操作，或在单一程序中使用数据库存储数据而无需考虑接口，SQLite3都是不错的选择。

创建一个DataFrame对象，我们将用它在SQLite3数据库新建一张表。

```
>>> frame = pd.DataFrame( np.arange(20).reshape(4,5),
...                       columns=['white','red','blue','black','green'])
>>> frame
   white  red  blue  black  green
0      0    1     2      3      4
1      5    6     7      8      9
2     10   11    12     13     14
3     15   16    17     18     19
```

接着，连接SQLite3数据库。

```
>>> engine = create_engine('sqlite:///foo.db')
```

把DataFrame转换为数据库表。

```
>>> frame.to_sql('colors',engine)
```

反之，读取数据库，则需要使用read_sql()函数，参数为表名和engine实例。

```
>>> pd.read_sql('colors',engine)
   Index  white  red  blue  black  green
0      0      0    1     2      3      4
1      1      5    6     7      8      9
2      2     10   11    12     13     14
3      3     15   16    17     18     19
```

如上所见，由于pandas库提供了I/O API，数据库写操作也变得非常简单。

接下来，我们来看一下在不使用I/O API的情况下，应该怎样实现相同的操作。这个例子有助于理解为什么说pandas是读写数据库的好工具。

首先，连接数据库，创建数据表，正确定义数据类型，数据类型应与要加载的数据对得上。

```
>>> import sqlite3
>>> query = """
... CREATE TABLE test
... (a VARCHAR(20), b VARCHAR(20),
```

```
... c REAL,         d INTEGER
... );"""
>>> con = sqlite3.connect(':memory:')
>>> con.execute(query)
<sqlite3.Cursor object at 0x0000000009E7D730>
>>> con.commit()
```

接着，使用SQL INSERT语句插入数据。

```
>>> data = [('white','up',1,3),
...         ('black','down',2,8),
...         ('green','up',4,4),
...         ('red','down',5,5)]
>>> stmt = "INSERT INTO test VALUES(?,?,?,?)"
>>> con.executemany(stmt, data)
<sqlite3.Cursor object at 0x0000000009E7D8F0>
>>> con.commit()
```

前面你已学过如何加载数据到数据表，现在我们学习如何从数据库查找刚插入的数据。我们可以用SQL SELECT语句。

```
>>> cursor = con.execute('select * from test')
>>> cursor
<sqlite3.Cursor object at 0x0000000009E7D730>
>>> rows = cursor.fetchall()
>>> rows
[(u'white', u'up', 1.0, 3), (u'black', u'down', 2.0, 8), (u'green', u'up', 4.0, 4),
(u'red', 5.0, 5)]
```

你可以把元组列表传给DataFrame的构造函数，如需要列名称，可以用游标的description属性来获取。

```
>>> cursor.description
(('a', None, None, None, None, None, None), ('b', None, None, None, None, None, None),
('c', None, None, None, None, None), ('d', None, None, None, None, None, None))
>>> pd.DataFrame(rows, columns=zip(*cursor.description)[0])
       A     b  c  d
0  white    up  1  3
1  black  down  2  8
2  green    up  4  4
3    red  down  5  5
```

你可能已经发现这种方法很费劲。

5.10.2　PostgreSQL 数据读写

从0.14版本起，pandas开始支持postgresql数据库。请再次确认自己的个人计算机上安装的版本是是这个版本还是版本号比它还要大。

```
>>> pd.__version__
>>> '0.15.2'
```

下面这个例子，要求系统中安装PostgreSQL数据库。在例子中，我创建了一个叫作postgres的数据库，用户名为postgres，密码为password。请按照自己的设置，对下面的代码做相应修改。

连接数据库。

```
>>> engine = create_engine('postgresql://postgres:password@localhost:5432/postgres')
```

注意 在这个例子中，你很可能会遇到以下错误信息，这与在Windows系统中安装包的方式有关：

```
from psycopg2._psycopg import BINARY, NUMBER, STRING, DATETIME, ROWID
ImportError: DLL load failed: The specified module could not be found.
```

错误信息可能指在PATH中没有找到PostgreSQL DLL（特指libpq.dll）。把postgres\x.x\bin目录添加到PATH后，就应该可以用Python连接到PostgreSQL数据库了。

创建DataFrame对象：

```
>>> frame = pd.DataFrame(np.random.random((4,4)),
              index=['exp1','exp2','exp3','exp4'],
              columns=['feb','mar','apr','may']);
```

我们看一下把这些数据转换为数据表是多么简单。使用to_sql()函数就能把数据写入到数据表dataframe中。

```
>>> frame.to_sql('dataframe',engine)
```

pgAdmin III是管理PostgreSQL数据库的图形化应用，它的用处很大，支持Linux和Windows系统。使用该应用，就可以轻松查看刚创建的数据框数据表（见图5-5）。

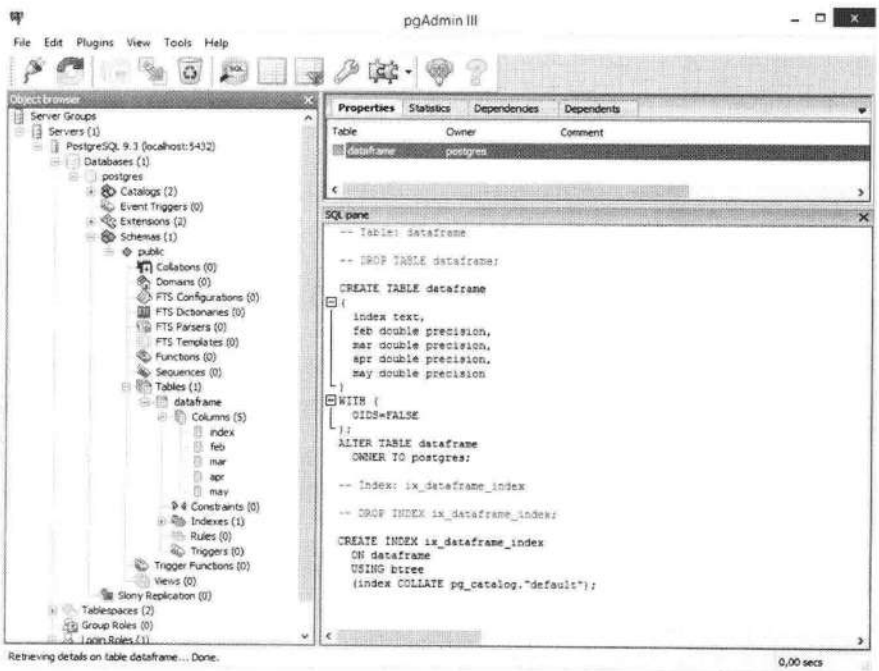


图5-5 PostgreSQL数据库的图形化管理工具pdAdminIII

你要是熟悉SQL语言，则要想查看新建的数据表及其内容，更经典的方法是借助psql会话。

```
>>> psql -U postgres
```

我用postgres用户连接数据库，你可能使用其他用户名。连接成功后，即可对新建的数据表执行SQL查询操作。

```
postgres=# SELECT * FROM DATAFRAME;
 index |        feb         |        mar         |        apr         |        may
-------+--------------------+--------------------+--------------------+--------------------
  exp1 | 0.757871296789076 | 0.422582915331819 | 0.979085739226726 |  0.332288515791064
  exp2 | 0.124353978978927 | 0.273461421503087 | 0.049433776453223 | 0.0271413946693556
  exp3 | 0.538089036334938 | 0.097041417119426 | 0.905979807772598 |  0.123448718583967
  exp4 | 0.736585422687497 | 0.982331931474687 | 0.958014824504186 |  0.448063967996436
(4 righe)
```

在pandas中，把数据表转换为DataFrame对象也是小菜一碟。pandas甚至提供了直接读取数据表返回DataFrame对象的read_sql_table()函数。

```
>>> pd.read_sql_table('dataframe',engine)
  Index       feb       mar       apr       may
0  exp1  0.757871  0.422583 0.979086  0.332289
1  exp2  0.124354  0.273461 0.049434  0.027141
2  exp3  0.538089  0.097041 0.905980  0.123449
3  exp4  0.736585  0.982332 0.958015  0.448064
```

读取数据库数据时，把整张表都转换为DataFrame并不是最常用的操作。使用关系型数据库的开发人员，更喜欢用SQL语言的SQL查询选择数据，指定导出形式。

可以将SQL查询语句整合到read_sql_table()函数中去。

```
>>> pd.read_sql_query('SELECT index,apr,may FROM DATAFRAME WHERE apr > 0.5',engine)
  index       apr       may
0  exp1  0.979086  0.332289
1  exp3  0.905980  0.123449
2  exp4  0.958015  0.448064
```

5.11　NoSQL 数据库 MongoDB 数据读写

在所有NoSQL数据库（BerkeleyDB、Tokyo Cabinet、MongoDB）中，MongoDB最为流行。由于它支持多种系统，我们有必要来了解一下如何把用pandas库分析数据所产生的结果写入MongoDB，以及如何从MongoDB读取数据到pandas。

首先，如已安装MongoDB，指定数据库目录，启动服务。

```
mongod --dbpath C:\MongoDB_data
```

服务正在监听27017端口，你可以用MongoDB官方提供的驱动器pymongo连接数据库。

```
>>> import pymongo
>>> client = MongoClient('localhost',27017)
```

一个MongoDB实例就能同时支持多个数据库。因此，你需要指定一个数据库。

```
>>> db = client.mydatabase
>>> db
Database(MongoClient('localhost', 27017), u'mycollection')
In order to refer to this object, you can also use
>>> client['mydatabase']
Database(MongoClient('localhost', 27017), u'mydatabase')
```

定义数据库后，你还需要定义集合（collection）。集合是指存储在MongoDB中的一组文档，可以将它理解为SQL数据库中的表。

```
>>> collection = db.mycollection
>>> db['mycollection']
Collection(Database(MongoClient('localhost', 27017), u'mydatabase'), u'mycollection')
>>> collection
Collection(Database(MongoClient('localhost', 27017), u'mydatabase'), u'mycollection')
Now it is the time to load the data in the collection. Create a DataFrame.
>>> frame = pd.DataFrame( np.arange(20).reshape(4,5),
...                       columns=['white','red','blue','black','green'])
>>> frame
   white  red  blue  black  green
0      0    1     2      3      4
1      5    6     7      8      9
2     10   11    12     13     14
3     15   16    17     18     19
```

添加到集合之前，DataFrame对象必须转换为JSON格式。转换过程比你可能想到的要复杂。这是因为你需要指定把哪些数据写入数据库，这样做同时也是为了将来可以尽可能简单地从数据库抽取数据到DataFrame中。

```
>>> import json
>>> record = json.loads(frame.T.to_json()).values()
>>> record
[{u'blue': 7, u'green': 9, u'white': 5, u'black': 8, u'red': 6}, {u'blue': 2, u'green': 4,
u'white': 0, u'black': 3, u'red': 1}, {u'blue': 17, u'green': 19, u'white': 15,
u'black': 18, u'red': 16}, {u'blue': 12, u'green': 14, u'white': 10, u'black': 13,
u'red': 11}]
Now you are finally ready to insert a document in the collection, and you can do this with
the insert() function.
>>> collection.mydocument.insert(record)
[ObjectId('54fc3afb9bfbee47f4260357'), ObjectId('54fc3afb9bfbee47f4260358'),
ObjectId('54fc3afb9bfbee47f4260359'), ObjectId('54fc3afb9bfbee47f426035a')]
```

如上所见，DataFrame对象的每一行都被转换为MongDB中的一个对象。数据加载到MongoDB数据库的文档之后，我们可以再来执行逆操作，也就是读取文档中的数据，然后将其转换为DataFrame对象。

```
>>> cursor = collection['mydocument'].find()
>>> dataframe = (list(cursor))
>>> del dataframe['_id']
>>> dataframe
   black  blue  green  red white
0      8     7      9    6     5
1      3     2      4    1     0
```

```
2    18    17    19    16    15
3    13    12    14    11    10
```

我们删除了用作MongoDB内部索引的ID编号这一列。

5.12　小结

本章中，你学会了如何用pandas库的I/O API工具，在保留DataFrame结构的前提下读写文件或数据库。我们着重介绍了几种适用于不同数据存储格式的读写模式。

本章最后一部分介绍了常用数据库的连接方法，以及如何从数据库读取数据并将其转换为可以用pandas工具直接处理的DataFrame对象，或者如何把DataFrame对象写入数据库。

下一章中，你将会学到pandas库最为高级的几个功能，我们会详细讲解GroupBy和其他数据处理方法。

第6章 深入pandas：数据处理

上一章讲解了从数据库或文件等数据源获取数据的方法。将数据转换为DataFrame格式后，你就可以对其进行处理了。数据处理的目的是准备数据，便于分析。数据处理很大程度上取决于必须进行数据分析的人员的目的，数据处理可以使要寻找的信息以更加清晰的方式呈现出来。尤其是为了做好下个阶段所需的准备工作，必须把数据处理成易于可视化的形式；可视化方法留待下一章讲解。

这一章，我们深入讲解pandas库在数据处理阶段的功能。数据处理又可以细分为三个阶段，我们将通过例子详细讲解各个阶段都会涉及哪些操作，以及如何充分利用pandas库提供的函数完成这些操作。数据处理的三个阶段为：

- 数据准备
- 数据转换
- 数据聚合

6.1 数据准备

开始处理数据工作之前，需要先行准备好数据，把数据组装成便于用pandas库的各种工具处理的数据结构。数据准备阶段包括以下步骤：

- 加载
- 组装
  - 合并（merging）
  - 拼接（concatenation）
  - 组合（combine）
- 变形（轴向旋转）
- 删除

说到数据加载，前一章主要就是介绍这个内容。在加载阶段也有部分数据准备工作，以把很多不同格式的数据转换为DataFrame等结构。数据可能是来自不同的数据源，有着不同的格式，但是即使获取到数据，把它们归并为一个DataFrame后，你还需要做进一步处理，才能把数据准备好。在本章，尤其是这一节，你将会学到把数据转换为统一的数据结构所需的各种操作。

对于存储在pandas对象中的各种数据，其组装方法有以下几种。

- 合并——pandas.mege()函数根据一个或多个键连接多行。SQL语言掌握得好的话，会觉得这种模式很熟悉，因为它同样实现了几种不同的join操作。
- 拼接——pandas.concat()函数按照轴把多个对象拼接起来。
- 结合——pandas.DataFrame.combine_first()函数从另外一个数据结构获取数据，连接重合的数据，以填充缺失值。

此外，数据准备过程还可能会涉及变换行、列位置的变形操作。

合并

对于合并操作，熟悉SQL的读者可以将其理解为JOIN操作，它使用一个或多个键把多行数据结合在一起。

事实上，跟关系型数据库打交道的开发人员通常使用SQL的JOIN查询，用几个表共有的引用值（键）从不同的表获取数据。以这些键为基础，我们能够获取到列表形式的新数据，这些数据是对几个表中的数据进行组合得到的。pandas库中这类操作叫作合并，执行合并操作的函数为merge()。

首先，导入pandas库，定义两个DataFrame对象，以用于这一节的例子。

```
>>> import numpy as np
>>> import pandas as pd
>>> frame1 = pd.DataFrame( {'id':['ball','pencil','pen','mug','ashtray'],
...                         'price': [12.33,11.44,33.21,13.23,33.62]})
>>> frame1
       id  price
0    ball  12.33
1  pencil  11.44
2     pen  33.21
3     mug  13.23
4 ashtray  33.62
>>> frame2 = pd.DataFrame( {'id':['pencil','pencil','ball','pen'],
...                         'color': ['white','red','red','black']})
>>> frame2
   color      id
0  white  pencil
1    red  pencil
2    red    ball
3  black     pen
```

对两个DataFrame对象应用merge()函数，执行合并操作。

```
>>> pd.merge(frame1,frame2)
      id  price  color
0   ball  12.33    red
1 pencil  11.44  white
2 pencil  11.44    red
3    pen  33.21  black
```

由结果可见，返回的DataFrame对象由原来两个DataFrame对象中ID相同的行组成。除了ID这一列，新DataFrame对象还包括原来分属于两个DataFrame的其他列。

这个例子中，我们没有为merge()指定基于哪一列进行合并。实际应用中，绝大部分情况下需要指定基于哪一列进行合并。

具体做法是增加on选项，把列的名称作为用于合并的键赋给它。

```
>>> frame1 = pd.DataFrame( {'id':['ball','pencil','pen','mug','ashtray'],
...                         'color': ['white','red','red','black','green'],
...                         'brand': ['OMG','ABC','ABC','POD','POD']})
>>> frame1
  brand  color      id
0   OMG  white    ball
1   ABC    red  pencil
2   ABC    red     pen
3   POD  black     mug
4   POD  green ashtray
>>> frame2 = pd.DataFrame( {'id':['pencil','pencil','ball','pen'],
...                         'brand': ['OMG','POD','ABC','POD']})
>>> frame2
  brand      id
0   OMG  pencil
1   POD  pencil
2   ABC    ball
3   POD     pen
```

由于我们刚定义的两个DataFrame对象，一个对象的列名称在另一个对象中也存在，所以对它们执行合并操作将得到一个空DataFrame对象。

```
>>> pd.merge(frame1,frame2)
Empty DataFrame
Columns: [brand, color, id]
Index: []
```

因此，有必要明确定义pandas合并操作所遵循的标准。我们用on选项指定合并操作所依据的基准列。

```
>>> pd.merge(frame1,frame2,on='id')
  brand_x  color      id brand_y
0     OMG  white    ball     ABC
1     ABC    red  pencil     OMG
2     ABC    red  pencil     POD
3     ABC    red     pen     POD

>>> pd.merge(frame1,frame2,on='brand')
  brand  color    id_x     id_y
0   OMG  white    ball   pencil
1   ABC    red  pencil     ball
2   ABC    red     pen     ball
3   POD  black     mug   pencil
4   POD  black     mug      pen
5   POD  green ashtray   pencil
6   POD  green ashtray      pen
```

如你所料，合并标准不同，结果差异很大。

然而，问题随之就来了。假如两个DataFrame基准列的名称不一致，该怎样进行合并呢？为了解决这个问题，你可以用left_on和right_on选项指定第一个和第二个DataFrame的基准列。举个例子：

```
>>> frame2.columns = ['brand','sid']
>>> frame2
  brand     sid
0   OMG  pencil
1   POD  pencil
2   ABC    ball
3   POD     pen
>>> pd.merge(frame1, frame2, left_on='id', right_on='sid')
  brand_x  color      id brand_y     sid
0     OMG  white    ball     ABC    ball
1     ABC    red  pencil     OMG  pencil
2     ABC    red  pencil     POD  pencil
3     ABC    red     pen     POD     pen
```

merge()函数默认执行的是内连接操作；上述结果中的键是由交叉操作（intersection）得到的。

其他选项有左连接、右连接和外连接。外连接把所有的键整合到一起，其效果相当于左连接和右连接的效果之和。连接类型用how选项指定。

```
>>> frame2.columns['brand','id']
>>> pd.merge(frame1,frame2,on='id')
  brand_x  color      id brand_y
0     OMG  white    ball     ABC
1     ABC    red  pencil     OMG
2     ABC    red  pencil     POD
3     ABC    red     pen     POD
>>> pd.merge(frame1,frame2,on='id',how='outer')
  brand_x  color       id brand_y
0     OMG  white     ball     ABC
1     ABC    red   pencil     OMG
2     ABC    red   pencil     POD
3     ABC    red      pen     POD
4     POD  black      mug     NaN
5     POD  green  ashtray     NaN

>>> pd.merge(frame1,frame2,on='id',how='left')
  brand_x  color       id brand_y
0     OMG  white     ball     ABC
1     ABC    red   pencil     OMG
2     ABC    red   pencil     POD
3     ABC    red      pen     POD
4     POD  black      mug     NaN
5     POD  green ashtray     NaN
>>> pd.merge(frame1,frame2,on='id',how='right')
  brand_x  color      id brand_y
0     OMG  white    ball     ABC
```

```
1    ABC    red  pencil    OMG
2    ABC    red  pencil    POD
3    ABC    red     pen    POD
```

要合并多个键，则把多个键赋给on选项。

```
>>> pd.merge(frame1,frame2,on=['id','brand'],how='outer')
  brand  color       id
0   OMG  white     ball
1   ABC    red   pencil
2   ABC    red      pen
3   POD  black      mug
4   POD  green  ashtray
5   OMG    NaN   pencil
6   POD    NaN   pencil
7   ABC    NaN     ball
8   POD    NaN      pen
```

根据索引合并

有时，合并操作不是用DataFrame的列而是用索引作为键。把left_index和right_index选项的值置为True，将其激活，就可将其作为合并DataFrame的基准。

```
>>> pd.merge(frame1,frame2,right_index=True, left_index=True)
  brand_x  color    id_x brand_y    id_y
0     OMG  white    ball     OMG  pencil
1     ABC    red  pencil     POD  pencil
2     ABC    red     pen     ABC    ball
3     POD  black     mug     POD     pen
```

但是DataFrame对象的join()函数更适合于根据索引进行合并。我们还可以用它合并多个索引相同或索引相同但列却不一致的DataFrame对象。

输入以下代码：

```
>>> frame1.join(frame2)
```

pandas将会给出错误信息，因为frame1的列名称与frame2有重合。因此在使用join()函数之前，要重命名frame2的列。

```
>>> frame2.columns = ['brand2','id2']
>>> frame1.join(frame2)
  brand  color       id brand2     id2
0   OMG  white     ball    OMG  pencil
1   ABC    red   pencil    POD  pencil
2   ABC    red      pen    ABC    ball
3   POD  black      mug    POD     pen
4   POD  green  ashtray    NaN     NaN
```

上述合并操作是以索引而不是列为基准。合并后得到的DataFrame对象包含只存在于frame1中的索引4，但是整合自frame2、索引号为4的各元素均为NaN。

6.2 拼接

另外一种数据整合操作叫作*拼接*（concatenation）。NumPy的concatenate()函数就是用于数组的拼接操作。

```
>>> array1
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> array2 = np.arange(9).reshape((3,3))+6
>>> array2
array([[ 6, 7, 8],
       [ 9, 10, 11],
       [12, 13, 14]])
>>> np.concatenate([array1,array2],axis=1)
array([[ 0, 1, 2, 6, 7, 8],
       [ 3, 4, 5, 9, 10, 11],
       [ 6, 7, 8, 12, 13, 14]])
>>> np.concatenate([array1,array2],axis=0)
array([[ 0, 1, 2],
       [ 3, 4, 5],
       [ 6, 7, 8],
       [ 6, 7, 8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

pandas库以及它的Series和DataFrame等数据结构实现了带编号的轴，它可以进一步扩展数组拼接功能。pandas的concat()函数实现了按轴拼接的功能。

```
>>> ser1 = pd.Series(np.random.rand(4), index=[1,2,3,4])
>>> ser1
1    0.636584
2    0.345030
3    0.157537
4    0.070351
dtype: float64
>>> ser2 = pd.Series(np.random.rand(4), index=[5,6,7,8])
>>> ser2
5    0.411319
6    0.359946
7    0.987651
8    0.329173
dtype: float64
>>> pd.concat([ser1,ser2])
1    0.636584
2    0.345030
3    0.157537
4    0.070351
5    0.411319
6    0.359946
7    0.987651
8    0.329173
dtype: float64
```

concat()函数默认按照axis=0这条轴拼接数据，返回Series对象。如果指定axis=1，返回结果将是DataFrame对象。

```
>>> pd.concat([ser1,ser2],axis=1)
          0         1
1  0.636584       NaN
2  0.345030       NaN
3  0.157537       NaN
4  0.070351       NaN
5       NaN  0.411319
6       NaN  0.359946
7       NaN  0.987651
8       NaN  0.329173
```

结果中无重合数据，因而刚执行的其实是外连接操作。把join选项置为'inner'，可执行内连接操作。[①]

```
>>> pd.concat([ser1,ser3],axis=1,join='inner')
          0         0   1
1  0.636584  0.636584 NaN
2  0.345030  0.345030 NaN
3  0.157537  0.157537 NaN
4  0.070351  0.070351 NaN
```

这种操作的问题是，从结果中无法识别被拼接的部分。假如你想在用于拼接的轴上创建等级索引，就需要借助keys选项来完成。

```
>>> pd.concat([ser1,ser2], keys=[1,2])
1  1    0.636584
   2    0.345030
   3    0.157537
   4    0.070351
2  5    0.411319
   6    0.359946
   7    0.987651
   8    0.329173
dtype: float64
```

按照axis=1拼接Series对象，所指定的键变为拼接后得到的DataFrame对象各列的名称。

```
>>> pd.concat([ser1,ser2], axis=1, keys=[1,2])
          1         2
1  0.636584       NaN
2  0.345030       NaN
3  0.157537       NaN
4  0.070351       NaN
5       NaN  0.411319
6       NaN  0.359946
7       NaN  0.987651
8       NaN  0.329173
```

① 下面代码中的ser3没有定义。据推测，作者应该是把pd.concat([ser1,ser2],axis=1)赋给了ser3。此外，ser3实际上是一个DataFrame对象，而再用表示“series”的ser命名不准确。

到目前为止，我们拼接的是Series对象，而DataFrame对象的拼接方法与之相同。

```
>>> frame1 = pd.DataFrame(np.random.rand(9).reshape(3,3), index=[1,2,3],
columns=['A','B','C'])
>>> frame2 = pd.DataFrame(np.random.rand(9).reshape(3,3), index=[4,5,6],
columns=['A','B','C'])
>>> pd.concat([frame1, frame2])
          A         B         C
1  0.400663  0.937932  0.938035
2  0.202442  0.001500  0.231215
3  0.940898  0.045196  0.723390
4  0.568636  0.477043  0.913326
5  0.598378  0.315435  0.311443
6  0.619859  0.198060  0.647902

>>> pd.concat([frame1, frame2], axis=1)
          A         B         C         A         B         C
1  0.400663  0.937932  0.938035       NaN       NaN       NaN
2  0.202442  0.001500  0.231215       NaN       NaN       NaN
3  0.940898  0.045196  0.723390       NaN       NaN       NaN
4       NaN       NaN       NaN  0.568636  0.477043  0.913326
5       NaN       NaN       NaN  0.598378  0.315435  0.311443
6       NaN       NaN       NaN  0.619859  0.198060  0.647902
```

6.2.1　组合

还有另外一种情况，我们无法通过合并或拼接方法组合数据。例如，两个数据集的索引完全或部分重合。

combine_first()函数可以用来组合Series对象，同时对齐数据。

```
>>> ser1 = pd.Series(np.random.rand(5),index=[1,2,3,4,5])
>>> ser1
1    0.942631
2    0.033523
3    0.886323
4    0.809757
5    0.800295
dtype: float64
>>> ser2 = pd.Series(np.random.rand(4),index=[2,4,5,6])
>>> ser2
2    0.739982
4    0.225647
5    0.709576
6    0.214882
dtype: float64
>>> ser1.combine_first(ser2)
1    0.942631
2    0.033523
3    0.886323
4    0.809757
5    0.800295
6    0.214882
```

```
dtype: float64
>>> ser2.combine_first(ser1)
1    0.942631
2    0.739982
3    0.886323
4    0.225647
5    0.709576
6    0.214882
dtype: float64
```

反之，如果你想进行部分合并，仅指定要合并的部分即可。

```
>>> ser1[:3].combine_first(ser2[:3])
1    0.942631
2    0.033523
3    0.886323
4    0.225647
5    0.709576
dtype: float64
```

6.2.2 轴向旋转

除了整合以统一来自不同数据源的数据，另外一种常用操作为*轴向旋转*（pivoting）。实际应用中，按行或列调整元素并不总能满足目标。有时，需要按照行重新调整列的元素或是按列调整行。

1. 按等级索引旋转

前面讲过，DataFrame对象支持等级索引。我们可以利用这一点，重新调整DataFrame对象中的数据。轴向旋转有两个基础操作。

- 入栈（stacking）：旋转数据结构，把列转换为行。
- 出栈（unstacking）：把行转换为列。

```
>>> frame1 = pd.DataFrame(np.arange(9).reshape(3,3),
...                       index=['white','black','red'],
...                       columns=['ball','pen','pencil'])
>>> frame1
       ball  pen  pencil
white     0    1       2
black     3    4       5
red       6    7       8
```

对DataFrame对象应用stack()函数，会把列转换为行，从而得到一个Series对象[①]：

```
>>> frame1.stack()
white ball      0
      pen       1
      pencil    2
black ball      3
      pen       4
```

① 这个Series对象就是下面提到的ser5。

```
      pencil     5
red   ball       6
      pen        7
      pencil     8
dtype: int32
```

在这个具有等级索引结构的Series对象上执行unstack()操作，可以重建之前的DataFrame对象，从而可以以数据透视表的形式来展示Series对象中的等级索引结构。

```
>>> ser5.unstack()
       ball  pen   pencil
white     0    1        2
black     3    4        5
red       6    7        8
```

出栈操作可以应用于不同的层级，为unstack()函数传入表示层级的编号或名称，即可对相应层级进行操作。

```
>>> ser5.unstack(0)
        white  black   red
ball        0      3     6
pen         1      4     7
pencil      2      5     8
```

2. 从“长”格式向“宽”格式旋转

数据集最通用的方式是，数据严格按照指定的字段进行记录，每一条数据作为一行写入CSV等文本文件或数据库表。如果数据来自仪器的读数，或是通过迭代计算得到的，或是由人工输入的一系列元素组成的，那么数据的格式很可能就是以上述方式存储的。例如，日志文件与这种文件类型具有相似的特点，它就是由一行行数据组成的。

该类数据集的特点是各列都有数据项，每一列后面的数据常常会跟前面的有所重复，并且这类数据常常为列表形式，你可以把它称作长格式或栈格式。

为了对这个概念有更清楚的认识，我们举个例子，请看下面这个DataFrame对象。

```
>>> longframe = pd.DataFrame({ 'color':['white','white','white',
...                                     'red','red','red',
...                                     'black','black','black'],
...                          'item':['ball','pen','mug',
...                                  'ball','pen','mug',
...                                  'ball','pen','mug'],
...                          'value': np.random.rand(9)})
>>> longframe
   color item     value
0  white ball  0.091438
1  white  pen  0.495049
2  white  mug  0.956225
3    red ball  0.394441
4    red  pen  0.501164
5    red  mug  0.561832
6  black ball  0.879022
7  black  pen  0.610975
8  black  mug  0.093324
```

然而，这种记录数据的模式有几个缺点。例如其中一个缺点是，因为一些字段具有多样性和重复性特点，所以选取列作为键时，这种格式的数据可读性较差，尤其是无法完全理解基准列和其他列之间的关系。

除了长格式，还有一种把数据调整为表格形式的宽格式。这种模式可读性强，也易于连接其他表，且占用空间较少。因此一般而言，用它存储数据效率更高，虽然它的可操作性差，这一点尤其体现在填充数据时。

如要选择一列或几列作为主键，所要遵循的规则是其中的元素必须是唯一的。

讲到格式转换，pandas提供了能够把长格式DataFrame转换为宽格式的pivot()函数，它以用作键的一列或多列作为参数。

接着上面的例子，选择color列作为主键，item列作为第二主键，而它们所对应的元素则作为DataFrame的新列。

```
>>> wideframe = longframe.pivot('color','item')
>>> wideframe
          value
item       ball       mug       pen
color
black  0.879022  0.093324  0.610975
red    0.394441  0.561832  0.501164
white  0.091438  0.956225  0.495049
```

如上所见，这种格式的DataFrame对象更加紧凑，它里面的数据可读性也更强。

6.2.3 删除

数据处理的最后一步是删除多余的列和行，第4章提到过这部分内容。然而，为了内容完整起见，这里再重新讲述一下。举例来说，我们还是先来定义一个DataFrame对象。

```
>>> frame1 = pd.DataFrame(np.arange(9).reshape(3,3),
...                       index=['white','black','red'],
...                       columns=['ball','pen','pencil'])
>>> frame1
       ball   pen   pencil
white     0     1        2
black     3     4        5
red       6     7        8
```

要删除一列，对DataFrame对象应用del命令，指定列名。

```
>>> del frame1['ball']
>>> frame1
       pen  pencil
white    1       2
black    4       5
red      7       8
```

要删除多余的行，使用drop()函数，将索引的名称作为参数。

```
>>> frame1.drop('white')
```

```
       pen  pencil
black    4       5
red      7       8
```

6.3 数据转换

至此，你已了解如何准备数据以便对其进行分析。这个过程实际体现在重组DataFrame中的数据上，我们可能要添加其他DataFrame或删除多余部分。

现在，该进行数据处理的第二步了：*数据转换*。调整过数据的形式和结构之后，接下来很重要的一步是对元素进行转换。从这一节，你将了解到数据转换的常见问题，以及用pandas函数解决这些问题需要哪些步骤。

在数据转换过程中，有些操作会涉及重复或无效元素，可能需要将其删除或替换为别的元素；而其他一些操作则跟删除索引相关；此外还有些步骤会涉及对数值或字符串类型的数据进行处理。

6.3.1 删除重复元素

出于多种原因，DataFrame对象可能包含重复的行。在大型DataFrame中，检测重复的行可能会遇到各种问题。pandas为此提供了多种工具，便于分析大型数据结构中的重复数据。

首先，创建一个包含重复行的简单DataFrame对象。

```
>>> dframe = pd.DataFrame({ 'color': ['white','white','red','red','white'],
...                         'value': [2,1,3,3,2]})
>>> dframe
   color value
0  white     2
1  white     1
2    red     3
3    red     3
4  white     2
```

DataFrame对象的duplicated()函数可用来检测重复的行，返回元素为布尔型的Series对象。每个元素对应一行，如果该行与其他行重复（也就是说该行不是第一次出现），则元素为True；如果跟前面不重复，则元素就为False。

```
>>> dframe.duplicated()
0    False
1    False
2    False
3     True
4     True
dtype: bool
```

返回元素为布尔值的Series对象用处很大，特别适用于过滤操作。如果要寻找重复的行，输入以下命令即可：

```
>>> dframe[dframe.duplicated()]
   color  value
3    red      3
4  white      2
```

通常，所有重复的行都需要从DataFrame对象中删除。pandas库的drop_duplicates()函数实现了删除功能，该函数返回的是删除重复行后的DataFrame对象。

```
>>> tokens = [s.strip() for s in text.split(',')]
>>> tokens
['16 Bolton Avenue', 'Boston']
```

6.3.2 映射

pandas提供了几个利用映射关系来实现某些操作的函数，这一节我们就来讲讲这几个函数。映射关系无非就是创建一个映射关系列表，把元素跟一个特定的标签或字符串绑定起来。

要定义映射关系，最好的对象莫过于dict。

```
map = {
   'label1' : 'value1,
   'label2' : 'value2,
   ...
}
```

这一节要讲的几个函数虽然执行的操作各不相同，但它们都以表示映射关系的dict对象作为参数。

- replace()：替换元素
- map()：新建一列
- rename()：替换索引

1. 用映射替换元素

组装完数据结构后，里面通常会有些元素不符合需求。例如，存在外语文本，一个元素是另一个元素的同义词，或者形状有出入。遇到这些情况，往往需要替换各种不同的元素。

举个例子，定义一个含有多种物体和颜色的DataFrame对象，其中有两种颜色不是用英语词汇来表示的。通常，组装数据时，有些元素的形式虽然不符合预期，但并没有对其进行处理。

```
>>> frame = pd.DataFrame({ 'item':['ball','mug','pen','pencil','ashtray'],
...                        'color':['white','rosso','verde','black','yellow'],
                           'price':[5.56,4.20,1.30,0.56,2.75]})
>>> frame
    color    item
0  white    ball
1    red     mug
2  green     pen
3  black  pencil
4 yellow ashtray
```

要用新元素替换不正确的元素，需要定义一组映射关系。在映射关系中，旧元素作为键，新

元素作为值。

```
>>> newcolors = {
...     'rosso': 'red',
...     'verde': 'green'
... }
```

接下来，调用replace()函数，传入表示映射关系的字典作为参数。

```
>>> frame.replace(newcolors)
    color     item price
0   white     ball  5.56
1     red      mug  4.20
2   green      pen  1.30
3   black   pencil  0.56
4  yellow  ashtray  2.75
```

由结果可知，DataFrame对象中两种旧颜色被替换为正确的元素。还有一种常见情况，是把NaN替换为其他值，比如0。这种情况下，仍然可以用replace()函数，它能优雅地完成该项操作。

```
>>> ser = pd.Series([1,3,np.nan,4,6,np.nan,3])
>>> ser
0    1
1    3
2  NaN
3    4
4    6
5  NaN
6    3
dtype: float64
>>> ser.replace(np.nan,0)
0    1
1    3
2    0
3    4
4    6
5    0
6    3
dtype: float64
```

2. 用映射添加元素

上例用映射关系替换元素。接下来这个例子将继续探索映射的用途。我们将利用映射关系从另外一个数据结构获取元素，将其添加到目标数据结构的列中。映射对象总是要单独定义的。

```
>>> frame = pd.DataFrame({ 'item':['ball','mug','pen','pencil','ashtray'],
...                        'color':['white','red','green','black','yellow']})
>>> frame
    color    item
0   white    ball
1     red     mug
2   green     pen
3   black  pencil
4  yellow ashtray
```

假如你想往DataFrame中添加一列商品价格信息。添加之前，假定你有一份价格清单，记录了每种商品的价格，只不过它在别处。再定义一个dict对象，它里面是一列商品及其价格信息。

```
>>> price = {
...     'ball' : 5.56,
...     'mug' : 4.20,
...     'bottle' : 1.30,
...     'scissors' : 3.41,
...     'pen' : 1.30,
...     'pencil' : 0.56,
...     'ashtray' : 2.75
... }
```

map()函数可应用于Series对象或DataFrame对象的一列，它接收一个函数或表示映射关系的字典对象作为参数。这里，在DataFrame的item这一列应用映射关系，用字典price作为参数，为DataFrame对象添加price列。

```
>>> frame['price'] = frame['item'].map(price)
>>> frame
    color     item   price
0   white     ball    5.56
1     red      mug    4.20
2   green      pen    1.30
3   black   pencil    0.56
4  yellow  ashtray    2.75
```

3. 重命名轴索引

我们可以采用跟操作Series和DataFrame对象的元素类似的方法，使用映射关系转换轴标签。pandas的rename()函数，以表示映射关系的字典对象作为参数，替换轴的索引标签。

```
>>> frame
    color    item    price
0   white    ball     5.56
1     red     mug     4.20
2   green     pen     1.30
3   black  pencil     0.56
4  yellow ashtray     2.75
>>> reindex = {
...   0: 'first',
...   1: 'second',
...   2: 'third',
...   3: 'fourth',
...   4: 'fifth'}
>>> frame.rename(reindex)
         color     item   price
first    white     ball    5.56
second     red      mug    4.20
third    green      pen    1.30
fourth   black   pencil    0.56
fifth   yellow  ashtray    2.75
```

如上所见，索引被重命名。若要重命名各列，必须使用columns选项。接下来我们把两个映

射对象分别赋给index和columns选项。

```
>>> recolumn = {
...     'item':'object',
...     'price': 'value'}
>>> frame.rename(index=reindex, columns=recolumn)
         color   object  value
first    white     ball   5.56
second     red      mug   4.20
third    green      pen   1.30
fourth   black   pencil  0.56
fifth   yellow  ashtray  2.75
```

对于只有单个元素要替换的最简单情况，可以对传入的参数做进一步限定，而无需把多个变量都写出来，也避免产生多次赋值操作。

```
>>> frame.rename(index={1:'first'}, columns={'item':'object'})
        color   object  price
0       white     ball   5.56
first     red      mug   4.20
2       green      pen   1.30
3       black   pencil   0.56
4     yellow  ashtray   2.75
```

前面这几个例子，rename()函数返回一个经过改动的新DataFrame对象，但原DataFrame对象仍保持不变。如果要改变调用函数的对象本身，可使用inplace选项，并将其值置为True。

```
>>> frame.rename(index={1:'first'}, columns={'item':'object'})
        color   object  price
0      white     ball   5.56
first    red      mug   4.20
2      green      pen   1.30
3      black   pencil   0.56
4     yellow  ashtray   2.75
```

6.4　离散化和面元划分

这一节，你将学到离散化这个更为复杂的数据转换过程。有时，尤其是在实验中，我们要处理的大量数据为连续型的。然而为了便于分析它们，我们需要把数据打散为几个类别，例如把（仪器）读数的取值范围划分为一个个小区间，统计每个区间的元素数量或其他统计量。另外一种情况是，对总体做出精确的测量，得到了大量个体。这种情况下，为了便于数据分析，也需要把元素分成几个类别，然后分别分析每个类别的个体数量及其他统计量。

举个例子，假如我们得到的实验读数介于0~100，且这些数据以列表形式存储。

```
>>> frame.rename(columns={'item':'object'}, inplace=True)
>>> frame
    color   object  price
0   white     ball   5.56
1     red      mug   4.20
2   green      pen   1.30
```

```
3   black   pencil   0.56
4  yellow  ashtray   2.75
```

已知实验数据的范围为0~100，因而我们可以把数据范围均分，比如分为四部分，也就是四个面元（bin）。第一个面元包含0~25的值，第二个为26~50，第三个为51~75，最后一个为76~100。

用pandas划分面元之前，首先要定义一个数组，存储用于面元划分的各数值。

```
>>> bins = [0,25,50,75,100]
```

然后，对results数组应用cut()函数，同时传入bins变量作为参数。

```
>>> cat = pd.cut(results, bins)
>>> cat
   (0, 25]
  (25, 50]
  (50, 75]
  (50, 75]
  (25, 50]
 (75, 100]
 (75, 100]
   (0, 25]
   (0, 25]
  (50, 75]
  (50, 75]
  (25, 50]
 (75, 100]
   (0, 25]
  (25, 50]
 (75, 100]
 (75, 100]
Levels (4): Index(['(0, 25]', '(25, 50]', '(50, 75]', '(75, 100]'], dtype=object)
```

cut()函数返回的对象为Categorical（类别型）类型，可以将其看作一个字符串数组，其元素为面元的名称。该对象内部的levels数组为不同内部类别的名称，labels数组的元素数量跟results数组（也就是说，划分成各面元的数组）相同，labels数组的各数字表示results元素所属的面元。

```
>>> cat.levels
Index([u'(0, 25]', u'(25, 50]', u'(50, 75]', u'(75, 100]'], dtype='object')
>>> cat.labels
array([0, 1, 2, 2, 1, 3, 3, 0, 0, 2, 2, 1, 3, 0, 1, 3, 3], dtype=int64)
```

如果你想知道每个面元的出现次数，即每个类别有多少个元素，可使用value_counts()函数。

```
>>> pd.value_counts(cat)
(75, 100]    5
(0, 25]      4
(25, 50]     4
(50, 75]     4
dtype: int64
```

如上所见，每个类别的下限用小括号表示，上限用方括号表示。这种标记方法与标识数字范

围的数学标记方法一致。如果小括号换为方括号，表示数字属于该范围（闭区间）；如果用小括号，表示数字不属于该范围（开区间）。

可以用字符串数组指定面元的名称，把它赋给cut()函数的labels选项，然后用该函数创建Categorical对象。

```
>>> bin_names = ['unlikely','less likely','likely','highly likely']
>>> pd.cut(results, bins, labels=bin_names)
      unlikely
   less likely
        likely
        likely
   less likely
 highly likely
 highly likely
      unlikely
      unlikely
        likely
        likely
   less likely
 highly likely
      unlikely
   less likely
 highly likely
 highly likely
Levels (4): Index(['unlikely', 'less likely', 'likely', 'highly likely'], dtype=object)
```

若不指定面元的各界限，而只传入一个整数作为参数，cut()函数就会按照指定的数字，把数组元素的取值范围划分为相应的几部分。

每个区间的上下限取决于样本数据（也就是要划分面元的数组）的最小值和最大值。

```
>>> pd.cut(results, 5)
 (2.904, 22.2]
  (22.2, 41.4]
  (60.6, 79.8]
  (41.4, 60.6]
  (22.2, 41.4]
    (79.8, 99]
    (79.8, 99]
 (2.904, 22.2]
 (2.904, 22.2]
  (41.4, 60.6]
  (60.6, 79.8]
  (41.4, 60.6]
    (79.8, 99]
  (22.2, 41.4]
  (41.4, 60.6]
    (79.8, 99]
    (79.8, 99]
Levels (5): Index(['(2.904, 22.2]', '(22.2, 41.4]', '(41.4, 60.6]',
                   '(60.6, 79.8]', '(79.8, 99]'], dtype=object)
```

除了cut()函数，pandas还有另外一个划分面元的函数：qcut()。这个函数直接把样本分成五

个面元。用cut()函数划分得到的面元，每个面元的个体数量不同，具体跟数据样例的分布相关。而qcut()函数能够保证每个面元的个体数相同，但每个面元的区间大小不等。

```
>>> quintiles = pd.qcut(results, 5)
>>> quintiles
    [3, 24]
   (24, 46]
 (62.6, 87]
 (46, 62.6]
   (24, 46]
   (87, 99]
   (87, 99]
    [3, 24]
    [3, 24]
 (46, 62.6]
 (62.6, 87]
   (24, 46]
 (62.6, 87]
    [3, 24]
 (46, 62.6]
   (87, 99]
 (62.6, 87]
Levels (5): Index(['[3, 24]', '(24, 46]', '(46, 62.6]', '(62.6, 87]',
                   '(87, 99]'], dtype=object)

>>> pd.value_counts(quintiles)
[3, 24]       4
(62.6, 87]    4
(87, 99]      3
(46, 62.6]    3
(24, 46]      3
dtype: int64
```

如上所见，qcut()函数和cut()函数所生成的区间具有不同的边界。更一进步来说，如果查看各面元所包含的个体数量，就会发现qcut()函数尝试为每个面元划分等量个体。但在我们这个例子中，前两个面元的个体数量比后面几个多，这是因为results个体数量无法被5整除。

异常值检测和过滤

在数据分析过程中，经常需要检测数据结构中的异常值。我们还是举个例子，先来创建一个包含三列的DataFrame对象，每一列都包含1000个随机数。

```
>>> randframe = pd.DataFrame(np.random.randn(1000,3))
```

可以用describe()函数查看每一列的描述性统计量。

```
>>> randframe.describe()
                0            1            2
count  1000.000000  1000.000000  1000.000000
mean      0.021609    -0.022926    -0.019577
std       1.045777     0.998493     1.056961
min      -2.981600    -2.828229    -3.735046
```

```
25%      -0.675005   -0.729834    -0.737677
50%       0.003857   -0.016940    -0.031886
75%       0.738968    0.619175     0.718702
max       3.104202    2.942778     3.458472
```

例如，你可能会将比标准差大3倍的元素视作异常值。用std()函数就可以求得DataFrame对象每一列的标准差。

```
>>> randframe.std()
0    1.045777
1    0.998493
2    1.056961
dtype: float64
```

接下来，根据每一列的标准差，对DataFrame对象的所有元素进行过滤。借助any()函数，就可以对每一列应用筛选条件。

```
>>> randframe[(np.abs(randframe) > (3*randframe.std())).any(1)]
            0         1         2
69  -0.442411 -1.099404  3.206832
576 -0.154413 -1.108671  3.458472
907  2.296649  1.129156 -3.735046
```

6.5　排序

用numpy.random.permutation()函数，调整Series对象或DataFrame对象各行的顺序（随机排序）很简单。

举个例子，创建一个元素为整数且按照升序排列的DataFrame对象。

```
>>> nframe = pd.DataFrame(np.arange(25).reshape(5,5))
>>> nframe
    0   1   2   3   4
0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
3  15  16  17  18  19
4  20  21  22  23  24
```

用permutation()函数创建一个包含0~4（顺序随机）这五个整数的数组。我们将按照这个数组元素的顺序为DataFrame对象的行排序。

```
>>> new_order = np.random.permutation(5)
>>> new_order
array([2, 3, 0, 1, 4])
```

对DataFrame对象的所有行应用take()函数，把新的次序传给它。

```
>>> nframe.take(new_order)
    0   1   2   3   4
2  10  11  12  13  14
3  15  16  17  18  19
0   0   1   2   3   4
```

```
1   5   6   7   8   9
4  20  21  22  23  24
```

如上所见，DataFrame对象各行的位置已发生改变。新索引的顺序跟new_order数组的元素顺序保持一致。

你甚至还可以只对DataFrame对象的一部分进行排序操作。它将生成一个数组，只包含特定索引范围的数据。例如，我们这里的2~4。

```
>>> new_order = [3,4,2]
>>> nframe.take(new_order)
    0   1   2   3   4
3  15  16  17  18  19
4  20  21  22  23  24
2  10  11  12  13  14
```

随机取样

上面刚讲了如何通过指定排列次序，从DataFrame对象中抽取一部分数据。若DataFrame规模很大，有时可能需要从中随机取样，最快的方法莫过于使用np.random.randint()函数。

```
>>> sample = np.random.randint(0, len(nframe), size=3)
>>> sample
array([1, 4, 4])
>>> nframe.take(sample)
    0   1   2   3   4
1   5   6   7   8   9
4  20  21  22  23  24
4  20  21  22  23  24
```

从随机取样这个例子可知，你可以多次获取相同的样本。

6.6 字符串处理

Python语言由于处理字符串和文本很方便，因而很受欢迎。大多数字符串操作用Python的内置函数就能轻松实现。字符串匹配及其他更为复杂的字符串处理，就有必要使用正则表达式了。

6.6.1 内置的字符串处理方法

你常常需要将复合字符串分成几部分，分别赋给不同的变量。split()函数以参考点为分隔符，比如逗号，将文本分为几部分。

```
>>> text = '16 Bolton Avenue , Boston'
>>> text.split(',')
['16 Bolton Avenue ', 'Boston']
```

如上所见，切分后得到的第一个元素以空白字符结尾。这个问题很常见。为了解决这个问题，使用split()函数切分后，还要再用strip()函数删除多余的空白字符（包括换行符）。

```
>>> tokens = [s.strip() for s in text.split(',')]
```

```
>>> tokens
['16 Bolton Avenue', 'Boston']
```

这样我们就得到了一个字符串数组。如果元素数量较少且固定不变，可使用下面这种非常有意思的赋值方法：

```
>>> address, city = [s.strip() for s in text.split(',')]
>>> address
'16 Bolton Avenue'
>>> city
'Boston'
```

上面讲的是文本的切分方法，但是我们通常还需要其逆操作，也就是把多个字符串拼接在一起形成一段长文本。

最直观和最简单的方法是用+运算符把几个文本片段拼接在一起。

```
>>> address + ',' + city
'16 Bolton Avenue, Boston'
```

若只有两三个字符串，这种拼接方法很好用。若要拼接很多字符串，更为实用的方法则是，在作为连接符的字符上调用join()函数。

```
>>> strings = ['A+','A','A-','B','BB','BBB','C+']
>>> ';'.join(strings)
'A+;A;A-;B;BB;BBB;C+'
```

另一类字符串操作是查找子串。Python的in关键字是检测子串的最好方法。

```
>>> 'Boston' in text
True
```

而这两个函数能够实现字符串查找：index()和find()。

```
>>> text.index('Boston')
19
>>> text.find('Boston')
19
```

这两个函数均返回子串在字符串中的索引。但是，如若没能找到子串，这两个函数的表现有所不同。

```
>>> text.index('New York')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> text.find('New York')
-1
```

若子串找不到，index()函数会报错，而find()函数会返回-1。再看看子串的其他操作，我们可以获知字符串或字符串组合在文本中的出现次数，用count()函数即可。

```
>>> text.count('e')
2
>>> text.count('Avenue')
1
```

针对字符串的另外一种操作是替换或删除字符串中的子串（或单个字符）。这两种操作都可以用replace()函数实现，如用空字符替换子串，效果等同于删除子串。

```
>>> text.replace('Avenue','Street')
'16 Bolton Street , Boston'
>>> text.replace('1','')
'16 Bolton Avenue, Boston'
```

6.6.2 正则表达式

用正则表达式在文本中查找和匹配字符串模式很灵活。单条正则表达式通常被称作regex，它是根据正则表达式语言编写的字符串。Python内置的re模块用于操作regex对象。

只有先导入re模块，才能使用正则表达式。

```
>>> import re
```

re模块所提供的函数可以分为以下几个类别：

- 模式匹配
- 替换
- 切分

我们先来看几个例子。例如，表示一个或多个空白字符的正则表达式为\s+。上一节中，我们用split()函数把文本切分为几部分。re模块中也有一个执行相同操作的split()函数，只不过它能够以正则表达式作为分隔符，使用起来更加灵活。

```
>>> text = "This is      an\t odd \n text!"
>>> re.split('\s+', text)
['This', 'is', 'an', 'odd', 'text!']
```

但若深入分析re模块的工作原理就能发现，调用re.split()函数时，首先编译正则表达式，然后在作为参数传入的文本上调用split()函数。你可以用re.compile()函数编译正则表达式，得到一个可以重用的正则表达式对象，从而节省CPU周期。

在字符串组合或数组中，迭代查找子串时，预先编译正则表达式，能显著提升效率。

```
>>> regex = re.compile('\s+')
```

用compile()函数创建regex对象后，可直接像下面这样调用它的split()方法。

```
>>> regex.split(text)
['This', 'is', 'an', 'odd', 'text!']
```

findall()函数可匹配文本中所有符合正则表达式的子串。该函数返回一个列表，元素为文本中所有符合正则表达式的子串。

例如，你想找出字符串中所有以大写字母A开头的单词，或者不区分大小写，需要使用以下代码。

```
>>> text = 'This is my address: 16 Bolton Avenue, Boston'
>>> re.findall('A\w+',text)
```

```
['Avenue']
>>> re.findall('[A,a]\w+',text)
['address', 'Avenue']
```

跟findall()函数相关的另外两个函数是：match()和search()。findall()函数返回一列所有符合模式的子串，而search()函数仅返回第一处符合模式的子串。此外，后者的返回结果为一个特殊类型的对象：

```
>>> re.search('[A,a]\w+',text)
<_sre.SRE_Match object at 0x0000000007D7ECC8>
```

该对象并不包含符合模式的子串，而是子串在字符串中的开始和结束位置。

```
>>> search = re.search('[A,a]\w+',text)
>>> search.start()
11
>>> search.end()
18
>>> text[search.start():search.end()]
'address'
```

match()函数从字符串开头开始匹配；如果第一个字符就不匹配，它不会再搜索字符串内部。如果没能找到任何匹配的子串，它不会返回任何对象。

```
>>> re.match('[A,a]\w+',text)
>>>
```

如果match()函数有返回内容，则它所返回的对象与search()函数返回的相同。

```
>>> re.match('T\w+',text)
<_sre.SRE_Match object at 0x0000000007D7ECC8>
>>> match = re.match('T\w+',text)
>>> text[match.start():match.end()]
'This'
```

6.7　数据聚合

数据处理的最后一步为数据聚合，通常指的是转换数据，使每一个数组生成一个单一的数值。我们已做过多种数据聚合操作，例如sum()、mean()和count()。这些函数均是操作一组数据，得到的结果只有一个数值。然而，对数据进行分类等聚合操作更为正式，对数据的控制力更强。

数据分类是为了把数据分成不同的组，通常是数据分析的关键步骤。之所以把它归到数据转换过程，是因为先把数据分成几组，再为不同组的数据应用不同的函数以转换数据。分组和应用函数这两个阶段经常用一步来完成。

对于数据分类，pandas提供了非常灵活和高效的GroupBy工具。

跟join操作类似，熟悉关系型数据库和SQL语言的读者将会发现，GroupBy和他们所使用的方法具有相似性。然而像SQL这类语言，它们的分组能力很有限。实际上，我们若使用Python这样非常灵活的编程语言，再加上pandas等库，可以实现很复杂的分组操作。

6.7.1 GroupBy

现在，我们来深入分析GroupBy过程及其工作原理。GroupBy通常指的是它的内部机制——SPLIT-APPLY-COMBINE（分组-用函数处理-合并结果）过程。它的操作模式由三个阶段组成，每个阶段可以用一种操作来准确地表示。

- 分组：将数据集分成多个组
- 用函数处理：用函数处理每一组
- 合并：把不同组得到的结果合并起来

接着详细分析上述三个阶段（图6-1）。在第一个阶段，也就是分组阶段，根据给定标准，把Series或DataFrame等数据结构中的数据分为几个不同的组，分组标准常与索引或某一列具体的元素相关。用SQL的行话来说，作为分组标准的这一列被称作键。进一步来说，如果你处理的是DataFrame等二维对象，分组标准可以既应用于行（`axis=0`），也应用于列（`axis=1`）。

第二个阶段也称作“用函数处理”，使用函数处理或者执行由函数定义的计算，为每组数据生成单一的值。

最后一步为合并，把来自每一组的结果汇集到一起，合并成一个新对象。

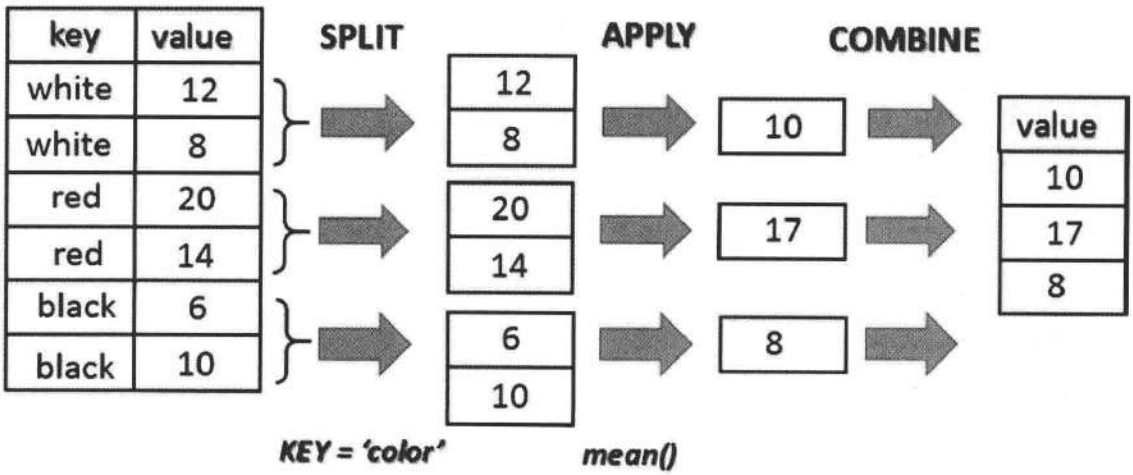


图6-1 SPLIT-APPLY-COMBINE原理

6.7.2 实例

刚讲过，pandas的数据聚合过程可分为SPLIT-APPLY-COMBINE三个阶段。pandas并没有使用你预想的三个函数来表示这个过程，而是只使用了`groupby()`函数，它生成的GroupBy对象是整个过程的核心。

为了更好地理解它的工作原理，必须来看一个实际的例子。首先，定义一个既包含数值又包含字符串的DataFrame对象。

```
>>> frame = pd.DataFrame({ 'color': ['white','red','green','red','green'],
...                        'object': ['pen','pencil','pencil','ashtray','pen'],
...                        'price1' : [5.56,4.20,1.30,0.56,2.75],
...                        'price2' : [4.75,4.12,1.60,0.75,3.15]})
>>> frame
   color  object  price1  price2
0  white     pen    5.56    4.75
1    red  pencil    4.20    4.12
2  green  pencil    1.30    1.60
3    red ashtray    0.56    0.75
4  green     pen    2.75    3.15
```

假如你想使用color列的组标签，计算price1列的均值，方法有多种。例如，你可以先获取到price1列，然后再调用groupby()函数，用参数指定color这一列。

```
>>> group = frame['price1'].groupby(frame['color'])
>>> group
<pandas.core.groupby.SeriesGroupBy object at 0x00000000098A2A20>
```

我们得到的对象为GroupBy对象。你刚执行的操作其实没有进行任何计算，它只是把计算需要的所有信息放到了一起。你刚进行的操作其实就是分组操作，把含有相同颜色的行分到同一个组里。

调用GroupBy对象的groups属性，我们来详细看一下DataFrame各行的分组情况。

```
>>> group.groups
{'white': [0L], 'green': [2L, 4L], 'red': [1L, 3L]}
```

如上所见，每个组都指定好了它所包含的行。现在，就可以对每组进行操作以获取结果了。

```
>>> group.mean()
color
green   2.025
red     2.380
white   5.560
Name: price1, dtype: float64

>>> group.sum()
color
green    4.05
red      4.76
white    5.56
Name: price1, dtype: float64
```

6.7.3　等级分组

前面讲过用一列元素作为键为数据分组。同理，还可以用多列，也就是使用多个键，按照等级关系分组。

```
>>> ggroup = frame['price1'].groupby([frame['color'],frame['object']])
>>> ggroup.groups
{('red', 'ashtray'): [3L], ('red', 'pencil'): [1L], ('green', 'pen'): [4L], ('green', '
], ('white', 'pen'): [0L]}
```

```
>>> ggroup.sum()
color  object
green  pen         2.75
       pencil      1.30
red    ashtray     0.56
       pencil      4.20
white  pen         5.56
Name:  price1, dtype: float64
```

到目前为止，我们按照一列数据对数据分类。事实上，我们可以按照多列数据或整个DataFrame把数据分为几组。如果你不想重复多次使用GroupBy对象，最方便的方法是一次就把所有的分组依据和计算方法都指定好，而无需定义任何中间变量。

```
>>> frame[['price1','price2']].groupby(frame['color']).mean()
       price1   price2
color
green   2.025    2.375
red     2.380    2.435
white   5.560    4.750
>>> frame.groupby(frame['color']).mean()
       price1 price2
color
green   2.025  2.375
red     2.380  2.435
white   5.560  4.750
```


6.8 组迭代

GroupBy对象还支持迭代操作，它可以生成一系列由各组名称及其数据部分组成的元组。

```
>>> for name, group in frame.groupby('color'):
...     print name
...     print group
...
green
   color object  price1  price2
2  green pencil    1.30    1.60
4  green    pen    2.75    3.15
red
  color  object  price1  price2
1   red  pencil    4.20    4.12
3   red ashtray    0.56    0.75
white
   color object price1 price2
0 white pen 5.56 4.75
```

上面这个例子中，我们只执行了输出变量的操作以了解详细信息。在实际应用中，可用具体的函数替换输出操作。

6.8.1 链式转换

从上述几个分组操作的例子中我们可以发现，用函数进行计算或执行其他操作时，不管各组是怎么得到的以及选取标准是什么，最终结果不是Series（如果只选择一列数据）就是DataFrame数据结构，它们保留了索引系统和列名称。

```
>>> result1 = frame['price1'].groupby(frame['color']).mean()
>>> type(result1)
<class 'pandas.core.series.Series'>
>>> result2 = frame.groupby(frame['color']).mean()
>>> type(result2)
<class 'pandas.core.frame.DataFrame'>
```

因此，在GroupBy过程的任何一个阶段都可以任意选择一列数据。下面分别在任一阶段选择一列数据。该例子展示了pandas库在分组操作上的巨大灵活性。

```
>>> frame['price1'].groupby(frame['color']).mean()
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
>>> frame.groupby(frame['color'])['price1'].mean()
color

green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
>>> (frame.groupby(frame['color']).mean())['price1']
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
```

此外，执行聚合操作后，某些列的名称可能存在表意不明确的现象。这时，在列名称前加上描述操作类型的前缀很有用。注意，添加前缀而不是完全替换名称，这样可以便于跟踪聚合数据的源数据。如果你采用的是链式转换过程（DataFrame之间存在生成关系），而又需要保留和源数据的对应关系，就可以使用这种方法。

```
>>> means = frame.groupby('color').mean().add_prefix('mean_')
>>> means
       mean_price1 mean_price2
color
green        2.025       2.375
red          2.380       2.435
white        5.560       4.750
```

6.8.2 分组函数

虽然很多函数不是专门为GroupBy对象实现的，它们却适用于Series数据结构。上一节中，我们讲过如何从GroupBy对象得到Series对象，即指定列名称，然后用函数执行计算就可以。例如，你可以用quantile()函数计算分位数。

```
>>> group = frame.groupby('color')
>>> group['price1'].quantile(0.6)
color
green    2.170
red      2.744
white    5.560
Name: price1, dtype: float64
```

你还可以自定义聚合函数。定义好函数后，将其作为参数传给agg()函数。例如，定义一个函数，计算每一组元素的取值范围。

```
>>> def range(series):
...       return series.max() - series.min()
...
>>> group['price1'].agg(range)
color
green    1.45
red      3.64
white    0.00
Name: price1, dtype: float64
```

可以对整个DataFrame对象应用agg()函数。

```
>>> group.agg(range)
       price1  price2
color
green    1.45    1.55
red      3.64    3.37
white    0.00    0.00
```

还可以同时使用多个聚合函数，把存放有表示聚合操作类型的数组传给agg()函数。这些操作将分别为DataFrame对象添加相应的新列。

```
>>> group['price1'].agg(['mean','std',range])
        mean       std  range
color
green  2.025  1.025305  1.45
red    2.380  2.573869  3.64
white  5.560 NaN 0.00
```

6.9 高级数据聚合

这一节介绍transform()和apply()函数，它们可以用来执行多种甚至是非常复杂的组操作。

假如我们想把下面的内容放到同一个DataFrame对象中：原DataFrame（含有数据的）和聚合

操作（比如求和）得到的计算结果。

```
>>> frame = pd.DataFrame({ 'color':['white','red','green','red','green'],
...                        'price1':[5.56,4.20,1.30,0.56,2.75],
...                        'price2':[4.75,4.12,1.60,0.75,3.15]})
>>> frame
   color  price1  price2
0  white    5.56    4.75
1    red    4.20    4.12
2  green    1.30    1.60
3    red    0.56    0.75
4  green    2.75    3.15
>>> sums = frame.groupby('color').sum().add_prefix('tot_')
>>> sums
       tot_price1 tot_price2
color
green        4.05       4.75
red          4.76       4.87
white        5.56       4.75
>>> merge(frame,sums,left_on='color',right_index=True)
  color price1 price2 tot_price1 tot_price2
0 white   5.56   4.75       5.56       4.75
1   red   4.20   4.12       4.76       4.87
3   red   0.56   0.75       4.76       4.87
2 green   1.30   1.60       4.05       4.75
4 green   2.75   3.15       4.05       4.75
```

可以用merge()函数把聚合操作的计算结果添加到DataFrame对象的每一行。实际上，你也可以用transform()方法实现这种操作。该函数执行聚合操作的方式跟前面讲过的相同，但它还可以根据DataFrame对象每一行的关键字显示聚合结果。

```
>>> frame.groupby('color').transform(np.sum).add_prefix('tot_')
   tot_price1 tot_price2
0        5.56       4.75
1        4.76       4.87
2        4.05       4.75
3        4.76       4.87
4        4.05       4.75
```

如上所见，transform()函数更适用于聚合操作，但是它对参数有特定要求：作为参数的函数必须生成一个标量（聚合），因为只有这样才能进行广播[①]。

apply()函数则适用于执行更为一般的GroupBy操作。这个方法完全实现了SPLIT-APPLY-COMBINE机制。它把对象分为几部分后，再用函数处理每一部分，各步骤之间用链式方法连接在一起。

```
>>> frame = DataFrame( { 'color':['white','black','white','white','black','black'],
...                      'status':['up','up','down','down','down','up'],
...                      'value1':[12.33,14.55,22.34,27.84,23.40,18.33],
```

① NumPy不同形状的数组做算术运算时的处理方法。短数组对长数组“广播”一遍，以保证它们同型。详见http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html。

```
...                     'value2':[11.23,31.80,29.99,31.18,18.25,22.44]})
>>> frame
   color status value1 value2
0  white     up  12.33  11.23
1  black     up  14.55  31.80
2  white   down  22.34  29.99
3  white   down  27.84  31.18
4  black   down  23.40  18.25

>>> frame.groupby(['color','status']).apply( lambda x: x.max())
              color status value1 value2
color status
black down    black down 23.40 18.25
      up      black   up 18.33 31.80
white down    white down 27.84 31.18
      up      white   up 12.33 11.23
5 black      up   18.33  22.44

>>> frame.rename(index=reindex, columns=recolumn)
         color   object value
first    white     ball 5.56
second     red      mug 4.20
third    green      pen 1.30
fourth   black   pencil 0.56
fifth   yellow  ashtray 2.75
>>> temp = date_range('1/1/2015', periods=10, freq= 'H')
>>> temp
<class 'pandas.tseries.index.DatetimeIndex'>
[2015-01-01 00:00:00, ..., 2015-01-01 09:00:00]
Length: 10, Freq: H, Timezone: None
>>> timeseries = Series(np.random.rand(10), index=temp)
>>> timeseries
2015-01-01 00:00:00    0.368960
2015-01-01 01:00:00    0.486875
2015-01-01 02:00:00    0.074269
2015-01-01 03:00:00    0.694613
2015-01-01 04:00:00    0.936190
2015-01-01 05:00:00    0.903345
2015-01-01 06:00:00    0.790933
2015-01-01 07:00:00    0.128697
2015-01-01 08:00:00    0.515943
2015-01-01 09:00:00    0.227647
Freq: H, dtype: float64

>>> timetable = DataFrame( {'date': temp, 'value1' : np.random.rand(10),
...                                       'value2' : np.random.rand(10)})
>>> timetable
                 date    value1   value2
0 2015-01-01 00:00:00  0.545737 0.772712
1 2015-01-01 01:00:00  0.236035 0.082847
2 2015-01-01 02:00:00  0.248293 0.938431
3 2015-01-01 03:00:00  0.888109 0.605302
4 2015-01-01 04:00:00  0.632222 0.080418
5 2015-01-01 05:00:00  0.249867 0.235366
```

```
6 2015-01-01 06:00:00  0.993940 0.125965
7 2015-01-01 07:00:00  0.154491 0.641867
8 2015-01-01 08:00:00  0.856238 0.521911
9 2015-01-01 09:00:00  0.307773 0.332822
```

接下来为上面的DataFrame对象再添加一列文本值，可以当作基准列使用。

```
>>> timetable['cat'] = ['up','down','left','left','up','up','down','right','right','up']
>>> timetable
                 Date   value1   value2  cat
0 2015-01-01 00:00:00 0.545737 0.772712    up
1 2015-01-01 01:00:00 0.236035 0.082847  down
2 2015-01-01 02:00:00 0.248293 0.938431  left
3 2015-01-01 03:00:00 0.888109 0.605302  left
4 2015-01-01 04:00:00 0.632222 0.080418    up
5 2015-01-01 05:00:00 0.249867 0.235366    up
6 2015-01-01 06:00:00 0.993940 0.125965  down
7 2015-01-01 07:00:00 0.154491 0.641867 right
8 2015-01-01 08:00:00 0.856238 0.521911 right
9 2015-01-01 09:00:00 0.307773 0.332822    up
```

上述DataFrame对象的基准列包含重复的键。

6.10　小结

本章介绍了数据处理的三个基本阶段：数据准备、数据转换和数据聚合。我们通过一系列的例子，讲解了实现这些操作的pandas函数。

学习了如何用这些函数处理简单的数据结构后，你就能熟悉它们的工作原理，理解如何将其用于更复杂的数据结构。

学完本章，你就掌握了为数据分析的下一阶段——数据可视化——准备数据集所需的全部工具。

下一章将介绍Python库matplotlib，它能将数据结构转换为各种类型的图表。

第7章 用matplotlib实现数据可视化

介绍完Python用于数据处理的几个库之后，该来了解一下实现数据可视化的库matplotlib了。

在数据分析工作中，人们往往对数据可视化这一步不够重视，但实际上它非常重要，因为错误或不充分的数据表示方法可能会毁掉原本很出色的数据分析工作。本章，我们将介绍matplotlib库各方面的知识，包括它的架构以及怎样充分发挥它的潜力。

7.1 matplotlib 库

matplotlib库是专门用于开发2D图表（包括3D图表）的，近年来被广泛应用于科技圈（http://matplolib.org）。

在促使它成为使用最多的数据图形化表示工具的众多优点中，以下几点最为突出：

- 使用起来极其简单
- 以渐进、交互式方式实现数据可视化
- 表达式和文本使用LaTeX排版
- 对图像元素控制力更强
- 可输出PNG、PDF、SVG和EPS等多种格式

matplotlib的设计初衷是在图形视图和句法形式方面尽可能重建跟Matlab类似的环境。这种做法已斩获成功，因为它能充分利用已有软件（Matlab）的设计经验。要知道Matlab已面市多年，现今广泛应用于科技圈。因此，不但matplotlib所依据的工作模式对业内专家来说再熟悉不过，而且其还充分利用了多年来总结得到的优化经验，提升其在使用方面的可推断性和简洁性。因此它非常适合第一次接触数据可视化的人员使用，尤其是那些没有任何Matlab或类似应用的使用经验的人。

除了简洁性和可推断性，matplotlib还继承了Matlab的交互性。也就是说，分析师可逐条输入命令，为数据生成渐趋完整的图形表示。这种模式很适合于用IPython QtConsole和IPython Notebook（参见第2章）等互动性更强的Python工具进行开发，这些工具所提供的数据分析环境堪与Mathematica、IDL和Matlab相媲美。

在开发matplotlib这个优美的库时，天才开发者使用和整合了优秀的技术和强大的工具，而这两者当前仍为科学圈所用。这不限于前面提过的Matlab的操作模式等，matplotlib还整合了LaTeX

用以表示科学表达式和符号的文本格式模型。LaTeX擅长展现科学表达式，所以它已成为任何要用到积分、求和及微分等公式的科学出版物或文档所不可或缺的排版工具。因而，为了提升图表的表现力，matplotlib整合了这个出色的工具。

此外，不容忽视的是，matplotlib不是一个单独的应用，而是编程语言Python的一个库。因此，它还充分利用了编程语言所能提供的潜力。matplotlib像是一个图形库，可通过编程来管理组成图表的图形元素，因此生成图形的全过程尽在其掌控之中。用编程方法生成图形，便于在多种环境下重新生成，尤其在改动或更新数据之后。

而且，由于matplotlib是一个Python库，所以用Python实现功能时，可充分利用所有Python开发人员都可以使用的其他各种库。事实上，虽然做数据分析时，matplotlib通常与NumPy和pandas等库配合使用，但其实其他很多库也都能无缝整合进来。

最后，用这个库编码实现的图形表示可以输出为最通用的图像格式（比如PNG和SVG），可以方便地用于其他应用、文档和网页等。

7.2　安装

matplotlib库的安装方法有多种。如果使用Anaconda或Enthought Canopy等发行版，则安装matplotlib非常简单。例如，若用conda包管理器安装，只需输入以下命令：

```
conda install matplotlib
```

如果要直接安装这个库，安装命令因操作系统而异。

Debian-Ubuntu Linux系统：

```
sudo apt-get install python-matplotlib
```

Fedora-Redhat Linux系统：

```
sudo yum install python-matplotlib
```

Windows或Mac OS X系统上，使用pip命令安装。

7.3　IPython 和 IPython QtConsole

为了熟悉Python世界所提供的所有工具，我尝试从命令行和QtConsole使用IPython，以充分利用IPython增强过的终端的交互能力以及QtConsole直接在控制台显示图像的长处。

运行下述命令，启动IPython：

```
ipython

Python 2.7.8 (default, Jul 2 2014, 15:12:11) [MSC v.1500 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 3.1.0 -- 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]:
```

然而，如果想在IPython QtConsole窗口以行内形式显示图像，请输入以下命令[①]：

```
ipython qtconsole --matplotlib inline
```

新的IPython会话窗口会立即打开，如图7-1所示。

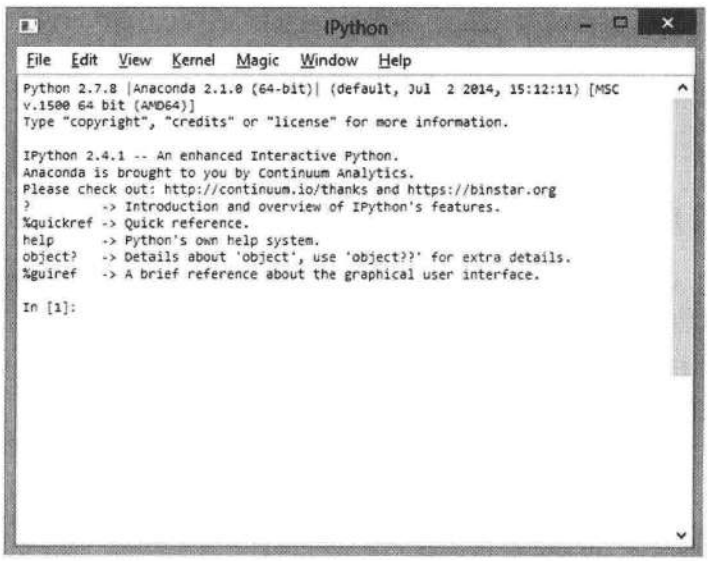


图7-1 IPython QtConsole

然而，你可以继续使用标准的Python会话。假如你不喜欢用IPython而想继续从终端使用Python，本章的所有例子依然有效。

7.4 matplotlib 架构

matplotlib的主要任务之一，就是提供一套表示和操作图形对象（主要对象）以及它的内部对象的函数和工具。然而，matplotlib不仅可以处理图形，还提供事件处理工具，具有为图形添加动画效果的能力。有了这些附加功能，matplotlib就能够生成以键盘按键或鼠标移动触发的事件的交互式图表。

从逻辑上来讲，matplotlib的整体架构由位于三个不同层级的三层组成（见图7-2）。各层之间单向通信，即每一层只能与它的下一层通信，而下层无法与上层通信。

matplotlib的架构分为以下三层：

- Scripting（脚本）层

① 在Anaconda终端运行该命令会提示无法识别命令中的“--matplotlib”标记。

- Artist（表现）层
- Backend（后端）层

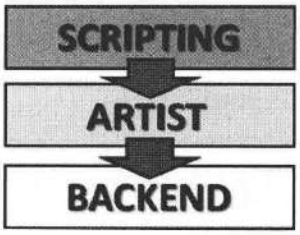


图7-2　matplotlib架构的三个层级

7.4.1　Backend 层

在上面matplotlib架构的图解中，最下面一层为Backend层。matplotlib API即位于该层，这些API是用来在底层实现图形元素的一个个类。

- FigureCanvas对象实现了绘图区域这一概念。
- Renderer对象在FigureCanvas上绘图。
- Event对象处理用户输入（键盘和鼠标事件）。

7.4.2　Artist 层

中间层为Artist层。图形中所有能看到的元素都属于Artist对象，即标题、轴标签、刻度等组成图形的所有元素都是Artist对象的实例。图形中每个元素的实例在层级结构中有着自己的位置（见图7-3）。

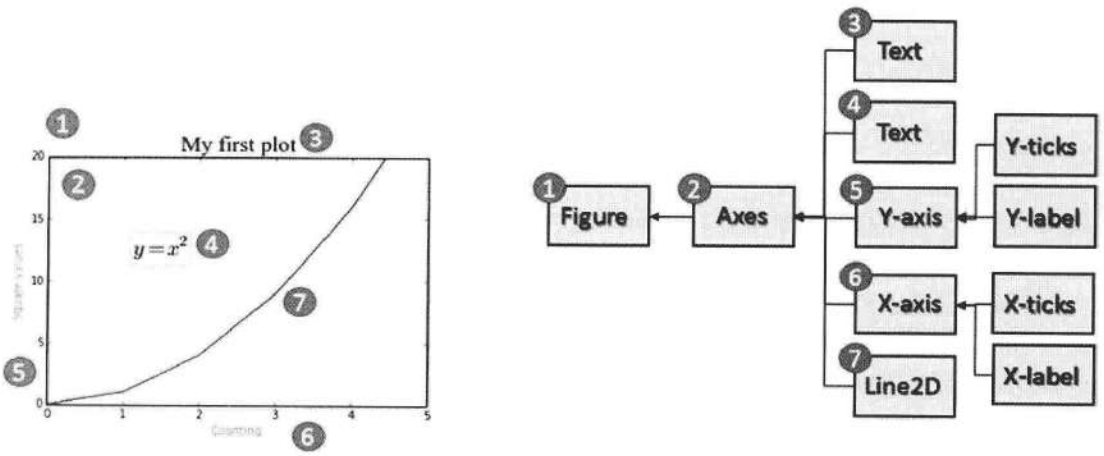


图7-3　图表的各元素分别对应一个Artist实例，所有实例形成层级结构

Artist类分为两种：原始（primitive）和复合（composite）。

绘制Line2D或矩形、圆形等几何图形，甚至文本等图形时，形成图形表示的基础元素由primitive artist单个对象组成。

由多个基础元素——primitive artist——组成的图表中的图像元素叫作composite artist，例如Axis（单条轴）、Ticks（刻度）、Axes（轴）和Figure（图形），请见图7-4。

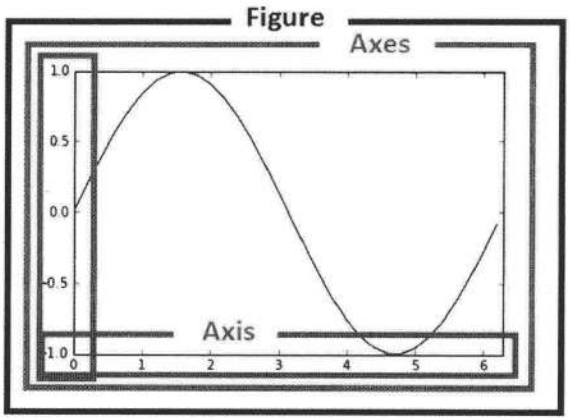


图7-4 Artist层的三个主要Artist对象

一般而言，在这个阶段（Artist），你通常需要处理Figure、Axes和Axis等位于高层级的对象。因此，透彻理解这些对象和它们在图形表示中所扮演的角色很重要。图7-4介绍了三个主要的Artist对象（composite artist），通常所有图形实现的Artist层都离不开它们。

Figure对象在Artist层的最上面，对应整个图形表示，通常可包含多条轴（Axes）。

Axes对象通常表示图形或图表是对什么内容进行作图的。每个Axes对象只属于一个Figure对象，由两个（三维就有三个）Artist Axis对象组成。标题、*x*标签和*y*标签等对象都属于Axes这个composite artist类型的对象。

Axis对象负责展示在Axes对象上面的数值，定义数值范围，管理刻度（轴上的标记）和刻度值标签（代表每个刻度大小的文本标签）。刻度的位置用Locator对象调整，刻度标签的格式用Formatter对象调整。

7.4.3 Scripting 层（pyplot）

Artist类和相关函数（matplotlib API）非常适合开发人员，尤其是Web应用服务器或GUI开发者使用。但是对于计算，尤其是数据分析和可视化，Scripting层最适合。该层包含pyplot接口。

7.4.4 pylab 和 pyplot

关于pylab和pyplot，人们做过不少讨论。这两个模块有哪些不同呢？pylab模块跟matplotlib

一起安装，而pyplot则是matplotlib的内部模块。两者的导入方法也有所不同，可选择其中一种进行导入。

```
from pylab import *
```

或

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

pylab在同一命名空间整合了pyplot和NumPy的功能，因此无需再单独导入NumPy。更进一步来说，导入pylab后，pyplot和NumPy的函数就可以直接调用，而不用再指定其所属模块（命名空间），从而使得matplotlib开发环境更像是Matlab。

```
plot(x,y)
array([1,2,3,4])
```

而不用指定模块名称：

```
plt.plot()
np.array([1,2,3,4]
```

pyplot模块提供操作matplotlib库的经典Python编程接口，有着自己的命名空间，需要单独导入NumPy库。书中选择使用pyplot模块，它是本章要讲解的主要内容，后续章节也将继续使用。事实上，大多数Python开发者认可并乐于使用这个模块。

7.5　pyplot

pyplot模块由一组命令式函数组成，因而matplotlib的使用方法跟Matlab极为相似，通过pyplot函数操作或改动Figure对象，例如创建Figure对象和绘图区域、表示一条线或为图形添加标签等。

pyplot还具有状态性特性，它能跟踪当前图形和绘图区域的状态。调用函数时，函数只对当前图形起作用。

7.5.1　生成一幅简单的交互式图表

为了熟悉matplotlib库，尤其是pyplot，我们来生成一幅简单的交互式图表。用matplotlib生成这个图表很简单，三行代码就能搞定。

首先要导入pyplot模块，并将其命名为plt。

```
In [1]: import matplotlib.pyplot as plt
```

Python通常不需要构造函数，因为一切都已经清楚地定义好了。导入这个模块时，plt对象及其图像处理功能已被实例化且可以使用。因此，把数据传给plot()函数，直接使用即可。

请把要绘制其图像的一列整数传给plot()函数。

```
In [2]: plt.plot([1,2,3,4])
Out[2]: [<matplotlib.lines.Line2D at 0xa3eb438>]
```

如上所见，生成了一个Line2D对象。该对象为一条直线，它表示图表中各数据点的线性延伸

趋势。

生成图表对象之后，只需要使用show()函数就能显示图表。

```
In [3]: plt.show()
```

结果如图7-5所示。它生成了一个绘图窗口，上面是工具栏，下面是绘制的图像，跟用Matlab作图效果相同。

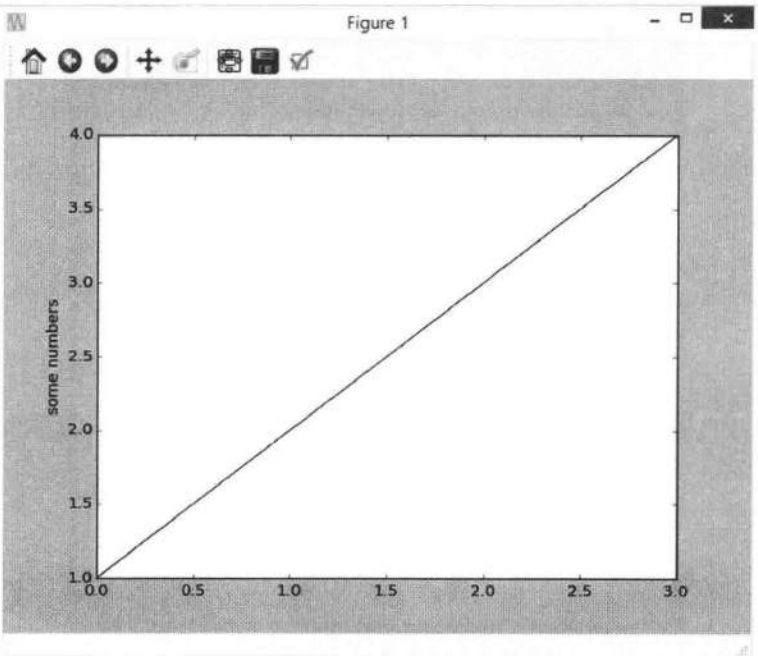


图7-5 绘图窗口

绘图窗口

绘图窗口顶部是一条工具栏，包含以下按钮。

- 重置为原始视图
- 去往前/后一个视图
- 用鼠标左键查看图形，用鼠标右键放大或缩小图形
- 框选设定视图放缩
- 设置子图
- 保存/导出图形
- 编辑曲线或轴的相关参数

如果你使用Python自带的shell，所要输入的代码与在IPython控制台输入的相同。

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1,2,3,4])
[<matplotlib.lines.Line2D object at 0x0000000007DABFD0>]
>>> plt.show()
```

如使用IPython QtConsole，你肯定已经发现调用plot()函数后，无需激活show()函数，图形就会在控制台显示（见图7-6）。

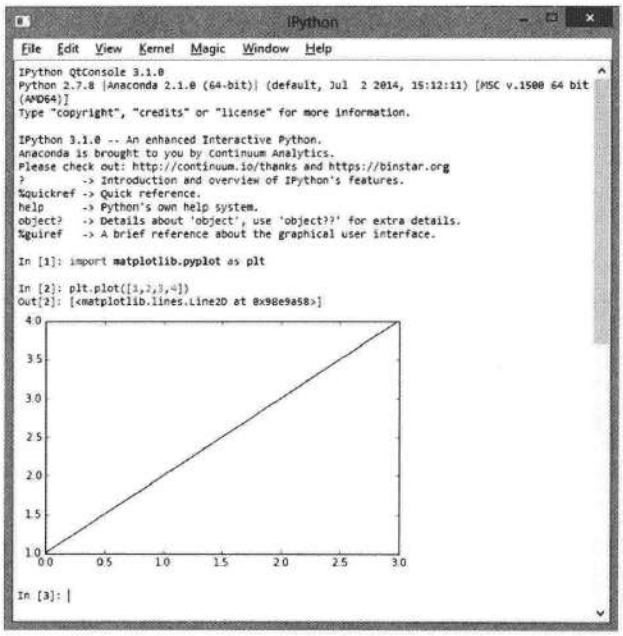


图7-6　QtConsole直接输出图表

如果只是将一个数字列表或数组传递给plt.plot()函数，matplotlib就会假定你所传入的是图表的y值，于是将其跟一个序列的x值对应起来，x的取值依次为0、1、2、3……。

通常，图形表示的是一对对的(x,y)，因此如果你想正确定义图表，必须定义两个数组，其中第一个数组为x轴的各个值，第二个数组为y轴的值。此外，plot()函数还可以接收第三个参数，它描述的是数据点在图表中的显示方式。

7.5.2　设置图形的属性

由图7-6可见，数据点用蓝线串在一起。如果你不指图表样式，matplotlib使用plt.plot()函数的默认设置绘制图像。

- 轴长与输入数据范围一致
- 无标题和轴标签

- 无图例
- 用蓝色线条连接各数据点

现在需要你修改图形，用红点来表示每一对(x, y)，生成一幅像模像样的图形（见图7-7）。

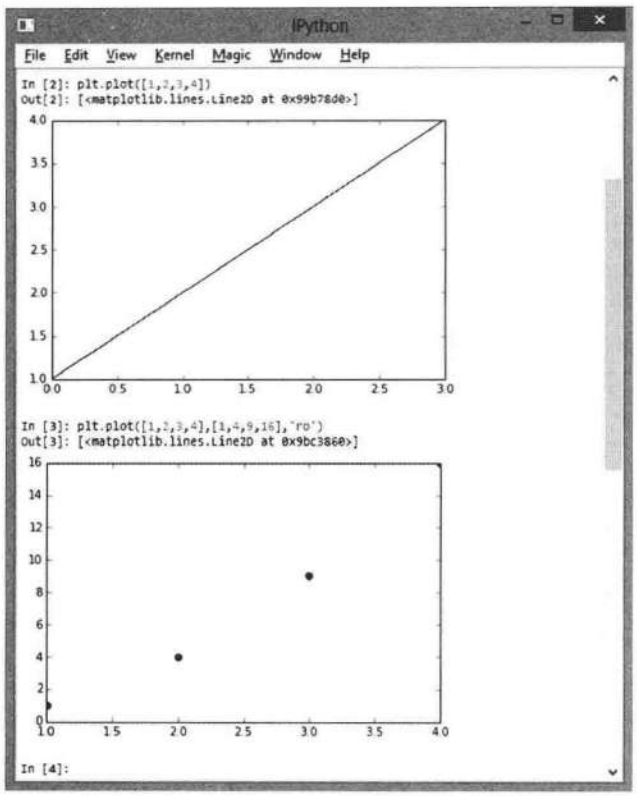


图7-7 图形中，数据点(x,y)用红点来表示

如果你正在使用IPython，请关闭图形窗口，再次回到处于活动状态的命令行界面，输入新命令。接着调用show()函数，观察图形发生了什么变化。

```
In [4]: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[4]: [<matplotlib.lines.Line2D at 0x93e6898>]

In [4]: plt.show()
```

然而，如果你使用的是IPython QtConsole，则不用执行上述操作，直接输入新命令（交互性）。

```
In [3]: plt.plot([1,2,3,4],[1,4,9,16],'ro')
```

注意 本书讲到这里，你已很清楚各种环境之间的不同点。为了避免带来不必要的混乱，从现在起，我将用IPython QtConsole作为开发环境。

你可以用列表[xmin, xmax, ymin, ymax]定义好x轴和y轴的取值范围，把该列表作为参数传给axis()函数。

注意　在IPython QtConsole中，生成一张图表，有时需要输入多行命令。为了避免每次按Enter键（换行）就会生成图表，从而丢失先前的设置，请按Ctrl+Enter键换行。最后要生成图表时，请按两次Enter键。

绘图时，有多个属性可以设置，例如，可以用title()函数增加标题。

```
In [4]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot')
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[4]: [<matplotlib.lines.Line2D at 0x97f1c18>]
```

由图7-8可见，新的设置增强了图形的可读性。数据集的两个端点在图形内，而不像之前那样在图形的边缘显示，图形的标题也在图形上方显示出来了。

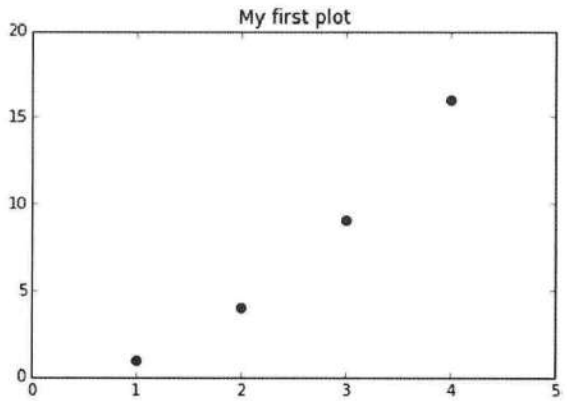


图7-8　设置属性后的图形

7.5.3　matplotlib 和 NumPy

尽管Matplot是一个图形库，它却以NumPy库作为基础。你已经见过如何传递列表作为参数，以表示数据点和设置轴的数值范围。这些列表在内部实际上被转换为NumPy数组。

因而，你可以直接把NumPy数组作为输入数据。数组经过pandas处理后，无需做进一步处理，可直接供matplotlib使用。

举个例子，我们来看一下如何在同一图形（见图7-9）中绘制三种不同的趋势图。这个例子中，我们将使用math模块的sin()函数，需先把它导入进来。我们使用NumPy库生成呈正弦趋势分布的数据点。用arange()函数生成x轴的一系列数据点t，而对于每个点所对应的y值，我们用map()函数，对x轴的一系列数据点t应用sin()函数（无需使用for循环）。

```
In [5]: import math
In [6]: import numpy as np
In [7]: t = np.arange(0,2.5,0.1)
   ...: y1 = map(math.sin,math.pi*t)
   ...: y2 = map(math.sin,math.pi*t+math.pi/2)
   ...: y3 = map(math.sin,math.pi*t-math.pi/2)
In [8]: plt.plot(t,y1,'b*',t,y2,'g^',t,y3,'ys')
Out[8]:
[<matplotlib.lines.Line2D at 0xcbd2e48>,
 <matplotlib.lines.Line2D at 0xcbe10b8>,
 <matplotlib.lines.Line2D at 0xcbe15c0>]
```

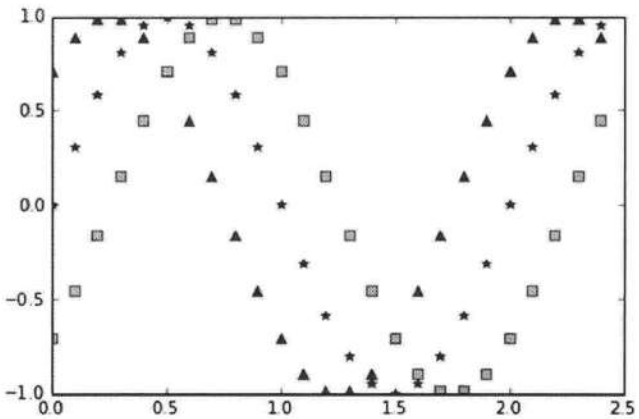


图7-9 用不同符号表示的正弦图像，各图像相位差π/4

注意 如果你不是在IPython QtConsole中以行内形式使用matplotlib，或者你只是在简单的Python会话中实现上述代码，记得在代码的最后添加plt.show()命令，以得到如图7-9所示的图表。

图7-9中用三种颜色和三种符号表示三种不同的趋势。这几条曲线上，函数图像的趋势很明显，因此使用符号可能不是最佳表示方法，用线条效果要更好（见图7-10）。除了用三种颜色区分三种趋势，我们还可以用由点和线（.和-）组成的不同线型。

```
In [9]: plt.plot(t,y1,'b--',t,y2,'g',t,y3,'r-.')
Out[9]:
[<matplotlib.lines.Line2D at 0xd1eb550>,
 <matplotlib.lines.Line2D at 0xd1eb780>,
 <matplotlib.lines.Line2D at 0xd1ebd68>]
```

注意 如果你不是在IPython QtConsole中以行内形式使用matplotlib，或者你在简单的Python会话中实现上述代码，记得在代码的最后添加plt.show()命令，以得到如图7-10所示的图表。

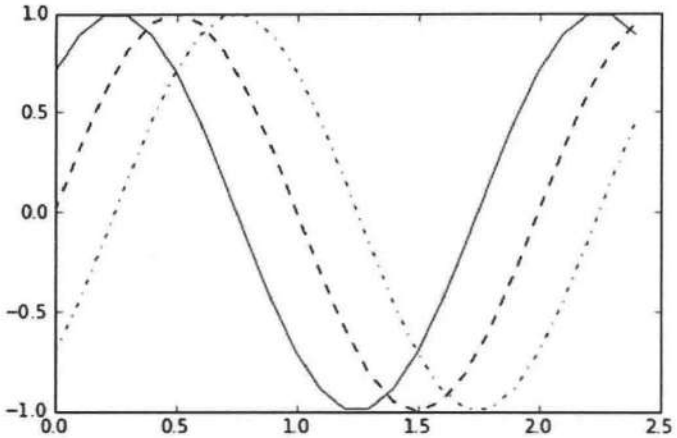


图7-10　用彩色线条表示的三种正弦趋势

7.6　使用 kwargs

组成图表的各个对象有很多用以描述它们特点的属性。这些属性均有默认值，但可以用关键字参数（keyword args，常称作kwargs）设置。

这些关键字作为参数传递给函数。在matplotlib库各个函数的参考文档中，每个函数的最后一个参数总是kwargs。例如，前几个例子一直在使用的plot()函数在文档中是这么定义的：

```
matplotlib.pyplot.plot(*args, **kwargs)
```

再举一个更加具体的例子，设置linewidth关键字参数，可以改变线条的粗细（见图7-11）。

```
In [10]: plt.plot([1,2,4,2,1,0,1,2,1,4],linewidth=2.0)
Out[10]: [<matplotlib.lines.Line2D at 0xc909da0>]
```

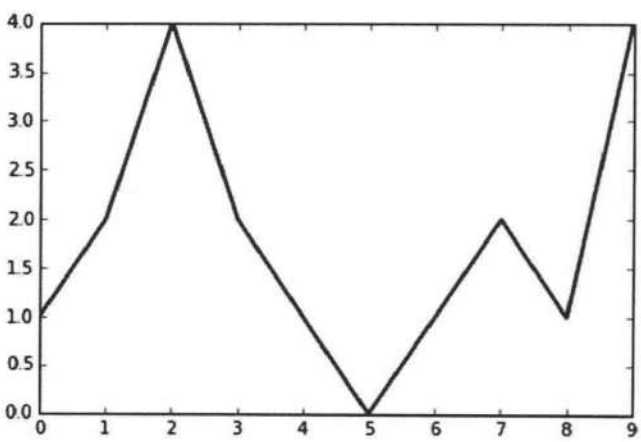


图7-11　线条的粗细可以直接在plot()函数中设置

处理多个 Figure 和 Axes 对象

至此，你见过的所有pyplot命令都是用来绘制单个图形的。其实，你还可以同时管理多个图形，而在每个图形中，又可以绘制几个不同的子图。

因此，使用pyplot时，必须时刻注意当前Figure对象和当前Axes对象的概念（也就是Figure对象中当前所显示的图形）。

我们来看一个在一幅图形中有两个子图的例子。subplot()函数不仅可以将图形分为不同的绘图区域，还能激活特定子图，以便用命令控制它。

subplot()函数用参数设置分区模式和当前子图。只有当前子图会受到命令的影响。subplot()函数的参数由三个整数组成：第一个数字决定图形沿垂直方向被分为几部分，第二个数字决定图形沿水平方向被分为几部分，第三个数字设定可以直接用命令控制的子图。

接着，我们来绘制两种正弦趋势图（正弦和余弦）。最佳方式是把画布分为上下两个向水平方向延伸的子图（见图7-12）。因此，作为参数传入的两个数字应分别为211和212。

```
In [11]: t = np.arange(0,5,0.1)
   ... : y1 = np.sin(2*np.pi*t)
   ... : y2 = np.sin(2*np.pi*t)
In [12]: plt.subplot(211)
    ...: plt.plot(t,y1,'b-.')
    ...: plt.subplot(212)
    ...: plt.plot(t,y2,'r--')
Out[12]: [<matplotlib.lines.Line2D at 0xd47f518>]
```

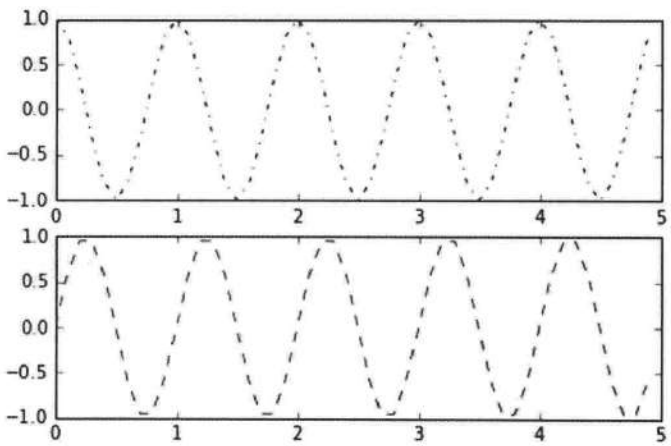


图7-12 分为上下两个子图的图形

我们还可以把图形分为左右两个子图。这时，subplot()函数的参数为121和122（见图7-13）。

```
In [ ]: t = np.arange(0.,1.,0.05)
   ...: y1 = np.sin(2*np.pi*t)
   ...: y2 = np.cos(2*np.pi*t)
In [ ]: plt.subplot(121)
   ...: plt.plot(t,y1,'b-.')
```

```
   ...: plt.subplot(122)
   ...: plt.plot(t,y2,'r--')
Out[94]: [<matplotlib.lines.Line2D at 0xed0c208>]
```

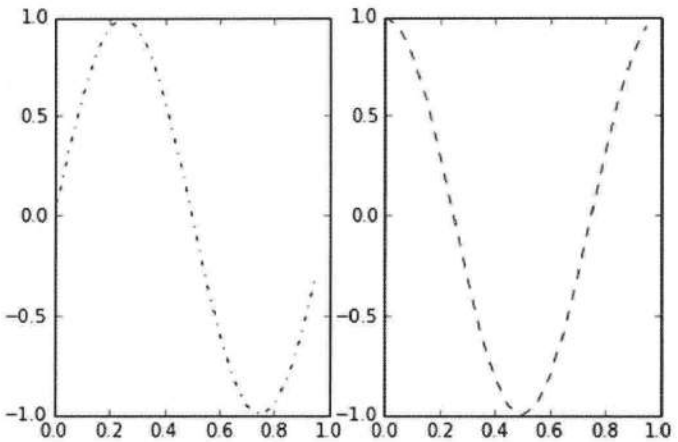


图7-13　分为左右两个子图的图形

7.7　为图表添加更多元素

为了使图表的信息更加丰富，很多时候会用线条或符号表示数据，用两条轴指定数值范围。其实仅仅这样做，表现力还是不足。为了添加额外信息，丰富图表，我们还可以向图表中添加更多元素。

这一节将学习如何为图表添加文字标签、图例等元素。

7.7.1　添加文本

标题的添加方法前面已讲过，用title()函数即可。另外两个很重要也需要添加到图表中的文本标识为轴标签。xlabel()和ylabel()函数专门用于添加轴标签。把要显示的文本以字符串形式传给这两个函数作为参数。

注意　生成图表所需命令的行数在逐渐增加。你不需要每次都重写这些命令，使用键盘上的方向键，可以找到之前使用过的命令，再加上新的命令（在下面的代码中用粗体表示）。

现在把两条轴的标签添加到图表中，它们描述的是坐标轴数值的含义（见图7-14）。

```
In [10]: plt.axis([0,5,0,20])
  ...: plt.title('My first plot')
  ...: plt.xlabel('Counting')
  ...: plt.ylabel('Square values')
  ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
```

```
Out[10]: [<matplotlib.lines.Line2D at 0x990f3c8>]
```

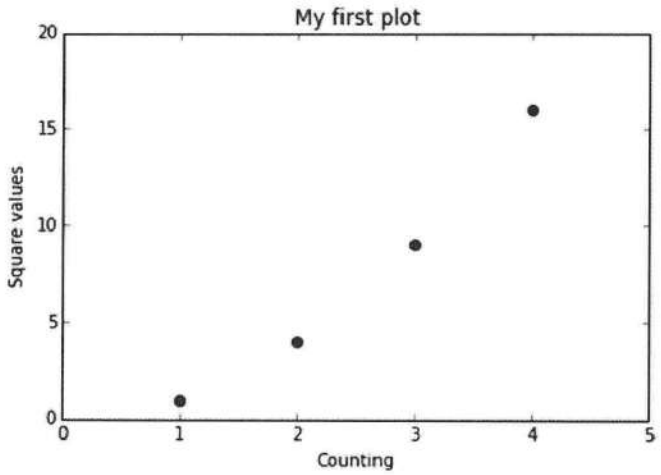


图7-14 添加坐标轴标签后的图形，信息更为丰富

我们可以用关键字参数修改文本属性。例如，你可以修改标题的字体，使用更大的字号。你还可以指定轴标签的颜色为灰色，从而反衬出图形的标题（见图7-15）。

```
In [ ]: plt.axis([0,5,0,20])
  ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
  ...: plt.xlabel('Counting',color='gray')
  ...: plt.ylabel('Square values',color='gray')
  ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[116]: [<matplotlib.lines.Line2D at 0x11f17470>]
```

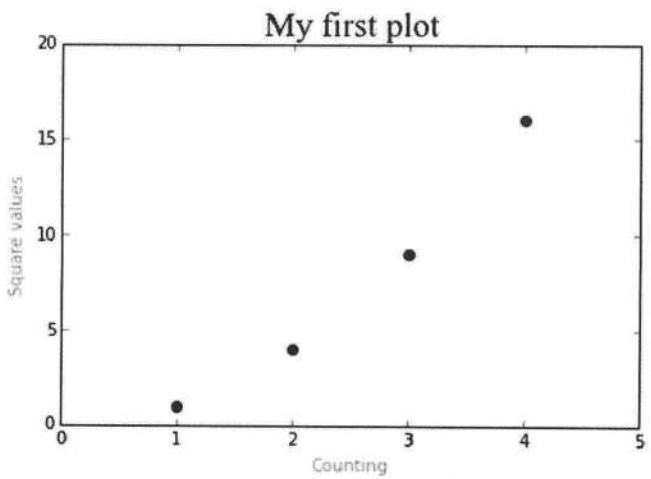


图7-15 设置关键字参数，可修改文本样式

但是，matplotlib的功能不只限于此：pyplot允许你在图表任意位置添加文本。这个功能由text()函数来实现。

```
text(x,y,s, fontdict=None, **kwargs)
```

前两个参数为文本在图形中位置的坐标。s为要添加的字符串，fontdict（可选）为文本要使用的字体。最后，还可以使用关键字参数。

接下来，我们为图形的各个数据点添加标签。text()函数的前两个参数为标签在图形中位置的坐标，所以我们可以使用四个数据点的坐标作为各标签的坐标，但每个标签的*y*值较相应数据点的*y*值有一点偏差。

```
In [ ]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.text(1,1.5,'First')
   ...: plt.text(2,4.5,'Second')
   ...: plt.text(3,9.5,'Third')
   ...: plt.text(4,16.5,'Fourth')
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[108]: [<matplotlib.lines.Line2D at 0x10f76898>]
```

在图7-16中，每个数据点都带有一个描述其相关信息的标签。

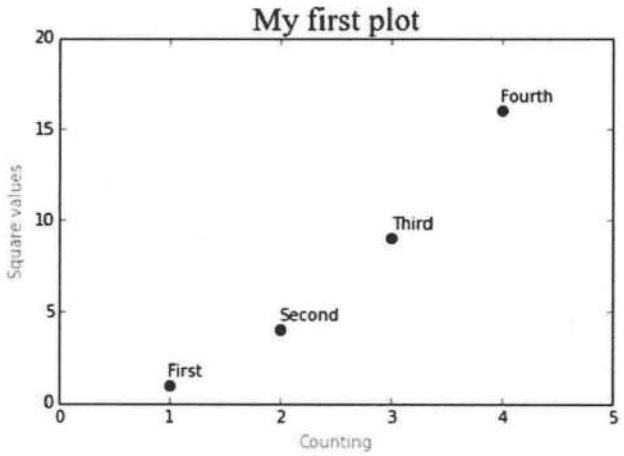


图7-16　图形每个数据点都有一个表示其含义的标签

既然matplotlib是专门为科学圈开发的图形库，它必须能够充分利用包含数学表达式在内的科学语言的潜能。matplotlib整合了LaTeX表达式，支持在图表中插入数学表达式。

将表达式内容置于两个$符号之间，可在文本中添加LaTeX表达式。解释器会将该符号之间的文本识别成LaTeX表达式，把它们转换为数学表达式、公式、数学符号或希腊字母等，然后在图像中显示出来。通常，你需要在包含LaTeX表达式的字符串前添加r字符，表明它后面是原始文本，不能对其进行转义操作。

你还可以使用关键字参数进一步丰富图形中的文本。例如，添加描述图形各数据点趋势的公式，并为公式添加一个彩色边框（见图7-17）。

```
In [ ]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.text(1,1.5,'First')
   ...: plt.text(2,4.5,'Second')
   ...: plt.text(3,9.5,'Third')
   ...: plt.text(4,16.5,'Fourth')
   ...: plt.text(1.1,12,r'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')

Out[130]: [<matplotlib.lines.Line2D at 0x13920860>]
```

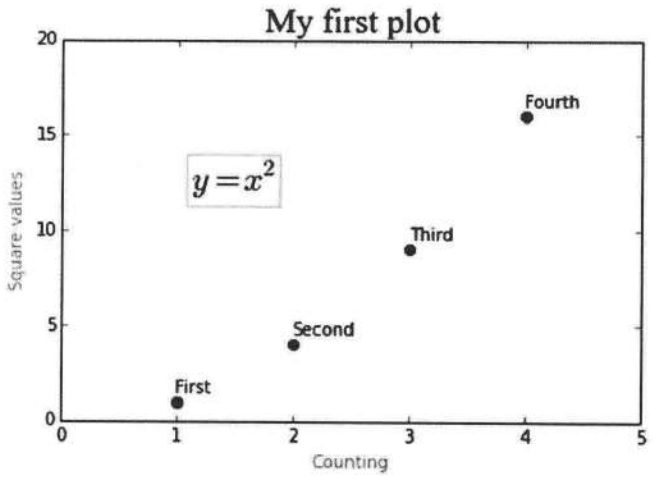


图7-17 图表中可以添加数学公式

如果要全面了解LaTeX的功能，请参考本书后面的附录A。

7.7.2 添加网格

图形中可以添加的另一个元素是网格。添加网格能更好地理解图表每个数据点的位置，因此往往很有必要。

在图表中添加网格其实很简单：直接在代码中加入grid()函数，传入参数True（见图7-18）。

```
In [ ]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.text(1,1.5,'First')
   ...: plt.text(2,4.5,'Second')
   ...: plt.text(3,9.5,'Third')
```

```
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,r'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
...: plt.grid(True)
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[108]: [<matplotlib.lines.Line2D at 0x10f76898>]
```

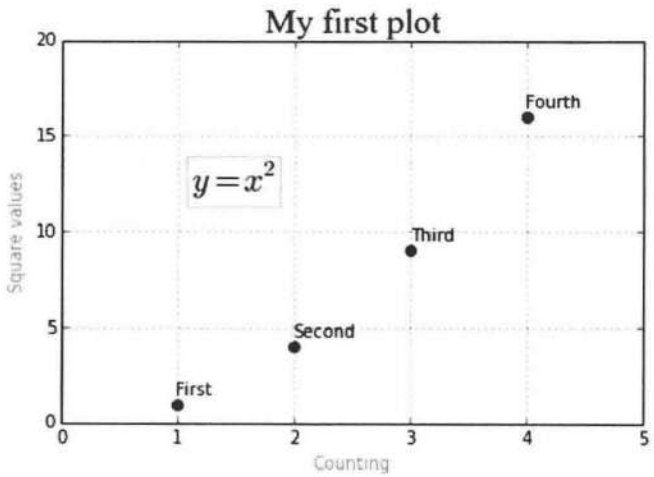


图7-18　有了网格，读取图表数据点的数值更简单

7.7.3　添加图例

另外一个任何图表都应该有的元素是图例。pyplot专门提供了legend()函数，用于操作该类对象。

请使用legend()函数将图例和字符串类型的图例说明添加到图表中。下面这个例子中，我们把输入的四个数据点统称为“First series”（序列一）（见图7-19）。

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
...: plt.grid(True)
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
...: plt.legend(['First series'])
Out[156]: <matplotlib.legend.Legend at 0x16377550>
```

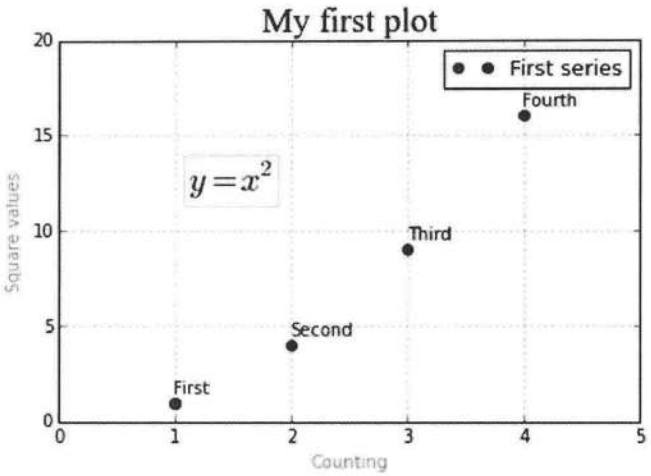


图7-19 图例默认添加到图形的右上角

如图7-19所示，图例默认添加到图形的右上角。如果要修改图例的位置，还是需要添加几个关键字参数。例如，图例的位置由loc关键字控制，其取值范围为0~10。每个数字代表图表中的一处位置（表7-1），默认为1，也就是右上角位置。下一个例子中，你需要把图例移到左上角，以免遮住数据点。

表7-1 关键字参数loc所有可能的取值

位置编号	位置表述
0	最佳位置
1	右上角
2	左上角
3	右下角
4	左下角
5	右侧[①]
6	左侧垂直居中
7	右侧垂直居中
8	下方水平居中
9	上方水平居中
10	正中间

修改位置编号以移动图例之前，我想先提醒你一下。通常，图例通过标签、颜色和（或）符号向读者表明这是哪一个序列，应该与其他序列区分开来。在前面所有的例子中，我们只使用了由一个plot()函数绘制的单个序列。现在，我们来看看更常见的情况，在一幅图中同时显示多个序列。图表中每个序列用一种特定的颜色和符号来表示（见图7-20）。从代码实现的角度来看，

① 5和7表示的位置其实相同。

每个序列都要调用一次plot()函数，调用顺序跟传给legend()函数作为参数的文本标签顺序应保持一致。

```
In [ ]: import matplotlib.pyplot as plt
   ...: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.text(1,1.5,'First')
   ...: plt.text(2,4.5,'Second')
   ...: plt.text(3,9.5,'Third')
   ...: plt.text(4,16.5,'Fourth')
   ...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
   ...: plt.grid(True)
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
   ...: plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
   ...: plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
   ...: plt.legend(['First series','Second series','Third series'],loc=2)
Out[170]: <matplotlib.legend.Legend at 0x1828d7b8>
```

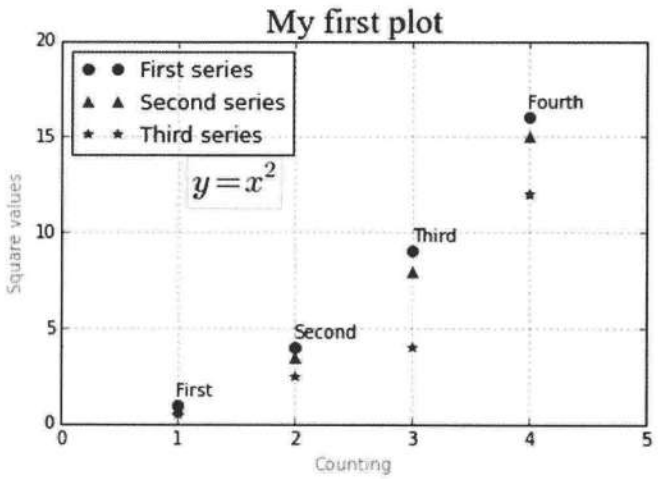


图7-20　多个序列的图表，图例就很有必要

7.8　保存图表

这一节，我们来看一下如何根据使用目的，以不同的方式保存图表。如要在不同的IPython Notebook文件、Python会话中重建图表，或以后想用于其他项目，最好把Python代码保存下来。若是要做报告或演示，最好将图表保存为图片。此外，如要将工作成果发布到网上，还可以把图表保存为HTML页面。

7.8.1 保存代码

从前几节的例子可知，生成单个图表所需代码的行数已增长到一个不小的数目。在开发过程中每实现一个里程碑式的功能，就可以把所有的代码保存到一个.py文件里，以便随时调用。

你可以使用%save魔术命令，后面跟着你想使用的文件名和代码对应的命令提示符号码。如果所有的代码写在一个命令提示符后，比如我们这里，只需使用这个号码即可；否则，如果要保存多个命令提示符后的代码，需要用连字符“-”连接两个数字，比如从10到20，就要写成10-20。

这里，生成第一幅图形的代码在命令提示符号码171后面，我们需要把这些代码保存起来。

```
In [171]: import matplotlib.pyplot as plt
...
```

用以下命令把代码保存到一个新.py文件中。

```
%save my_first_chart 171
```

运行上述命令后，会在工作目录生成my_first_chart.py文件（见代码清单7-1）。

代码清单7-1 my_first_chart.py

```
# coding: utf-8
import matplotlib.pyplot as plt
plt.axis([0,5,0,20])
plt.title('My first plot',fontsize=20,fontname='Times New Roman')
plt.xlabel('Counting',color='gray')
plt.ylabel('Square values',color='gray')
plt.text(1,1.5,'First')
plt.text(2,4.5,'Second')
plt.text(3,9.5,'Third')
plt.text(4,16.5,'Fourth')
plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
plt.grid(True)
plt.plot([1,2,3,4],[1,4,9,16],'ro')
plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
plt.legend(['First series','Second series','Third series'],loc=2)
```

稍后，打开新IPython会话，输入以下命令，可把图表恢复到最后一次保存时的状态，之后你就可以在此基础上继续修改代码了。

```
ipython qtconsole --matplotlib inline -m my_first_chart.py
```

你还可以在QtConsole中使用%load魔术命令加载所有代码。

```
%load my_first_chart.py
```

或者用%run魔术命令，在会话中运行代码。

```
%run my_first_chart.py
```

注意 在我的系统中，上面这条命令须等它前面的两条命令运行后方能运行。[1]

① 在笔者所用的Anaconda中，无须运行前两条命令，可直接运行%run命令。

7.8.2　将会话转换为 HTML 文件

如果使用QtConsole，你还能把当前会话中的所有代码和图形转换为HTML页面：从菜单中依次选择File ➤ Save to HTML / XHTML（见图7-21）。

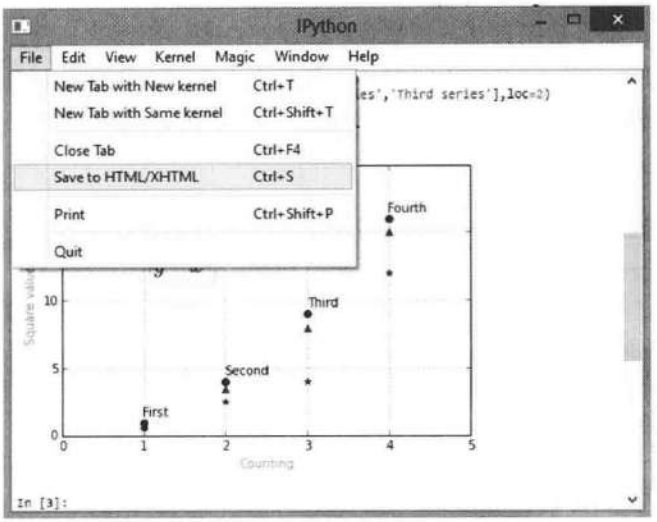


图7-21　把当前会话保存为网页

系统会询问你想把会话保存为哪种格式的网页：HTML或XHTML。这两种格式的区别在于将图形转换成哪种类别的图像。选择HTML作为输出格式，会话中的图形将被转换为PNG格式；而选择XHTML作为输出格式，图形将被转换为SVG格式。

下面这个例子中，我们把会话保存为HTML文件my_session.html，如图7-22所示。

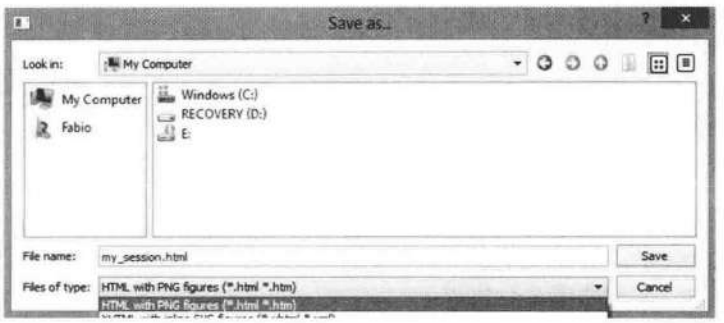


图7-22　会话可以保存成HTML或XHTML文件

这时，系统会询问你是想把图片保存到外部目录还是在行内显示[①]（见图7-23）。

① 只生成一个html文件，图像使用base64编码。

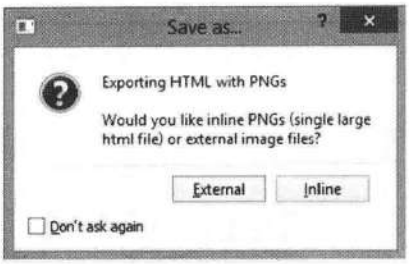


图7-23 你可以创建外部图片文件或把PNG格式图片直接嵌入到HTML页面

选择外部选项的话，图表中的所有图像会汇总在一起，置于my_session_files的目录下；反之，选择另一个选项，图片的相关信息将会被嵌入到HTML代码中。

7.8.3 将图表直接保存为图片

如果只想把图表保存为图像文件，可以忽略会话中输入的所有代码。实际上，你可以用savefig()函数直接把图表保存为PNG格式，但是请记得把这个函数添加到用于生成图表的一系列命令的最后（否则将得到一个空白PNG文件[①]）。

```
In [ ]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.text(1,1.5,'First')
   ...: plt.text(2,4.5,'Second')
   ...: plt.text(3,9.5,'Third')
   ...: plt.text(4,16.5,'Fourth')
   ...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
   ...: plt.grid(True)
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
   ...: plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
   ...: plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
   ...: plt.legend(['First series','Second series','Third series'],loc=2)
   ...: plt.savefig('my_chart.png')
```

执行上述代码，工作目录中会生成一个新文件my_chart.png，该图像文件的内容即是图表。

7.9 处理日期值

数据分析过程中，最常见的一个问题就是日期类型数据的处理。在轴上（通常为*x*轴）显示日期，问题很多，尤其是用日期做标签时难以管理（见图7-24）。

① 注意，保存成图像的命令之前不要使用plt.show()，否则也将得到空白图像。

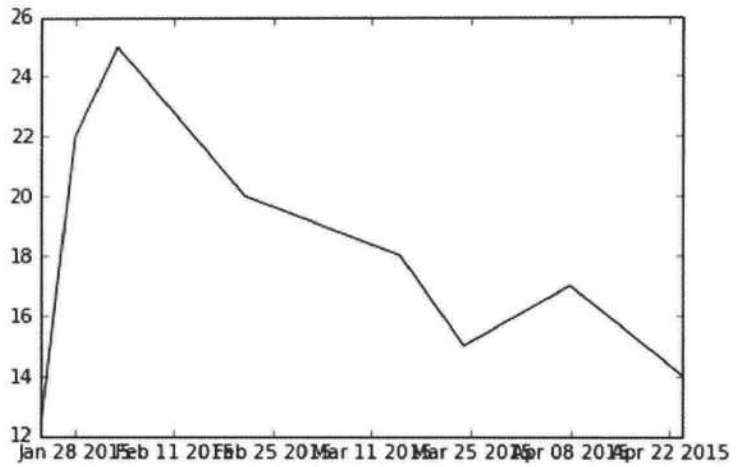


图7-24　未经处理，日期数据的显示会有问题

举个例子，线性图表有八个数据点，我们按照日-月-年格式在*x*轴显示日期值。

```
In [ ]: import datetime
   ...: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...: events = [datetime.date(2015,1,23),datetime.date(2015,1,28),datetime.
        date(2015,2,3),datetime.date(2015,2,21),datetime.date(2015,3,15),datetime.
        date(2015,3,24),datetime.date(2015,4,8),datetime.date(2015,4,24)]
   ...: readings = [12,22,25,20,18,15,17,14]
   ...: plt.plot(events,readings)
Out[83]: [<matplotlib.lines.Line2D at 0x12666400>]
```

由图7-24所见，如果让matplotlib自动管理刻度，尤其是刻度的标签，其后果无疑是一场灾难。以这种方式显示日期时，可读性很差，两个数据点之间的时间间隔不清晰，并且还存在重影现象。

建议用合适的对象，定义时间尺度来管理日期。首先需要导入matplotlib.dates模块，该模块专门用于管理日期类型的数据。然后，定义时间尺度。这个例子中，我们用MonthLocator()和DayLocator()函数，分别表示月份和日子。日期格式也很重要，要避免出现重影问题或显示无效的日期数据，只显示必要的刻度标签就好。我们这里只显示年月，把这种格式作为参数传给DateFormatter()函数。

定义好两个时间尺度，一个用于日期，一个用于月份。你可以在xaxis对象上调用set_major_locator()函数和set_minor_locator()函数，为*x*轴设置两种不同的标签。此外，月份刻度标签的设置，需要用到set_major_formatter()函数。

使用上述设置，最终可以得到图7-25所示的图像。

```
In [ ]: import datetime
   ...: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...: import matplotlib.dates as mdates
   ...: months = mdates.MonthLocator()
```

```
...: days = mdates.DayLocator()
...: timeFmt = mdates.DateFormatter('%Y-%m')
...: events = [datetime.date(2015,1,23),datetime.date(2015,1,28),datetime.
     date(2015,2,3),datetime.date(2015,2,21),datetime.date(2015,3,15),datetime.
     date(2015,3,24),datetime.date(2015,4,8),datetime.date(2015,4,24)]
     readings = [12,22,25,20,18,15,17,14]
...: fig, ax = plt.subplots()
...: plt.plot(events,readings)
...: ax.xaxis.set_major_locator(months)
...: ax.xaxis.set_major_formatter(timeFmt)
...: ax.xaxis.set_minor_locator(days)
```

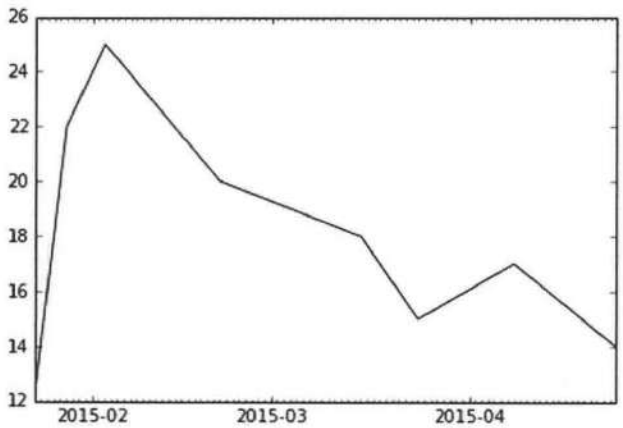


图7-25 x轴只显示月份的刻度标签，可读性更强

7.10 图表类型

前几节的例子多与matplotlib架构相关。至此，想必你已熟悉图表主要图像元素的用法，接下来再通过多个例子讲解不同类型图表的制作方法。我们先讲解线性图、条状图和饼状图，随后再介绍几种更为复杂却很常用的图表。

本章这一部分内容的重要性自不必多言，因为matplotlib库的目的就是将数据分析的结果可视化。因而为数据选择合适的图表类型是一项基本技能。请记住，即使数据分析结果再出色，选用的图表类型如不恰当，也将会导致对实验结果做出错误的解释。

7.11 线性图

在图表的所有类型中，线性图最为简单。线性图的各个数据点由一条线来连接。一对对(x,y)值组成的数据点在图表中的位置取决于两条轴（x和y）的刻度范围。

举例来说，你可以绘制由数学函数生成的数据点。比如为一个普通函数的图像作图。

$$y = \sin(3 * x) / x$$

如果要绘制一系列数据点，需要创建两个NumPy数组。首先，创建包含x值的数组，用作x轴。我们使用np.arange()函数定义一个元素依次递增的序列。这个数学函数是正弦函数，因此x应取希腊字母pi（np.pi）的倍数或是因数。然后，用np.sin()函数可直接求得这一列x值所对应的y值（多亏了NumPy!）。

完成上述运算后，只需调用plot()函数绘制图像即可。你将得到如图7-26所示的图像。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: plt.plot(x,y)
Out[393]: [<matplotlib.lines.Line2D at 0x22404358>]
```

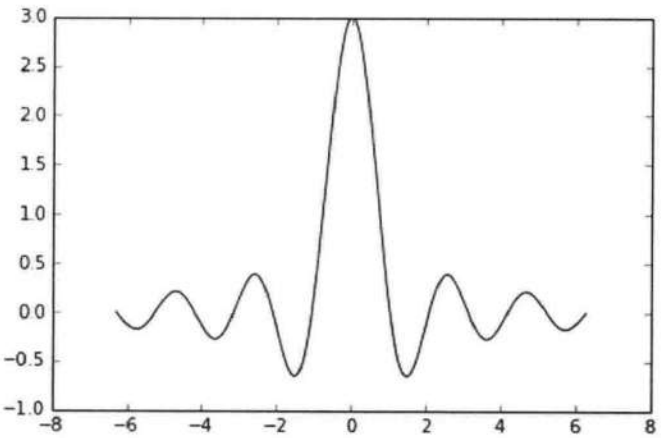


图7-26　用线性图表示的数学函数

你可以扩展这个例子，显示像下面这样一组函数的图像。

$$y = \sin(n * x) / x$$

改变n的取值即可。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: y2 = np.sin(2*x)/x
   ...: y3 = np.sin(3*x)/x
   ...: plt.plot(x,y)
   ...: plt.plot(x,y2)
   ...: plt.plot(x,y3)
```

如图7-27所示，matplotlib自动为每条线分配一种不同的颜色。几个函数的图像使用相同的刻度范围；换句话说，每个序列的数据点使用相同的x轴和y轴。Figure对象会记录先前的命令，每次调用plot()函数都会考虑之前是怎么调用的，并根据函数的调用方法实现相应的效果，直到该

对象不再显示为止（Python会话用show()显示图像，IPython QtConsole按Enter键显示图像）。

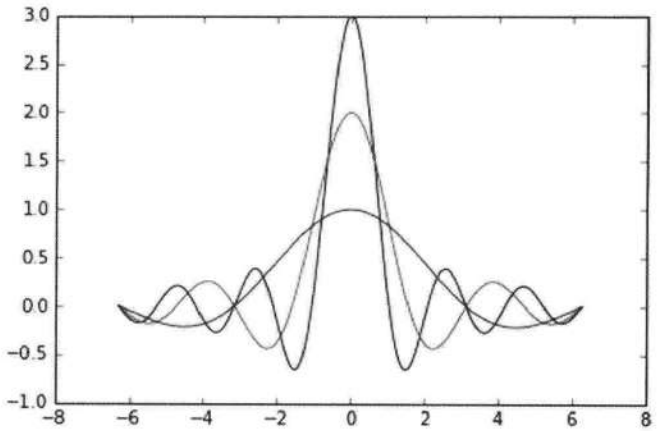


图7-27 在同一图表中用三种不同的颜色绘制三个序列数据点

前几节中，不管默认设置如何，我们总是可以自己选择线型、颜色等。你可以用plot()函数的第三个参数指定颜色（表7-2）、线型，把这些设置所用的字符编码放到同一个字符串即可。你还可以使用两个单独的关键字参数，用color指定颜色，用linestyle指定线型（见图7-28）。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: y2 = np.sin(2*x)/x
   ...: y3 = np.sin(3*x)/x
   ...: plt.plot(x,y,'k--',linewidth=3)
   ...: plt.plot(x,y2,'m-.')
   ...: plt.plot(x,y3,color='#87a3cc',linestyle='--')
```

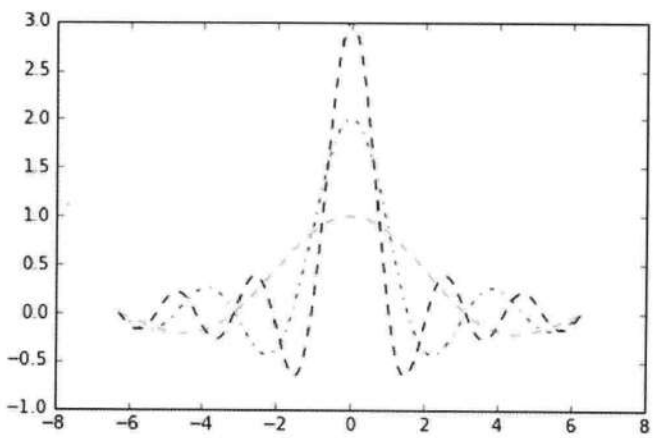


图7-28 用字符编码指定颜色和线型

表7-2　颜色编码

编　码	颜　色
b	蓝色
g	绿色
r	红色
c	蓝绿色
m	洋红
y	黄色
k	黑色
w	白色

*x*轴的数值范围为–2π ~ 2π，但是刻度标签默认使用数值形式。你需要用π的倍数代替数值。同理，你也可以替换*y*轴刻度的标签。方法是使用xticks()和yticks()函数，分别为每个函数传入两列数值。第一个列表存储刻度的位置，第二个列表存储刻度的标签。这个例子中，要正确显示符号π，需要使用含有LaTeX表达式的字符串。记得将其置于两个$之中，并在前面加上r前缀。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: y2 = np.sin(2*x)/x
   ...: y3 = np.sin(x)/x
   ...: plt.plot(x,y,color='b')
   ...: plt.plot(x,y2,color='r')
   ...: plt.plot(x,y3,color='g')
   ...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
           [r'$-2\pi$',r'$-\pi$',r'$0$',r'$+\pi$',r'$+2\pi$'])
   ...: plt.yticks([-1,0,+1,+2,+3],
           [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
Out[423]:
([<matplotlib.axis.YTick at 0x26877ac8>,
  <matplotlib.axis.YTick at 0x271d26d8>,
  <matplotlib.axis.YTick at 0x273c7f98>,
  <matplotlib.axis.YTick at 0x273cc470>,
  <matplotlib.axis.YTick at 0x273cc9e8>],
<a list of 5 Text yticklabel objects>)
```

最后，你将得到一幅清晰明了的线性图，其中轴标签使用了希腊字母，如图7-29所示。

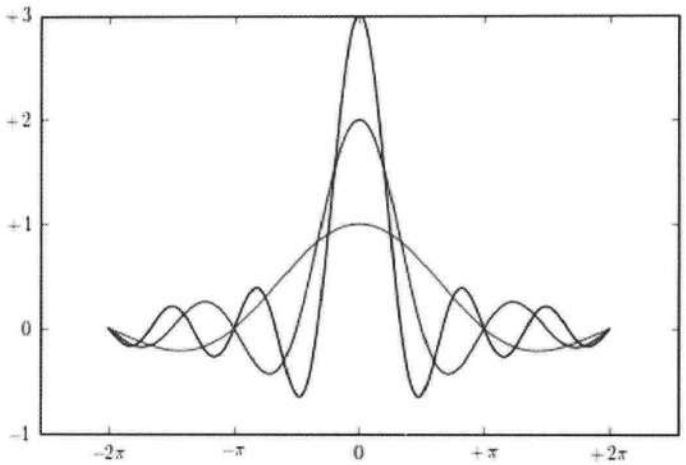


图7-29 添加LaTeX格式的文本，改进刻度标签显示效果

在前面你所见过的所有线性图中，*x*轴和*y*轴总是置于Figure的边缘（跟图像的边框重合）。另外一种显示轴的方法是，两条轴穿过原点(0,0)，也就是笛卡儿坐标轴。

具体做法是，首先必须用gca()函数获取Axes对象。接着通过这个对象，指定每条边的位置：右、左、下和上，可选择组成图形边框的每条边。使用set_color()函数，把颜色设置为none，删除跟坐标轴不符合的边（右和上）。然后，用set_position()函数移动跟*x*轴和*y*轴相符的边框，使其穿过原点(0,0)。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: y2 = np.sin(2*x)/x
   ...: y3 = np.sin(x)/x
   ...: plt.plot(x,y,color='b')
   ...: plt.plot(x,y2,color='r')
   ...: plt.plot(x,y3,color='g')
   ...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
          [r'$-2\pi$',r'$-\pi$',r'$0$',r'$+\pi$',r'$+2\pi$'])
   ...: plt.yticks([-1,0,+1,+2,+3],
          [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
   ...: ax = plt.gca()
   ...: ax.spines['right'].set_color('none')
   ...: ax.spines['top'].set_color('none')
   ...: ax.xaxis.set_ticks_position('bottom')
   ...: ax.spines['bottom'].set_position(('data',0))
   ...: ax.yaxis.set_ticks_position('left')
   ...: ax.spines['left'].set_position(('data',0))
```

这时，两条轴在图表中部位置交叉，这个位置就是笛卡儿坐标系的原点，如图7-30所示。

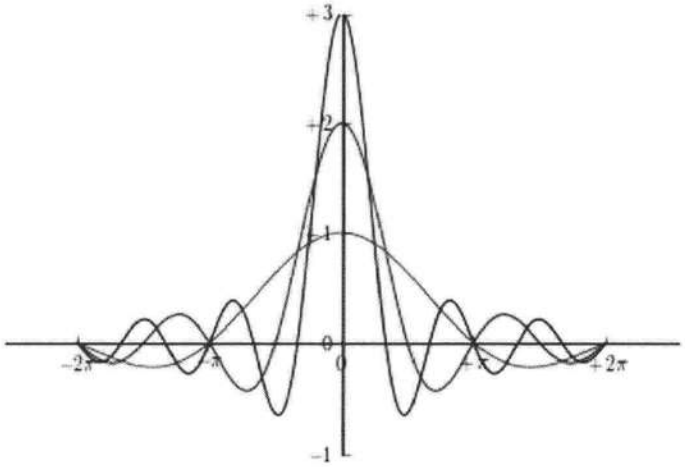


图7-30　图表中有两条笛卡儿坐标轴

用注释和箭头(可选)标明曲线上某一数据点的位置，这一功能非常有用。例如，可以用LaTeX表达式作为注释，比如添加表示x趋于0时函数$\sin x/x$的极限的公式。

matplotlib库的annotate()函数特别适用于添加注释。虽然它有多个关键字参数，这些参数可改善显示效果，但是参数设置略显繁琐，也就使得annotate()函数看似有些麻烦。第一个参数为含有LaTeX表达式、要在图形中显示的字符串；随后是各种关键字参数。注释在图表中的位置用存放数据点[x,y]坐标的列表来表示，需把它们传给xy关键字参数。文本注释跟它所解释的数据点之间的距离用xytext关键字参数指定，用曲线箭头将其表示出来。箭头的属性则由arrowprops关键字参数指定。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: y2 = np.sin(2*x)/x
   ...: y3 = np.sin(x)/x
   ...: plt.plot(x,y,color='b')
   ...: plt.plot(x,y2,color='r')
   ...: plt.plot(x,y3,color='g')
   ...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
           [r'$-2\pi$',r'$-\pi$',r'$0$',r'$+\pi$',r'$+2\pi$'])
   ...: plt.yticks([-1,0,+1,+2,+3],
           [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
...: plt.annotate(r'$\lim_{x\to 0}\frac{\sin(x)}{x}= 1$', xy=[0,1],xycoords='data',
     xytext=[30,30],fontsize=16,textcoords='offset points',arrowprops=dict(arrowstyle="->",
     connectionstyle="arc3,rad=.2"))
   ...: ax = plt.gca()
   ...: ax.spines['right'].set_color('none')
   ...: ax.spines['top'].set_color('none')
   ...: ax.xaxis.set_ticks_position('bottom')
   ...: ax.spines['bottom'].set_position(('data',0))
```

```
...: ax.yaxis.set_ticks_position('left')
...: ax.spines['left'].set_position(('data',0))
```

运行上述代码，你就会得到带有极限公式的图表。该公式所表示的数据点即是图7-31中箭头所指向的。

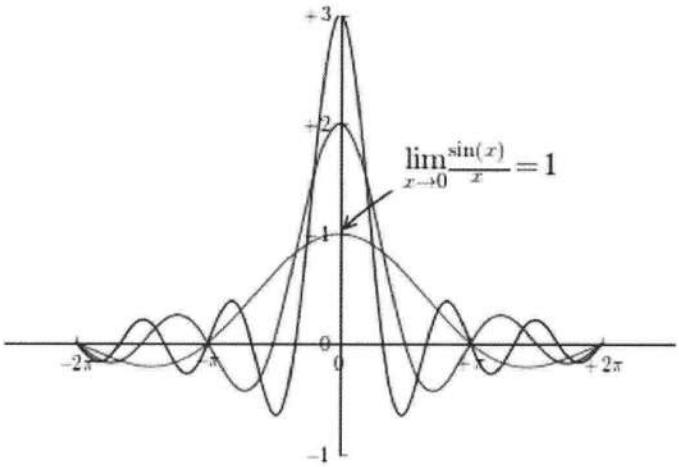


图7-31 数学表达式可以用annotate()函数添加到图表中

为 pandas 数据结构绘制线性图

接下来请移步更实用的场景，或者至少说是跟数据分析紧密相关的场景。我们来见识一下用matplotlib库把pandas库的DataFrame数据结构绘制成图表是多么容易。将DataFrame中的数据做成线性图表很简单，只需把DataFrame作为参数传入plot()函数，就能得到多序列线性图（见图7-32）。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: import pandas as pd
   ...: data = {'series1':[1,3,4,3,5],
                'series2':[2,4,5,2,4],
                'series3':[3,2,3,1,3]}
   ...: df = pd.DataFrame(data)
   ...: x = np.arange(5)
   ...: plt.axis([0,5,0,7])
   ...: plt.plot(x,df)
   ...: plt.legend(data, loc=2)
```

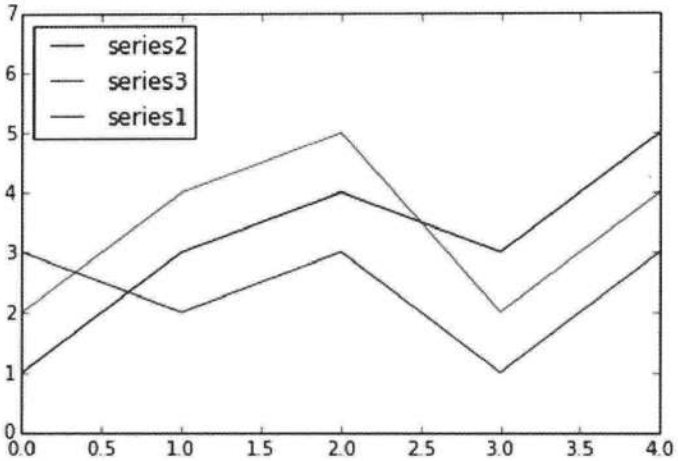


图7-32　将pandas DataFrame中的数据做成三个序列图表

7.12　直方图

直方图由竖立在*x*轴上的多个相邻的矩形组成，这些矩形把*x*轴拆分为一段段彼此不重叠的线段（线段两个端点所标识的数据范围也叫面元），矩形的面积跟落在其所对应的面元的元素数量成正比。这种可视化方法常被用于样本分布等统计研究。

pyplot用于绘制直方图的函数为hist()，该函数具有一个其他绘图函数所没有的功能。它除了绘制直方图外，还以元组形式返回直方图的计算结果。事实上，hist()函数还可以实现直方图的计算。也就是说，它能够接收一系列样本个体和期望的面元数量作为参数，会把样本范围分成多个区间（面元），然后计算每个面元所包含的样本个体的数量。运算结果除了以图形形式（见图7-33）表示外，还能以元组形式返回。

```
(n, bins, patches)
```

要理解上述操作，最好的办法莫过于看一个实际的例子。首先使用random.randint()函数生成100个0~100的随机数作为样本。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: pop = np.random.randint(0,100,100)
   ...: pop
Out[ ]:
array([32, 14, 55, 33, 54, 85, 35, 50, 91, 54, 44, 74, 77,  6, 77, 74, 2,
       54, 14, 30, 80, 70,  6, 37, 62, 68, 88,  4, 35, 97, 50, 85, 19, 90,
       65, 86, 29, 99, 15, 48, 67, 96, 81, 34, 43, 41, 21, 79, 96, 56, 68,
       49, 43, 93, 63, 26,  4, 21, 19, 64, 16, 47, 57,  5, 12, 28, 7, 75,
        6, 33, 92, 44, 23, 11, 61, 40,  5, 91, 34, 58, 48, 75, 10, 39, 77,
       70, 84, 95, 46, 81, 27,  6, 83,  9, 79, 39, 90, 77, 94, 29])
```

现在，我们把刚生成的样本数据作为参数传给hist()函数，创建一个直方图。例如，你想把

样本个体分到20个面元中（如未指定，默认分为10个面元），关键字参数bin的值就为20（见图7-33）。

```
In [ ]: n,bins,patches = plt.hist(pop,bins=20)
```

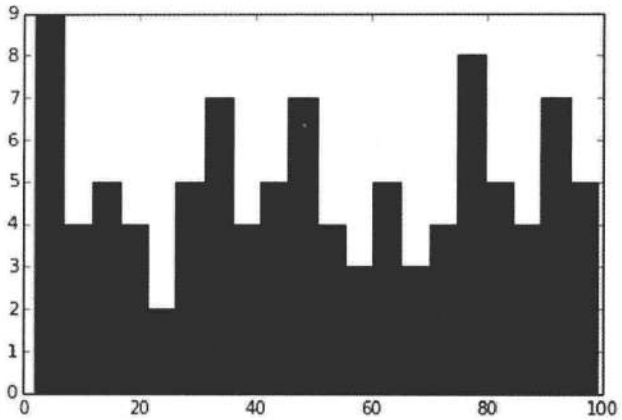


图7-33 显示每个面元个体数量的直方图

7.13 条状图

另外一种常用的图表类型为条状图。它跟直方图很相似，只不过x轴表示的不是数值而是类别。用matplotlib的bar()函数生成条状图很简单。

```
In [ ]: import matplotlib.pyplot as plt
   ...: index = [0,1,2,3,4]
   ...: values = [5,7,3,4,6]
   ...: plt.bar(index,values)
Out[15]: <Container object of 5 artists>
```

用上述几行代码就能绘制出如图7-34所示的条状图。

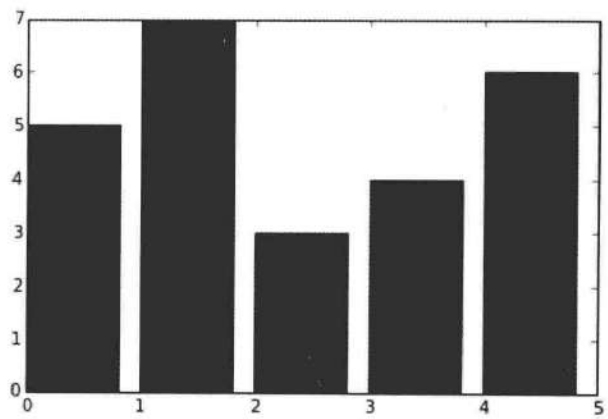


图7-34 用matplotlib实现的最简单的条状图

如图7-34所见，*x*轴上所有的标签显示在每个长条的左下角。但是由于每个长条对应的是一种类别，最好用刻度标签标明其类别，方法是把表示各个类别的字符串传递给xticks()参数。至于刻度标签的位置，你需要把表示它们在*x*轴上位置的数值列表传递给xticks()函数，作为它的第一个参数。最终，将得到如图7-35所示的条状图。

```
In [ ]: import numpy as np
   ...: index = np.arange(5)
   ...: values1 = [5,7,3,4,6]
   ...: plt.bar(index,values1)
   ...: plt.xticks(index+0.4,['A','B','C','D','E'])
```

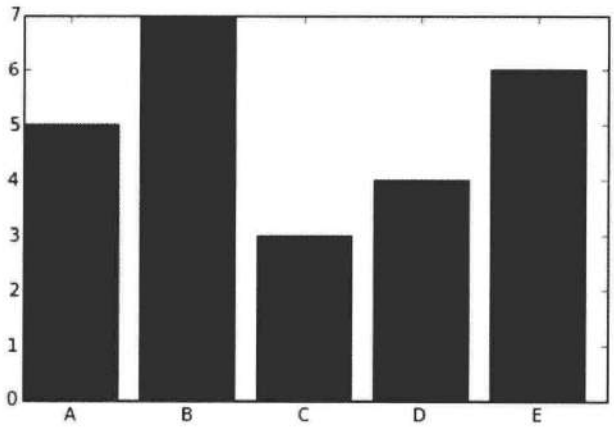


图7-35　*x*轴显示类别的简单条状图

实际上，我们还可以借助很多其他步骤进一步改进条状图。每一种改进方法都是通过在bar()函数中添加特定的关键字参数来实现的。例如，把包含标准差的列表传给yerr关键字参数，就能添加标准差。这个参数常跟error_kw参数一起使用，而后者又接收其他可用于显示误差线的关键字参数。常用的两个是eColor和capsize，eColor指定误差线的颜色，而capsize指定误差线两头横线的宽度。

还有一个参数叫作alpha，它控制的是彩色条状图的透明度。alpha的取值范围为0~1。0表示对象完全透明，随着alpha值的增加，对象逐渐清晰起来，到1时，颜色显示全了。

照例我建议你在图表中添加图例。请用label关键字参数，为图表中的序列指定名称。

最终，你将得到如图7-36所示的带有误差线的条状图。

```
In [ ]: import numpy as np
   ...: index = np.arange(5)
   ...: values1 = [5,7,3,4,6]
   ...: std1 = [0.8,1,0.4,0.9,1.3]
   ...: plt.title('A Bar Chart')
   ...: plt.bar(index,values1,yerr=std1,error_kw={'ecolor':'0.1',
        'capsize':6},alpha=0.7,label='First')
   ...: plt.xticks(index+0.4,['A','B','C','D','E'])
   ...: plt.legend(loc=2)
```

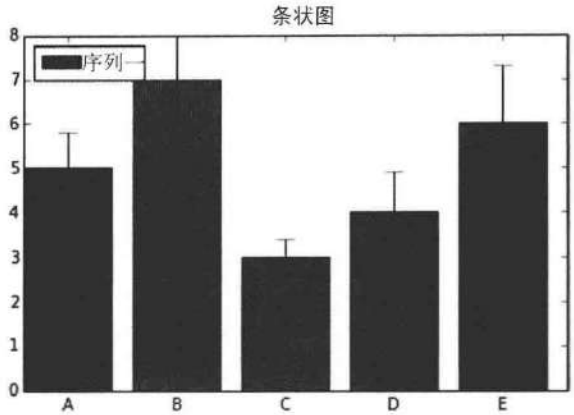


图7-36 带有误差线的条状图

7.13.1 水平条状图

前面讲的都是沿垂直方向排列的条状图，实际上还有水平方向的条状图。这种模式的条状图用barh()函数实现。bar()函数的参数和关键字参数对该函数依然有效。唯一需要注意的是，两条轴的用途跟垂直条状图刚好相反。水平条状图中，类别分布在*y*轴上，数值显示在*x*轴（见图7-37）。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: index = np.arange(5)
   ...: values1 = [5,7,3,4,6]
   ...: std1 = [0.8,1,0.4,0.9,1.3]
   ...: plt.title('A Horizontal Bar Chart')
   ...: plt.barh(index,values1,xerr=std1,error_kw={'ecolor':'0.1','capsize':6},alpha=0.7,
        label='First')
   ...: plt.yticks(index+0.4,['A','B','C','D','E'])
   ...: plt.legend(loc=5)
```

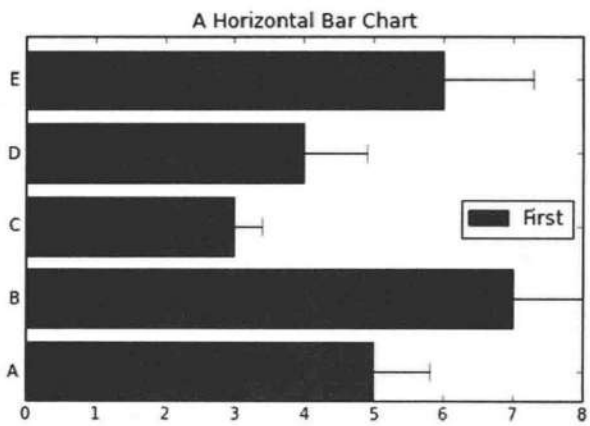


图7-37 简单的水平条状图

7.13.2　多序列条状图

条状图和线性图通常都是用来同时显示多个序列的数值。我们还是有必要说一说多序列条状图的组织方式。我们前面定义了一列索引值，把它们分配给*x*轴，每个值对应一条状图形，索引值代表类别。而接下来这个例子中，要求多个长条共用相同的类别。

解决方法是，把每个类别占据的空间（方便起见，宽度为1）分为多个部分。想显示几个长条，就将其分为几部分。建议再增加一个额外的空间，以便区分两个相邻的类别（见图7-38）。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: index = np.arange(5)
   ...: values1 = [5,7,3,4,6]
   ...: values2 = [6,6,4,5,7]
   ...: values3 = [5,6,5,4,6]
   ...: bw = 0.3
   ...: plt.axis([0,5,0,8])
   ...: plt.title('A Multiseries Bar Chart',fontsize=20)
   ...: plt.bar(index,values1,bw,color='b')
   ...: plt.bar(index+bw,values2,bw,color='g')
   ...: plt.bar(index+2*bw,values3,bw,color='r')
   ...: plt.xticks(index+1.5*bw,['A','B','C','D','E'])
```

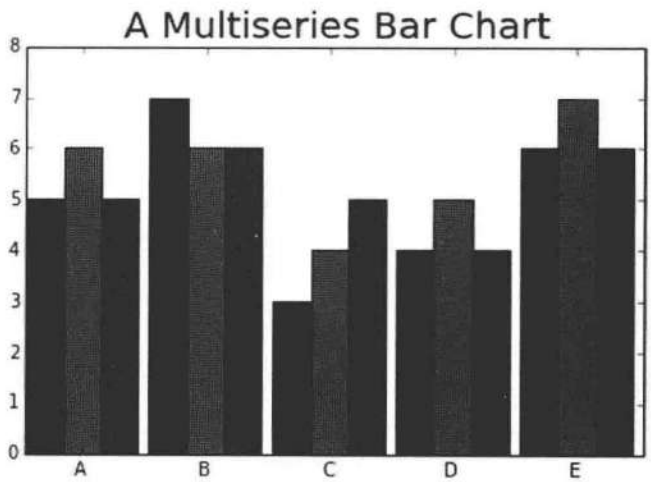


图7-38　多序列条状图：三个序列

多序列水平条状图（见图7-39）的生成方法也很简单。用barh()函数替换bar()函数，同时还要记得用yticks()函数替换为xticks()函数。此外，还需要交换axis()函数的参数中两条轴的取值范围。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: index = np.arange(5)
   ...: values1 = [5,7,3,4,6]
```

```
...: values2 = [6,6,4,5,7]
...: values3 = [5,6,5,4,6]
...: bw = 0.3
...: plt.axis([0,8,0,5])
...: plt.title('A Multiseries Horizontal Bar Chart',fontsize=20)
...: plt.barh(index,values1,bw,color='b')
...: plt.barh(index+bw,values2,bw,color='g')
...: plt.barh(index+2*bw,values3,bw,color='r')
...: plt.yticks(index+0.4,['A','B','C','D','E'])
```

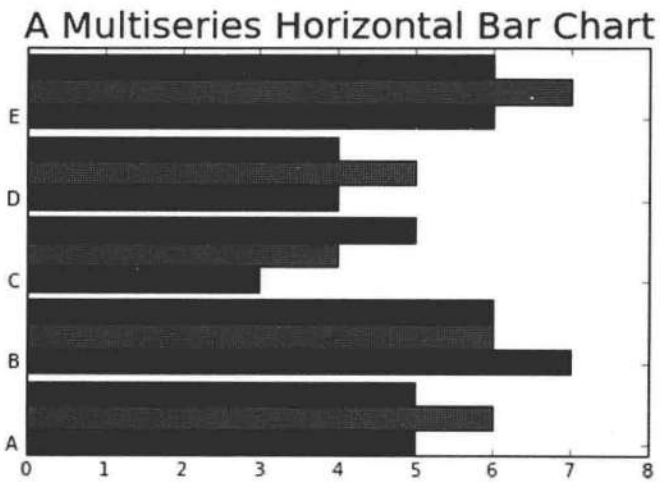


图7-39 多序列水平条状图

7.13.3 为 pandas DataFrame 生成多序列条状图

跟线性图那一节所讲的类似，matplotlib库还可以直接把存放数据分析结果的DataFrame对象做成条状图，甚至可以快速完成，实现自动化。你唯一需要做的就是在DataFrame对象上调用plot()函数，指定kind关键字参数，把图表类型赋给它，这里使用bar类型。无需其他设置，你就能得到如图7-40所示的条状图。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: import pandas as pd
   ...: data = {'series1':[1,3,4,3,5],
                'series2':[2,4,5,2,4],
                'series3':[3,2,3,1,3]}
   ...: df = pd.DataFrame(data)
   ...: df.plot(kind='bar')
```

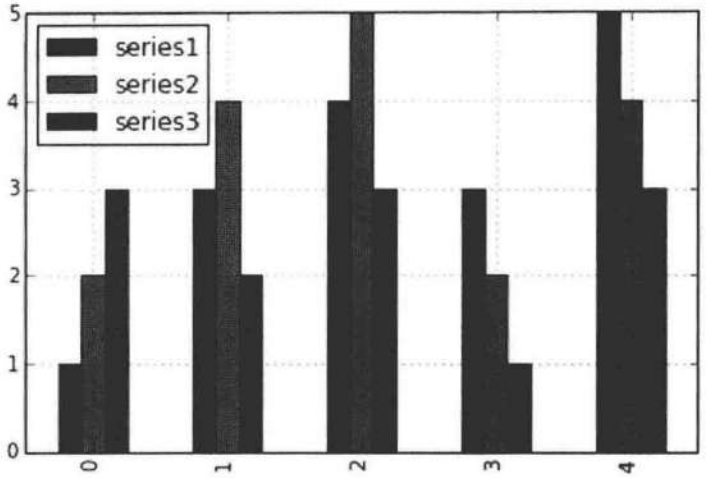


图7-40　DataFrame存储的数据可直接做成条状图

然而，如果想对图像生成过程拥有更多的控制权，或者需要对其进行改动，可以从DataFrame中抽取几部分数据，将其保存为NumPy数组，然后像本节前几个例子那样用这些数组绘图，把它们作为一个个单独的参数传递给matplotlib函数。

此外，同样的规则也适用于制作水平条状图，但记得把`barh`赋给`kind`关键字参数。你将得到如图7-41所示的多序列水平条状图。

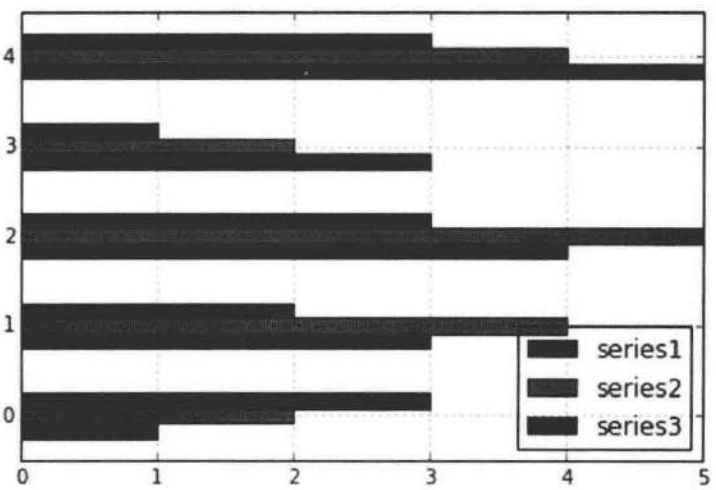


图7-41　水平条状图可以作为DataFrame数据的另外一种有效的可视化方法

7.13.4　多序列堆积条状图

多序列条状图的另外一种表现形式是堆积条状图，几个条状图形堆积在一起形成一个更大的

长条。如果想表示总和是由几个条状图相加得到的，堆积图就特别合适。

要把简单的多序列条状图转换为堆积图，需在每个bar()函数中添加bottom关键字参数，把每个序列赋给相应的bottom关键字参数。这样就能得到如图7-42所示的堆积条状图。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: series1 = np.array([3,4,5,3])
   ...: series2 = np.array([1,2,2,5])
   ...: series3 = np.array([2,3,3,4])
   ...: index = np.arange(4)
   ...: plt.axis([0,4,0,15])
   ...: plt.bar(index,series1,color='r')
   ...: plt.bar(index,series2,color='b',bottom=series1)
   ...: plt.bar(index,series3,color='g',bottom=(series2+series1))
   ...: plt.xticks(index+0.4,['Jan15','Feb15','Mar15','Apr15'])
```

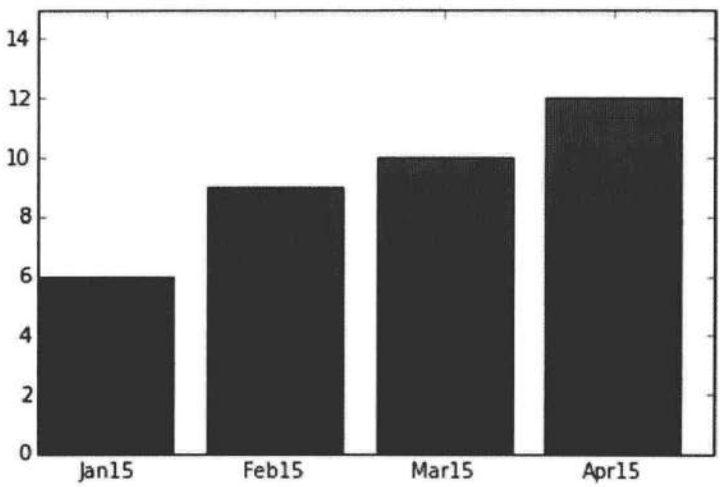


图7-42 多序列堆积条状图

同样，要创建相应的水平堆积条状图，则需用barh()函数替换bar()函数，记得同时修改其他参数。xticks()函数必须替换为yticks()函数，因为类别标签现在必须置于*y*轴之上。完成上述改动后，将得到如图7-43所示的水平堆积条状图。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: index = np.arange(4)
   ...: series1 = np.array([3,4,5,3])
   ...: series2 = np.array([1,2,2,5])
   ...: series3 = np.array([2,3,3,4])
   ...: plt.axis([0,15,0,4])
   ...: plt.title('A Multiseries Horizontal Stacked Bar Chart')
   ...: plt.barh(index,series1,color='r')
   ...: plt.barh(index,series2,color='g',left=series1)
   ...: plt.barh(index,series3,color='b',left=(series1+series2))
```

```
...: plt.yticks(index+0.4,['Jan15','Feb15','Mar15','Apr15'])
```

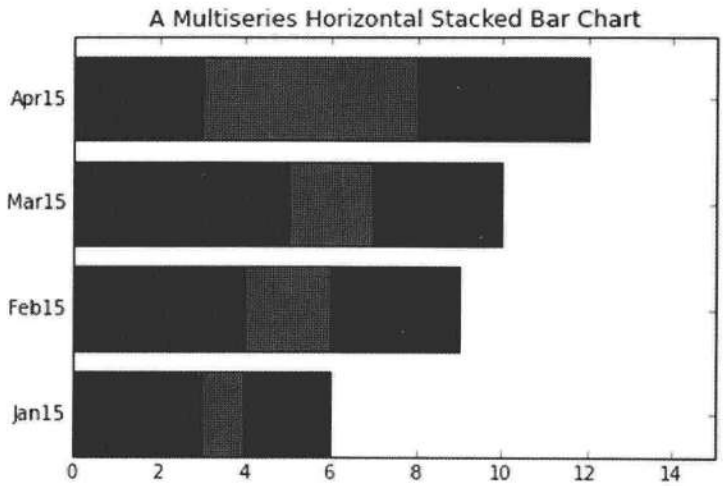


图7-43　多序列水平堆积条状图

前面我们一直用不同的颜色来区分多个序列，其实还可以用不同的影线填充条状图，方法如下。首先把条状图颜色设置为白色，然后用hatch关键字参数指定影线的类型。不同的影线使用不同的字符（|、/、-、\、*）表示，每种字符对应一种用来填充条状图形的线条类型。同一符号出现的次数越多，则形成阴影的线条越密集，例如，///比//密集，而//又比/密集，如图7-44所示。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: index = np.arange(4)
   ...: series1 = np.array([3,4,5,3])
   ...: series2 = np.array([1,2,2,5])
   ...: series3 = np.array([2,3,3,4])
   ...: plt.axis([0,15,0,4])
   ...: plt.title('A Multiseries Horizontal Stacked Bar Chart')
   ...: plt.barh(index,series1,color='w',hatch='xx')
   ...: plt.barh(index,series2,color='w',hatch='///', left=series1)
   ...: plt.barh(index,series3,color='w',hatch='\\\\\\',left=(series1+series2))
   ...: plt.yticks(index+0.4,['Jan15','Feb15','Mar15','Apr15'])

Out[453]:
([<matplotlib.axis.YTick at 0x2a9f0748>,
  <matplotlib.axis.YTick at 0x2a9e1f98>,
  <matplotlib.axis.YTick at 0x2ac06518>,
  <matplotlib.axis.YTick at 0x2ac52128>],
<a list of 4 Text yticklabel objects>)
```

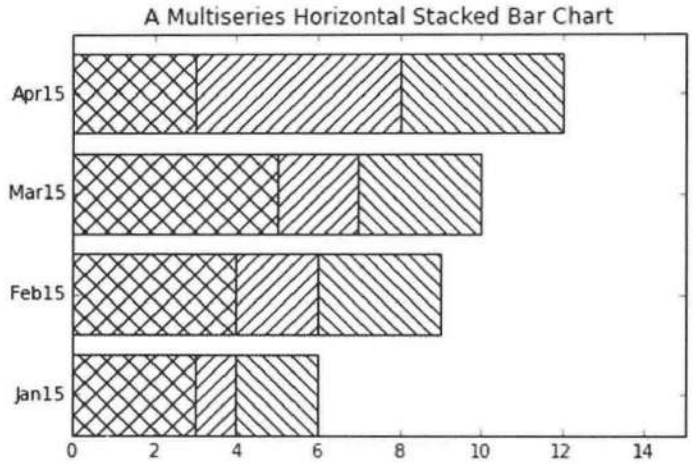


图7-44 堆积条状图可用影线表示不同序列

7.13.5 为 pandas DataFrame 绘制堆积条状图

同样，用plot()函数直接把DataFrame对象中的数据制作成堆积条状图也很简单。只需把stacked关键字参数置为True（见图7-45）。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import pandas as pd
   ...: data = {'series1':[1,3,4,3,5],
                'series2':[2,4,5,2,4],
                'series3':[3,2,3,1,3]}
   ...: df = pd.DataFrame(data)
   ...: df.plot(kind='bar', stacked=True)
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0xcda8f98>
```

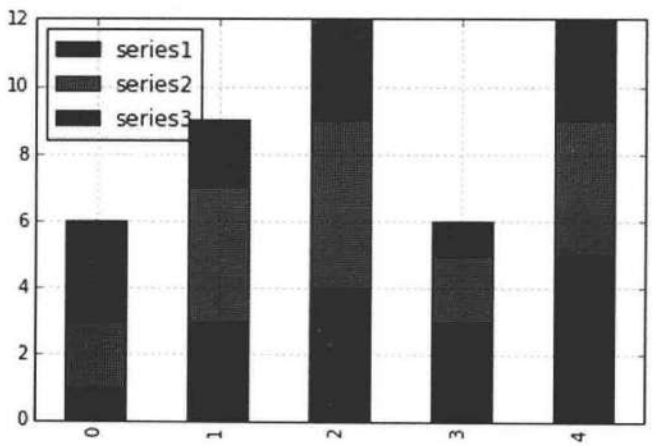


图7-45 DataFrame数据可直接作成堆积条状图

7.13.6　其他条状图

另外一种非常有用的图形表示法是用条状图表现对比关系。两列有着共同类别的数据，其条状图形分列于x轴两侧，沿y轴方向生长。要生成这类图形，需事先对其中一个序列的y值进行取相反数操作，详见下面的例子。从这个例子中，你还会学到如何修改条状图的边框和条状图内部区域的颜色。其实，只要用另外两个关键字参数facecolor和edgecolor设置两种不同的颜色即可。

此外，从这个例子，你还将学到如何在每个长条的末端显示y值标签，这有助于增强条状图的可读性。我们可以使用for循环，在循环体内再借助text()函数显示y值标签。标签的位置可用ha和va关键字参数来调整，它们分别控制着标签在水平和垂直方向上的对齐效果。结果如图7-46所示。

```
In [ ]: import matplotlib.pyplot as plt
   ...: x0 = np.arange(8)
   ...: y1 = np.array([1,3,4,6,4,3,2,1])
   ...: y 2 = np.array([1,2,5,4,3,3,2,1])
   ...: plt.ylim(-7,7)
   ...: plt.bar(x0,y1,0.9,facecolor='r',edgecolor='w')
   ...: plt.bar(x0,-y2,0.9,facecolor='b',edgecolor='w')
   ...: plt.xticks(())
   ...: plt.grid(True)
   ...: for x, y in zip(x0, y1):
           plt.text(x + 0.4, y + 0.05, '%d' % y, ha='center', va= 'bottom')
   ...:
   ...: for x, y in zip(x0, y2):
           plt.text(x + 0.4, -y - 0.05, '%d' % y, ha='center', va= 'top')
```

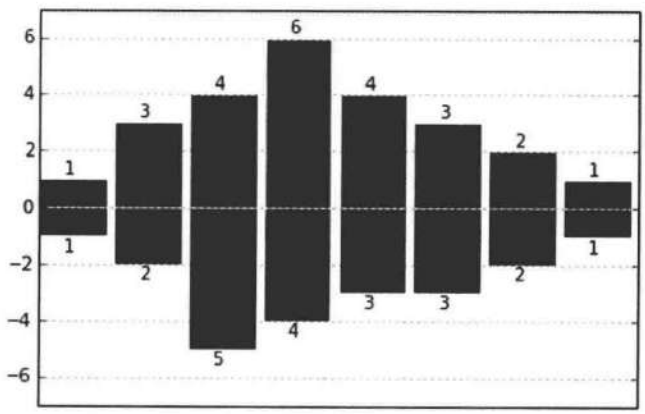


图7-46　两个序列可用这种条状图进行对比

7.14　饼图

除了条状图，饼图也可以用来表示数据。用pie()函数制作饼图很简单。

该函数仍然以要表示的一列数据作为主要参数。这里直接选用百分比（总和为100），实际上你可以选用每种类别的实际数值，而让pie()函数自己去计算每个类别所占的比例。

对于这类图表，仍需用关键字参数设置关键特征。例如，若要定义颜色列表，为作为输入的数据序列分配颜色，可使用colors关键字参数，把颜色列表赋给它。另外一个重要的功能是为饼图的每一小块添加标签，为此需使用labels关键字参数，把标签列表赋给它。

此外，为了绘制标准的圆形饼图，还需要在代码最后调用axis()函数，用字符串'equal'作为参数。最终将得到如图7-47所示的饼图。

```
In [ ]: import matplotlib.pyplot as plt
   ...: labels = ['Nokia','Samsung','Apple','Lumia']
   ...: values = [10,30,45,15]
   ...: colors = ['yellow','green','red','blue']
   ...: plt.pie(values,labels=labels,colors=colors)
   ...: plt.axis('equal')
```

为了增强饼图的表现力，还可以制作从圆饼中抽取出一块的效果。我们在关注某一块时，常会这么做。假如我们这个例子要突出显示Nokia这一块，则需要使用explode关键字参数。它的数据类型为浮点型，取值范围为0~1。1表示这一块完全脱离饼图；0表示没有抽取，也就是饼图仍然是一个完整的圆；而0~1的值则表示这一块未完全脱离饼图（见图7-48）。

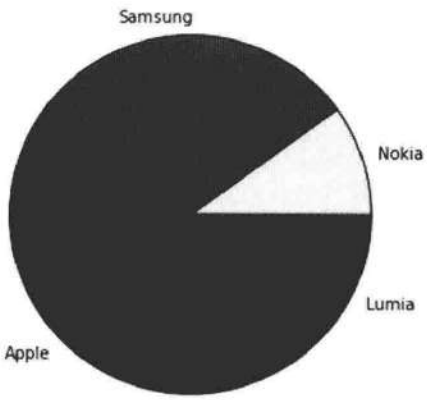


图7-47 一幅非常简单的饼图

同理，你也可以用title()函数为饼图添加标题。还可以用startangle关键字参数调整饼图的旋转角度，该参数接收一个0~360的整数，表示转换的角度（默认值为0）。

修改后的图表如图7-48所示。

```
In [ ]: import matplotlib.pyplot as plt
   ...: labels = ['Nokia','Samsung','Apple','Lumia']
   ...: values = [10,30,45,15]
   ...: colors = ['yellow','green','red','blue']
   ...: explode = [0.3,0,0,0]
   ...: plt.title('A Pie Chart')
   ...: plt.pie(values,labels=labels,colors=colors,explode=explode,startangle=180)
   ...: plt.axis('equal')
```

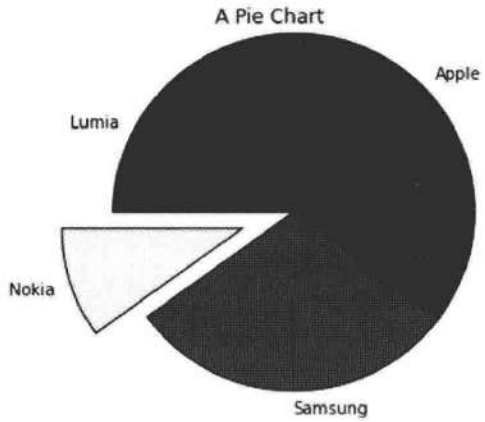


图7-48　更高级的饼图

你能往饼图中添加的元素可远不止这些。例如，由于饼图没有带刻度的轴，因此无法直观地了解每一块所表示的百分比大小。为了弥补这一不足，可以用autopct关键字参数，在每一块的中间位置添加文本标签来显示百分比。

如果你想让图表更具吸引力，还可以用shadow关键字参数添加阴影效果，将其置为True即可。最终将得到如图7-49所示的饼图。

```
In [ ]: import matplotlib.pyplot as plt
   ...: labels = ['Nokia','Samsung','Apple','Lumia']
   ...: values = [10,30,45,15]
   ...: colors = ['yellow','green','red','blue']
   ...: explode = [0.3,0,0,0]
   ...: plt.title('A Pie Chart')
   ...: plt.pie(values,labels=labels,colors=colors,explode=explode,
        shadow=True,autopct='%1.1f%%',startangle=180)
   ...: plt.axis('equal')
```

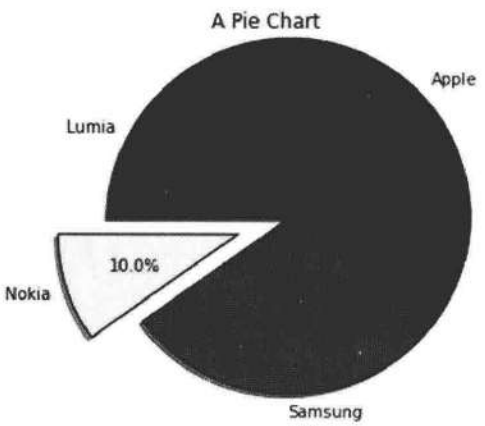


图7-49　比之前还要高级的饼图

为 DataFrame 绘制饼图

我们也可以用饼图表示DataFrame对象中的数据。但是，每幅饼图只能表示一个序列，因此在下面这个例子中，我们只将序列df['series1']作成图。我们需要使用plot()的kind关键字参数指定图表类型为pie。此外，要绘制一个标准的圆形饼图，就有必要添加figsize关键字参数。最终将得到如图7-50所示的饼图。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import pandas as pd
   ...: data = {'series1':[1,3,4,3,5],
                'series2':[2,4,5,2,4],
                'series3':[3,2,3,1,3]}
   ...: df = pd.DataFrame(data)
   ...: df['series1'].plot(kind='pie',figsize=(6,6))
Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0xe1ba710>
```

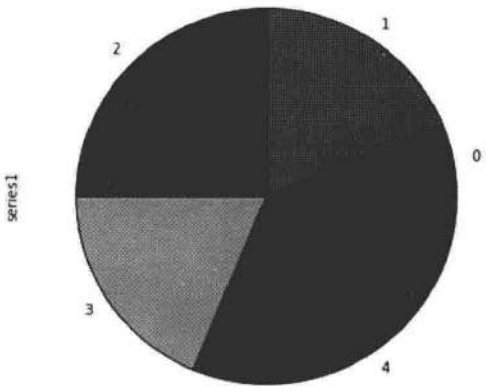


图7-50 pandas DataFrame中的数据可直接做成饼图

7.15 高级图表

除去条状图、饼图等较为传统的图表，我们还可能需要用到其他形式的图表。网上以及各种出版物中有很多关于其他类型图形解决方案的讨论和建议，其中有些非常出色，也很吸引人。但限于篇幅，这一节只介绍其中一些图形表示法，更为详细的讨论则超出了本书的范围。数据可视化世界的疆域在不断扩展之中，你可以把这一节作为它的入门材料。

7.15.1 等值线图

等值线图或等高线图在科学界很常用。这种可视化方法用由一圈圈封闭的曲线组成的等值线图表示三维结构的表面，其中封闭的曲线表示的是一个个处于同一层级或z值相同的数据点。

虽然等值线图看上去结构很复杂，其实用matplotlib实现起来并不难。首先，你需要用$z=f(x,y)$

函数生成三维结构。然后，定义x、y的取值范围，确定要显示的区域。之后使用$f(x,y)$函数计算每一对(x,y)所对应的z值，得到一个z值矩阵。最后，用contour()函数生成三维结构表面的等值线图。定义颜色表，为等值线图添加不同颜色，效果往往会更好；也就是说，用渐变色填充由等值线划分成的区域。如图7-51所示，用逐渐加深的蓝色阴影表示负值，而随着数值的增大，则逐渐改用黄色甚至红色。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: dx = 0.01; dy = 0.01
   ...: x = np.arange(-2.0,2.0,dx)
   ...: y = np.arange(-2.0,2.0,dy)
   ...: X,Y = np.meshgrid(x,y)
   ...: def f(x,y):
            return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
   ...: C = plt.contour(X,Y,f(X,Y),8,colors='black')
   ...: plt.contourf(X,Y,f(X,Y),8)
   ...: plt.clabel(C, inline=1, fontsize=10)
```

标准的渐变色组合（颜色表）如图7-51所示。在实际应用中，要从多种颜色中选定你需要的颜色，把它赋给cmap关键字参数。

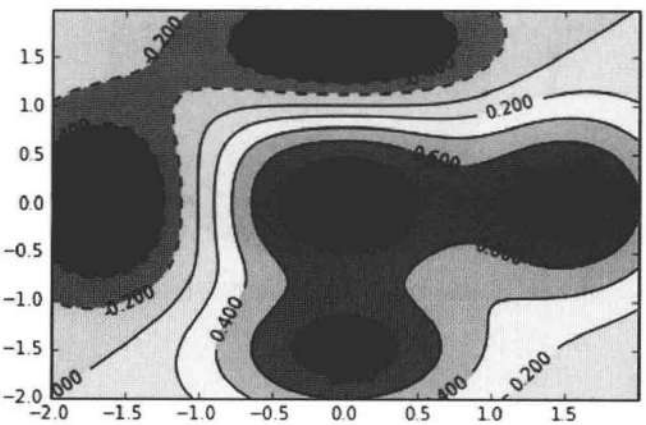


图7-51　等值线图可以表示一个表面的z值信息

此外，如若使用等值线图，在该图的一侧增加图例作为对图表中所用颜色的说明几乎是必需的。在代码的最后增加colorbar()函数即可实现该功能。图7-52所示的图表使用了另外一种颜色表，先是由黑色过渡到红色，再过渡到黄色，最后最大值使用白色。这种彩图中，cmap参数的值为plt.cm.hot。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: dx = 0.01; dy = 0.01
   ...: x = np.arange(-2.0,2.0,dx)
   ...: y = np.arange(-2.0,2.0,dy)
   ...: X,Y = np.meshgrid(x,y)
```

```
...:
...: C = plt.contour(X,Y,f(X,Y),8,colors='black')
...: plt.contourf(X,Y,f(X,Y),8,cmap=plt.cm.hot)
...: plt.clabel(C, inline=1, fontsize=10)
...: plt.colorbar()
```

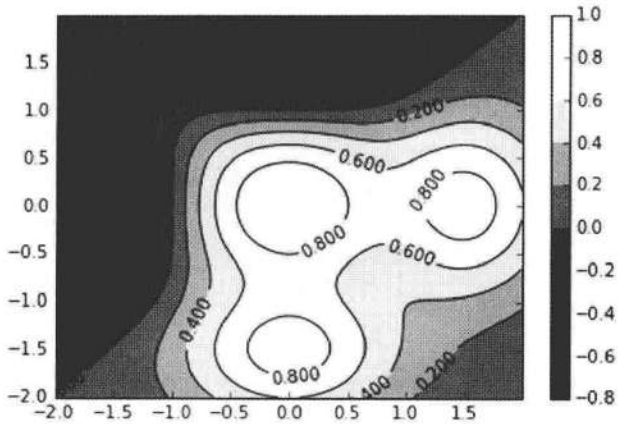


图7-52 用表示“热度”的颜色表可增强等值线图的吸引力

7.15.2 极区图

另一种取得了一定成功的高级图表是极区图。这种图表由一系列呈放射状延伸的区域组成，其中每块区域占据一定的角度。因此若要用极区图表示两个不同的数值，分别指定它们在极区图中所占的分量：每块区域的半径r和它所占的角度，其实这就是极坐标(r,θ)，是在坐标轴系中表示数据的另一种方法。从图表的角度来看，你可以将其视作兼有饼图和条状图特点的图表。之所以说它像饼图，是因为每个区域的角度所表示的是其所属类别占全部类别的比例。至于说它像条状图，是因为半径的长度表示某一类别的数值大小。

到目前为止，我们一直使用标准颜色集，每种颜色用单一字符颜色编码来表示（例如，r代表红色）。事实上，你可以自定义任意的颜色列表，方法是指定颜色列表，其中每个元素为字符串类型的RGB编码，其格式为#rrggbb。

奇怪的是，制作极区图需要使用bar()函数，把角度θ列表和半径列表传递给它。你将得到如图7-53所示的极图。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: N = 8
   ...: theta = np.arange(0.,2 * np.pi, 2 * np.pi / N)
   ...: radii = np.array([4,7,5,3,1,5,6,7])
   ...: plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)
   ...: colors = np.array(['#4bb2c5', '#c5b47f', '#EAA228', '#579575', '#839557', '#958c12',
        '#953579', '#4b5de4'])
   ...: bars = plt.bar(theta, radii, width=(2*np.pi/N), bottom=0.0, color=colors)
```

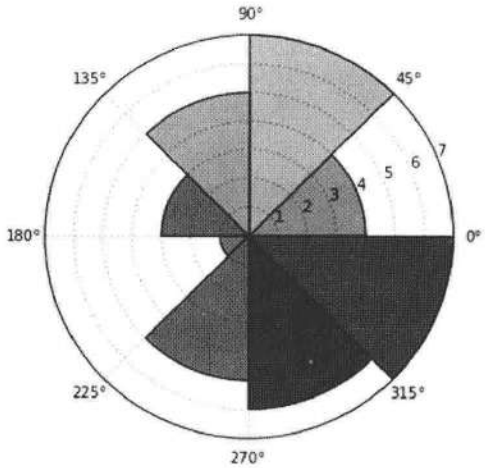


图7-53　极区图

这个例子中，我们定义了一列#rrggbb格式的颜色值，其实还可以用颜色的实际名称来表示颜色（见图7-54）。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: N = 8
   ...: theta = np.arange(0.,2 * np.pi, 2 * np.pi / N)
   ...: radii = np.array([4,7,5,3,1,5,6,7])
   ...: plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)
   ...: colors = np.array(['lightgreen', 'darkred', 'navy', 'brown', 'violet', 'plum',
        'yellow', 'darkgreen'])
   ...: bars = plt.bar(theta, radii, width=(2*np.pi/N), bottom=0.0, color=colors)
```

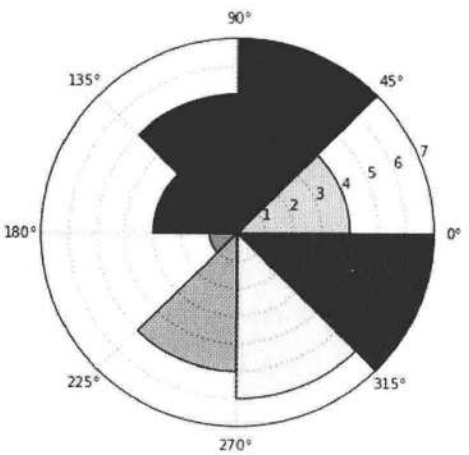


图7-54　使用另外一种颜色列表得到的极区图

7.16 mplot3d

mplot3d工具集是matplotlib内置的标配，可用来实现3D数据可视化功能。如果生成的图形在单独的窗口中显示，你还可以用鼠标旋转三维图形的轴进行查看。

mplot3d仍然使用Figure对象，只不过Axes对象要替换为该工具集的Axes3D对象。因此，使用Axes3D对象前，需先将其导入进来。

```
from mpl_toolkits.mplot3d import Axes3D
```

7.16.1 3D 曲面

在等值线图那一节，我们用等值线来表示三维曲面。而用mplot3d，可以将表面直接绘制成3D形状。下面例子中，我们将再次使用绘制等值线图所用到的$z=f(x,y)$函数。

计算出分割线坐标后，就可以用plot_surface()函数绘制曲面。蓝色三维曲面图表请见图7-55。

```
In [ ]: from mpl_toolkits.mplot3d import Axes3D
   ...: import matplotlib.pyplot as plt
   ...: fig = plt.figure()
   ...: ax = Axes3D(fig)
   ...: X = np.arange(-2,2,0.1)
   ...: Y = np.arange(-2,2,0.1)
   ...: X,Y = np.meshgrid(X,Y)
   ...: def f(x,y):
   ...:     return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
   ...: ax.plot_surface(X,Y,f(X,Y), rstride=1, cstride=1)
```

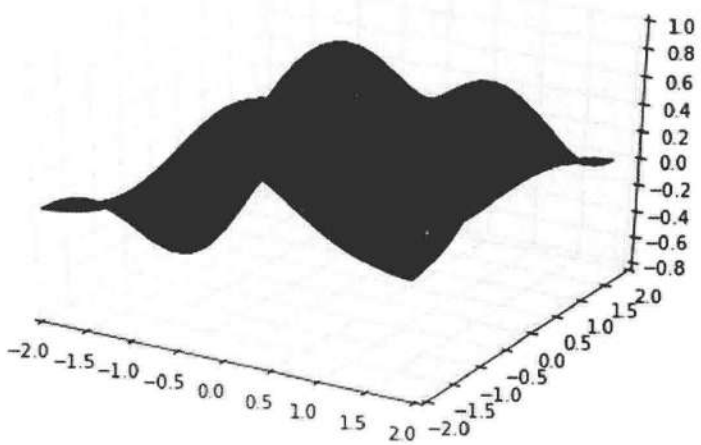


图7-55 用mplot3d工具集可以绘制3D曲面

修改颜色表，3D表面效果会更加突出，例如，可以用cmap关键字参数指定各颜色。还可以用view_init()函数旋转曲面，修改elev和azim两个关键字参数，从不同的视角查看曲面，其中第一个关键字参数指定从哪个高度查看曲面，第二个参数指定曲面旋转的角度。

例如，你可以使用plt.cm.hot颜色表，把视角设置为elev=30和azim=125，结果如图7-56所示。

```
In [ ]: from mpl_toolkits.mplot3d import Axes3D
   ...: import matplotlib.pyplot as plt
   ...: fig = plt.figure()
   ...: ax = Axes3D(fig)
   ...: X = np.arange(-2,2,0.1)
   ...: Y = np.arange(-2,2,0.1)
   ...: X,Y = np.meshgrid(X,Y)
   ...: def f(x,y):
            return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
   ...: ax.plot_surface(X,Y,f(X,Y), rstride=1, cstride=1, cmap=plt.cm.hot)
   ...: ax.view_init(elev=30,azim=125)
```

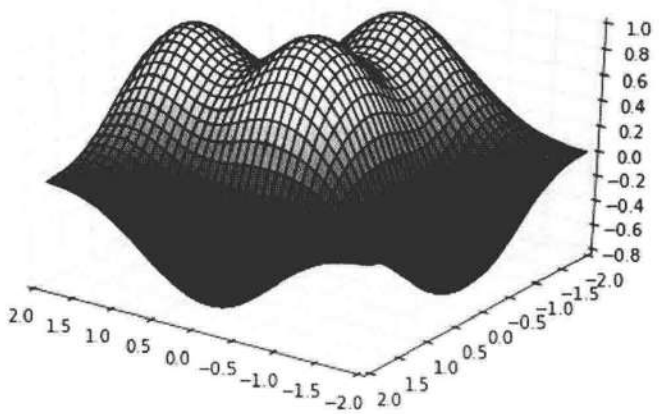


图7-56　旋转曲面，调整高度，从更高的视角查看3D曲面

7.16.2　3D 散点图

在所有3D图形中，散点图最常用。通过这种可视化方法，能够识别数据点的分布是否遵循某种特定趋势，尤其是可以识别它们是否有聚集成簇的趋势。

下面这个例子中，我们仍旧使用scatter()函数，使用方法跟绘制2D图形相同，但是要将其应用于Axes3D对象。这样做，你可以多次调用scatter()函数，在同一个3D对象中显示不同的序列（见图7-57）。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: from mpl_toolkits.mplot3d import Axes3D
```

```
...: xs = np.random.randint(30,40,100)
...: ys = np.random.randint(20,30,100)
...: zs = np.random.randint(10,20,100)
...: xs2 = np.random.randint(50,60,100)
...: ys2 = np.random.randint(30,40,100)
...: zs2 = np.random.randint(50,70,100)
...: xs3 = np.random.randint(10,30,100)
...: ys3 = np.random.randint(40,50,100)
...: zs3 = np.random.randint(40,50,100)
...: fig = plt.figure()
...: ax = Axes3D(fig)
...: ax.scatter(xs,ys,zs)
...: ax.scatter(xs2,ys2,zs2,c='r',marker='^')
...: ax.scatter(xs3,ys3,zs3,c='g',marker='*')
...: ax.set_xlabel('X Label')
...: ax.set_ylabel('Y Label')
...: ax.set_zlabel('Z Label')
Out[34]: <matplotlib.text.Text at 0xe1c2438>
```

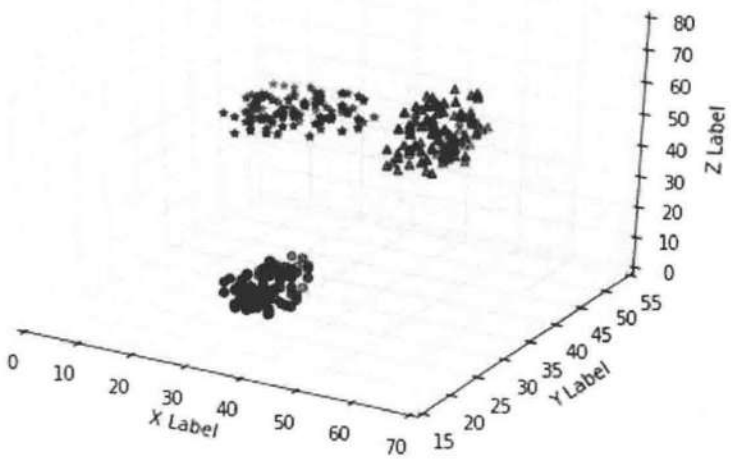


图7-57 包含三个簇的3D散点图

7.16.3 3D 条状图

数据分析常用的另一种3D图形是3D条状图。要绘制这种图表，同样是将bar()函数应用于Axes3D对象。如果定义了多个序列，可以在同一个3D对象上多次调用bar()函数（见图7-58）。

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: from mpl_toolkits.mplot3d import Axes3D
...: x = np.arange(8)
...: y = np.random.randint(0,10,8)
```

```
...: y2 = y + np.random.randint(0,3,8)
...: y3 = y2 + np.random.randint(0,3,8)
...: y4 = y3 + np.random.randint(0,3,8)
...: y5 = y4 + np.random.randint(0,3,8)
...: clr = ['#4bb2c5', '#c5b47f', '#EAA228', '#579575', '#839557', '#958c12', '#953579',
'#4b5de4']
...: fig = plt.figure()
...: ax = Axes3D(fig)
...: ax.bar(x,y,0,zdir='y',color=clr)
...: ax.bar(x,y2,10,zdir='y',color=clr)
...: ax.bar(x,y3,20,zdir='y',color=clr)
...: ax.bar(x,y4,30,zdir='y',color=clr)
...: ax.bar(x,y5,40,zdir='y',color=clr)
...: ax.set_xlabel('X Axis')
...: ax.set_ylabel('Y Axis')
...: ax.set_zlabel('Z Axis')
...: ax.view_init(elev=40)
```

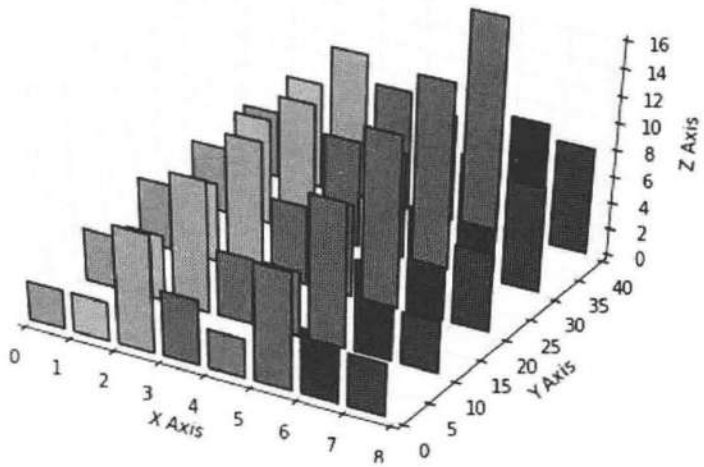


图7-58　3D条状图

7.17　多面板图形

至此，你已见过多种用图表表示数据的方法，也见过把一幅图分成多幅子图的情况。这一节将进一步加深你对这个主题的理解，我们来分析几种更为复杂的情况。

7.17.1　在其他子图中显示子图

我们现在来介绍一种更为高级的方法：把图表放入框架，在其他图表中显示。既然我们在讲

框架，也就是Axes对象，你需要把主Axes对象（也就是说主图表）跟放置另一个Axes对象实例的框架分开。用figure()函数取到Figure对象，用add_axes()函数在它上面定义两个Axes对象。结果请见图7-59。

```
In [ ]: import matplotlib.pyplot as plt
   ...: fig = plt.figure()
   ...: ax = fig.add_axes([0.1,0.1,0.8,0.8])
   ...: inner_ax = fig.add_axes([0.6,0.6,0.25,0.25])
```

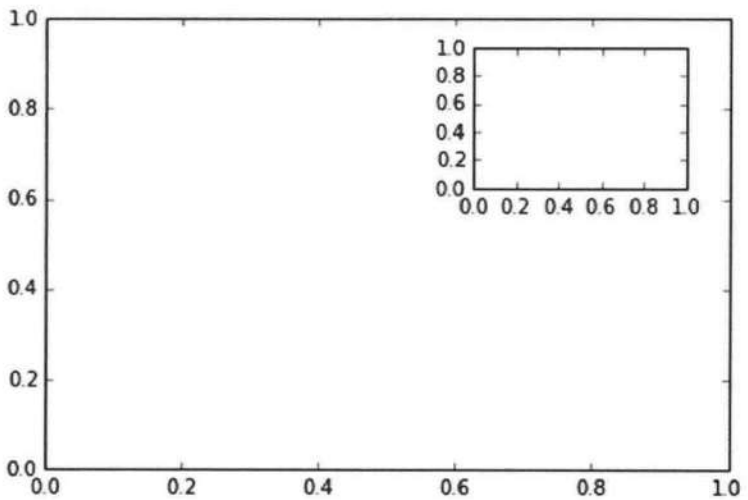


图7-59 带子图的图表

为了更好地理解这种图形模式，你可以为Axes对象的plot()方法传入两列数据，所实现的效果请见图7-60。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: fig = plt.figure()
   ...: ax = fig.add_axes([0.1,0.1,0.8,0.8])
   ...: inner_ax = fig.add_axes([0.6,0.6,0.25,0.25])
   ...: x1 = np.arange(10)
   ...: y1 = np.array([1,2,7,1,5,2,4,2,3,1])
   ...: x2 = np.arange(10)
   ...: y2 = np.array([1,3,4,5,4,5,2,6,4,3])
   ...: ax.plot(x1,y1)
   ...: inner_ax.plot(x2,y2)
Out[95]: [<matplotlib.lines.Line2D at 0x14acf6d8>]
```

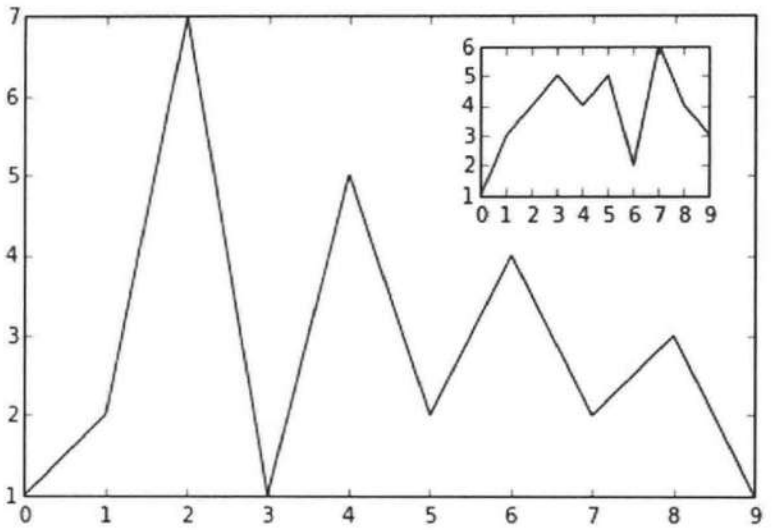


图7-60　更加真实的带子图的图表

7.17.2　子图网格

前面已讲过如何生成子图。而要把图形分成多个区域，添加多个子图，可以用subplots()函数，方法也很简单。matplotlib的GridSpec()函数可用来管理更为复杂的情况。它把绘图区域分成多个子区域，你可以把一个或多个子区域分配给每一幅子图，因此可以得到如图7-61所示的图表，其中每幅子图的大小、方位各不相同。

```
In [ ]: import matplotlib.pyplot as plt
   ...: gs = plt.GridSpec(3,3)
   ...: fig = plt.figure(figsize=(6,6))
   ...: fig.add_subplot(gs[1,:2])
   ...: fig.add_subplot(gs[0,:2])
   ...: fig.add_subplot(gs[2,0])
   ...: fig.add_subplot(gs[:2,2])
   ...: fig.add_subplot(gs[2,1:])
Out[97]: <matplotlib.axes._subplots.AxesSubplot at 0x12717438>
```

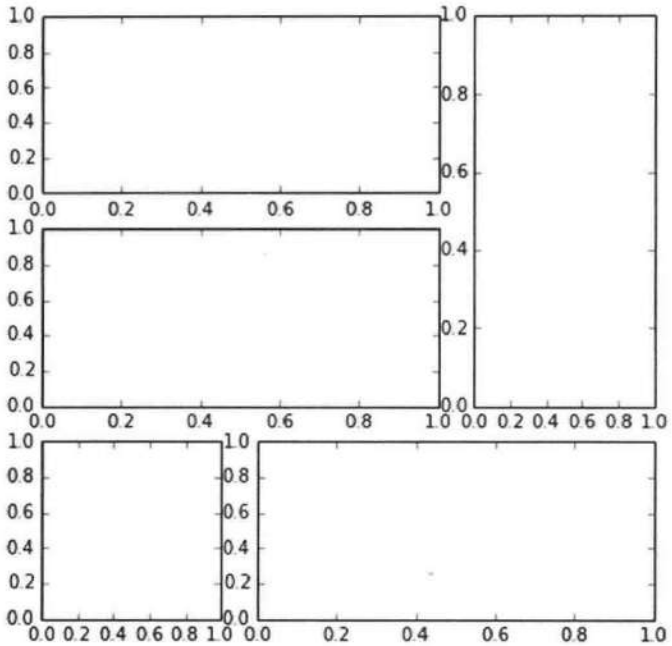


图7-61 在由子区域组成的网格中，可以绘制大小不同的子图

既然你已知道如何把不同的区域分配给子图，我们来看一下这些子图的使用方法。事实上，你可以在add_subplot()函数返回的Axes对象上调用plot()函数，绘制相应的图形（见图7-62）。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: gs = plt.GridSpec(3,3)
   ...: fig = plt.figure(figsize=(6,6))
   ...: x1 = np.array([1,3,2,5])
   ...: y1 = np.array([4,3,7,2])
   ...: x2 = np.arange(5)
   ...: y2 = np.array([3,2,4,6,4])
   ...: s1 = fig.add_subplot(gs[1,:2])
   ...: s1.plot(x,y,'r')
   ...: s2 = fig.add_subplot(gs[0,:2])

   ...: s2.bar(x2,y2)
   ...: s3 = fig.add_subplot(gs[2,0])
   ...: s3.barh(x2,y2,color='g')
   ...: s4 = fig.add_subplot(gs[:2,2])
   ...: s4.plot(x2,y2,'k')
   ...: s5 = fig.add_subplot(gs[2,1:])
   ...: s5.plot(x1,y1,'b^',x2,y2,'yo')
```

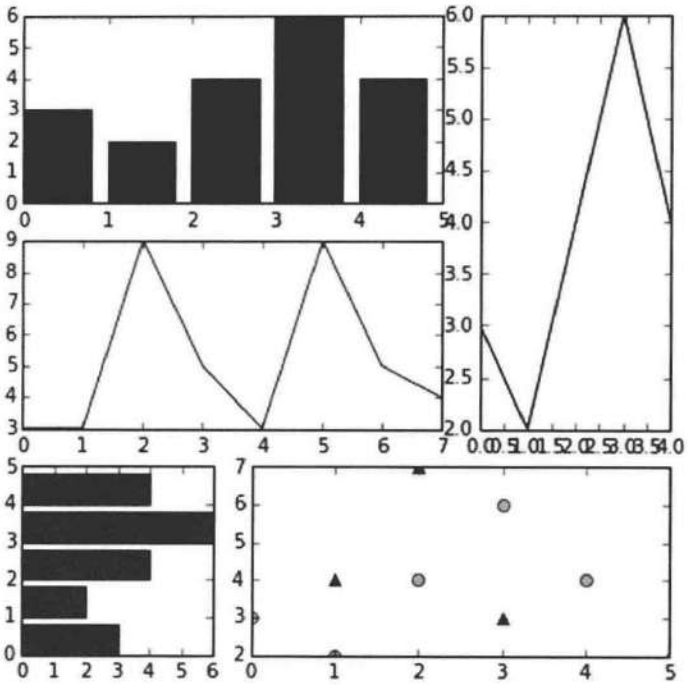


图7-62　子图网格可同时显示多个图形

7.18　小结

从本章你学到了matplotlib库的所有基础功能。通过对一系列例子的学习，你已掌握了数据可视化所需的基础工具。你熟悉了用几行代码就能生成各种类型图表的方法。

随着本章的结束，我们也介绍完了为数据分析工作提供基础工具的所有库。下一章将学习跟数据分析联系更为紧密的内容。

第8章

用scikit-learn库实现机器学习

数据分析由一连串步骤组成，对于其中预测模型的创建和验证这一步，我们用scikit-learn这个功能强大的库来完成。本章通过几个例子介绍了几种不同的预测模型的创建方法。

8.1 scikit-learn 库

Python库scikit-learn整合了多种机器学习算法。2007年，Cournapeu开始开发这个库，但直到2010年才发布它的第一个版本。

这个库是SciPy（Scientific Python，Python科学计算）工具集的一部分，该工具集包含多个为科学计算尤其是数据分析而开发的库，其中不少库都在本书讨论范围之列。通常这些库被称作SciKits，库名scikit-learn的前半部分正是来源于此，而后半部分则是来自该库所面向的应用领域——机器学习——的英文名“Machine Learning”。

8.2 机器学习

机器学习这门学科研究的是识别作为数据分析对象的数据集中模式的方法，尤其指研发算法，从数据中学习，并作出预测。所有的机器学习方法都是以建立特定的模型为基础。

要建立能够学习的机器，方法有多种，但各有各的特点，选用哪种方法取决于数据的特点和预测模型的类型。选用哪种方法这个问题被称作学习问题。

在学习阶段，遵从某种模式的数据可以是数组形式，其中每个元素只包含单个值或多个值。这些值常被称作特征（feature）或属性（attribute）。

8.2.1 有监督和无监督学习

根据数据和所要创建的模型的类型，学习问题可以分为两大类。

有监督学习。训练集包含作为预测结果（目标值）的额外的属性信息。这些信息可以指导模型对新数据（测试集）作出跟已有数据类似的预测结果。

- 分类：训练集数据属于两种或以上类别；已标注的数据可指导系统学习能够识别每个类别的特征。预测系统未见过的新数据时，系统将根据新数据的特征，评估它的类别。
- 回归：被预测结果为连续型变量。最易于理解的应用场景是，在散点图中找出能够描述一系列数据点趋势的直线。

无监督学习。训练集数据由一系列输入值x组成，其目标值未知。

- 聚类：发现数据集中由相似的个体组成的群组。
- 降维：将高维数据集的维数减少到两维或三维，这样不仅便于数据可视化，而且大幅降低维度后，每一维所传达的信息还会更多。

除了上述两大类别的方法之外，还有一类方法，它们以验证、评估模型为目的。

8.2.2　训练集和测试集

机器学习方法使得我们可以用数据集创建模型，识别模型的特性（property）之后，再用来处理新数据。在机器学习过程中，经常要评估算法的好坏。评估算法需要把算法分为训练集和测试集两部分，从前者学习数据的特性，再用后者测试得到的特性。

8.3　用 scikit-learn 实现有监督学习

这一章，我们将讲解以下几个有监督学习的例子。

- 用Iris数据集讲解分类
  - K-近邻分类器
  - 支持向量分类（SVC）
- 用Diabetes数据集介绍回归算法
  - 线性回归
  - 支持向量回归（SVR）

有监督学习方法从数据集读取数据，学习两个或以上特征之间可能的模式；因为训练集结果（目标或标签）已知，所以学习是可行的。scikit-learn的所有模型都被称作有监督估计器，训练估计器要用到`fit(x,y)`函数：其中x指观察到的特征，y指的是目标。估计器经过训练后，就能预测任何标签未知的新数据x的y值；预测是由`predict(x)`函数完成的。

8.4　Iris 数据集

Iris数据集（鸢尾花卉数据集）很特别。早在1936年，Sir Ronald Fisher就第一次将它用于数据挖掘实验。因为这些数据是安德森通过直接测量鸢尾花卉花朵的各个部分得到的，所以为了纪念他，该数据集也常被称作安德森鸢尾花卉数据集。该数据集的数据采自三种不同的鸢尾花卉（山鸢尾、变色鸢尾和维吉尼亚鸢尾），确切来说，这些数据表示的是萼片和花瓣的长宽（见图8-1）。

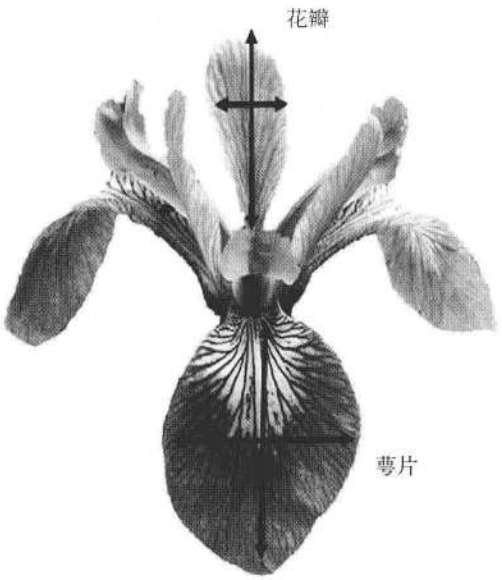


图8-1 变色鸢尾花瓣和萼片的长与宽

目前，人们常用该数据集讲解多种分析方法，尤其是可以用机器学习方法解决的分类问题。由此可见，scikit-learn库以150x4 NumPy数组形式内置了该数据集也并非出于偶然。

现在，我们来详细探究这个数据集。在IPython QtConsole或Python会话中导入它。

```
In [ ]: from sklearn import datasets
   ...: iris = datasets.load_iris()
```

这样，我们把鸢尾花卉数据集的所有数据和元数据都加载到iris变量中。因此使用iris变量的data属性，就能查看它所包含的数据。

```
In [ ]: iris.data
Out[ ]:
array([[ 5.1, 3.5, 1.4, 0.2],
       [ 4.9, 3. , 1.4, 0.2],
       [ 4.7, 3.2, 1.3, 0.2],
       [ 4.6, 3.1, 1.5, 0.2],
       ...
```

如你所见，输出结果为一个包含150个元素的数组，每个元素又包含四个数值：分别为萼片和花瓣的数据。

要知道每一种花卉的种类，访问iris的target属性即可。

```
In [ ]: iris.target
Out[ ]:
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

```
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

输出结果包含150个数值，其中共有三种可能的取值（0、1和2），分别代表三种不同的鸢尾花卉。访问iris的target_names属性，可以了解每个值所代表的花卉类别。

```
In [ ]: iris.target_names
Out[ ]:
array(['setosa', 'versicolor', 'virginica'],
      dtype='|S10')
```

为了更好地理解这个数据集，你可以使用matplotlib库。通过从第7章学到的技巧，用三种颜色表示三种花卉种类，绘制一幅散点图。*x*轴表示萼片的长度，*y*轴表示萼片的宽度。

```
In [ ]: import matplotlib.pyplot as plt
  ...: import matplotlib.patches as mpatches
  ...: from sklearn import datasets
  ...:
  ...: iris = datasets.load_iris()
  ...: x = iris.data[:,0] #X-Axis - sepal length
  ...: y = iris.data[:,1] #Y-Axis - sepal length
  ...: species = iris.target    #Species
  ...:
  ...: x_min, x_max = x.min() - .5,x.max() + .5
  ...: y_min, y_max = y.min() - .5,y.max() + .5
  ...:
  ...: #SCATTERPLOT
  ...: plt.figure()
  ...: plt.title('Iris Dataset - Classification By Sepal Sizes')
  ...: plt.scatter(x,y, c=species)
  ...: plt.xlabel('Sepal length')
  ...: plt.ylabel('Sepal width')
  ...: plt.xlim(x_min, x_max)
  ...: plt.ylim(y_min, y_max)
  ...: plt.xticks(())
  ...: plt.yticks(())
```

上述代码将生成如图8-2所示的散点图。蓝、绿和红分别代表山鸢尾、变色鸢尾和维吉尼亚鸢尾。

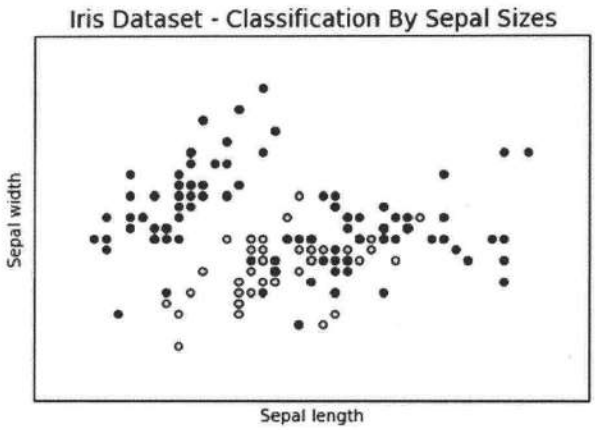


图8-2 用不同颜色表示的不同鸢尾花卉种类

由图8-2可见，山鸢尾跟另外两种花卉不同，表示山鸢尾的蓝色数据点形成一簇，与其他点区分开来。

仿照上述步骤，改用花瓣的长和宽这两个变量绘制图表。上述代码只需改动几处，就可以继续使用。

```
In [ ]: import matplotlib.pyplot as plt
   ...: import matplotlib.patches as mpatches
   ...: from sklearn import datasets
   ...:
   ...: iris = datasets.load_iris()
   ...: x = iris.data[:,2] #X-Axis - petal length
   ...: y = iris.data[:,3] #Y-Axis - petal length
   ...: species = iris.target #Species
   ...:
   ...: x_min, x_max = x.min() - .5,x.max() + .5
   ...: y_min, y_max = y.min() - .5,y.max() + .5
   ...: #SCATTERPLOT
   ...: plt.figure()
   ...: plt.title('Iris Dataset - Classification By Petal Sizes', size=14)
   ...: plt.scatter(x,y, c=species)
   ...: plt.xlabel('Petal length')
   ...: plt.ylabel('Petal width')
   ...: plt.xlim(x_min, x_max)
   ...: plt.ylim(y_min, y_max)
   ...: plt.xticks(())
   ...: plt.yticks(())
```

上述代码得到的散点图如图8-3所示。用花瓣的长和宽作为特征时，三种类别之间的区别更明显。由图可见，这次得到了三个不同的簇。

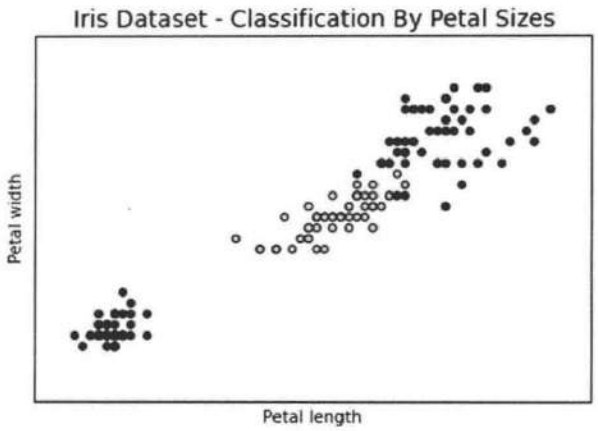


图8-3　用不同颜色表示的不同鸢尾花卉种类

主成分分解

上面，我们尝试用花瓣、萼片的四个测量数据来描述三种花卉的特点。两幅散点图是分别用萼片和花瓣的测量数据制作的，那么怎样才能把四个测量数据整合到一起呢？即使是3D散点图也无法整合四个维度。

关于这个问题，早已有现成的解决方法——*主成分分析法*（Principal Component Analysis，PCA）。该方法可以减少系统的维数，保留足以描述各数据点特征的信息，其中新生成的各维叫作*主成分*。对于上述这个例子，把四维减少到三维后，就能把得到的结果绘制为3D散点图。这样，我们就可以使用萼片和花瓣的测量数据来描述数据集中各种鸢尾花卉的特点。

scikit-learn库的`fit_transform()`函数就是用来降维的，属于PCA对象。使用前，要先导入PCA模块sklearn.decomposition，然后使用`PCA()`构造函数，用`n_components`选项指定要降到几维（主成分）。我们这里为三维。最后，调用`fit_transform()`函数，传入四维的Iris数据集作为参数。

```
from sklearn.decomposition import PCA
x_reduced = PCA(n_components=3).fit_transform(iris.data)
```

此外，绘制3D散点图要用到matplotlib的mpl_toolkits.mplot3d模块。如果不记得怎么用，请见7.16.2节。

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
from sklearn.decomposition import PCA

iris = datasets.load_iris()
x = iris.data[:,1] #X-Axis - petal length
y = iris.data[:,2] #Y-Axis - petal length
species = iris.target    #Species
x_reduced = PCA(n_components=3).fit_transform(iris.data)
```

```
#SCATTERPLOT 3D
fig = plt.figure()
ax = Axes3D(fig)
ax.set_title('Iris Dataset by PCA', size=14)
ax.scatter(x_reduced[:,0],x_reduced[:,1],x_reduced[:,2], c=species)
ax.set_xlabel('First eigenvector')
ax.set_ylabel('Second eigenvector')
ax.set_zlabel('Third eigenvector')
ax.w_xaxis.set_ticklabels(())
ax.w_yaxis.set_ticklabels(())
ax.w_zaxis.set_ticklabels(())
```

上述代码将生成如图8-4所示的散点图。三种鸢尾花卉被恰当地表示出来，各自形成一簇。

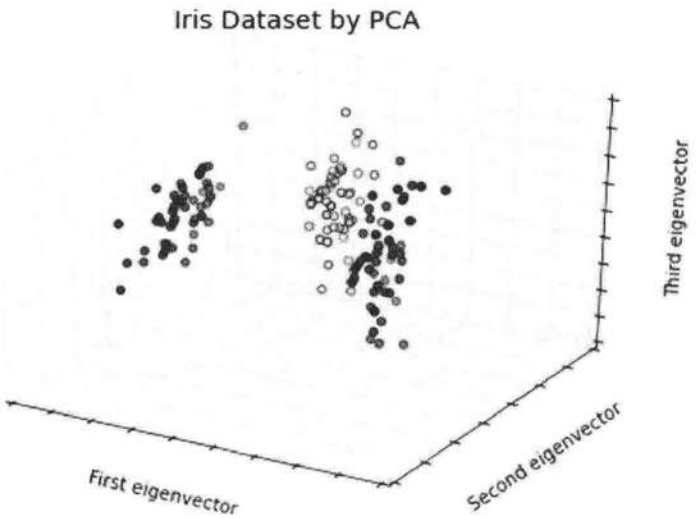


图8-4 包含三簇的3D散点图，其中每簇代表一种鸢尾花卉

8.5 K-近邻分类器

现在，请你来处理一个分类任务。需要用到scikit-learn库的分类器对象。

分类器要完成的任务是，给定一种鸢尾花卉的测量数据，为这种花卉分类。最简单的分类器是近邻分类器。近邻算法搜索训练集，寻找与用作测试的新个体最相似的观测记录。

讲到这里，弄清楚训练集和测试集这两个概念很重要（第1章提到过这两者）。如果确实只有一个数据集，这也没关系，重要的是不要使用相同的数据进行训练和测试。鉴于此，把数据集分成两份：一份专门用于训练算法，另一份用于验证算法。

因此在讲解后面的内容之前，先把Iris数据集分为两部分。最好是先打乱数组各元素的顺序，然后再切分，因为数据往往是按特定顺序采集来的，比如Iris数据集就是按照种类进行排序的。

我们用NumPy的random.permutation()函数打乱数据集的所有元素。打乱后的数据集依旧包含150条不同的观测数据，其中前140条用作训练集，剩余10条用作测试集。

```
import numpy as np
from sklearn import datasets
np.random.seed(0)
iris = datasets.load_iris()
x = iris.data
y = iris.target
i = np.random.permutation(len(iris.data))
x_train = x[i[:-10]]
y_train = y[i[:-10]]
x_test = x[i[-10:]]
y_test = y[i[-10:]]
```

现在，你就可以使用K-近邻算法。导入KneighborsClassifier，调用分类器的构造函数，然后用fit()函数对其进行训练。

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(x_train,y_train)
Out[86]:
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_neighbors=5, p=2, weights='uniform')
```

我们用140条观测数据训练knn分类器，得到了预测模型。我们随后将验证它的效果。分类器应该能够正确预测测试集中10条观测数据所对应的类别。要获取预测结果，可直接在预测模型knn上调用predict()函数。最后，将预测结果与y_test中的实际值进行比较。

```
knn.predict(x_test)
Out[100]: array([1, 2, 1, 0, 0, 0, 2, 1, 2, 0])
y_test
Out[101]: array([1, 1, 1, 0, 0, 0, 2, 1, 2, 0])
```

由上可知，错误率为10%。我们可以在用萼片测量数据绘制的2D散点图中，画出决策边界（decision boundary）。

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
iris = datasets.load_iris()
x = iris.data[:,:2]      #X-Axis - sepal length-width
y = iris.target          #Y-Axis - species

x_min, x_max = x[:,0].min() - .5,x[:,0].max() + .5
y_min, y_max = x[:,1].min() - .5,x[:,1].max() + .5

#MESH
cmap_light = ListedColormap(['#AAAAFF','#AAFFAA','#FFAAAA'])
h = .02
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
```

```
knn = KNeighborsClassifier()
knn.fit(x,y)
Z = knn.predict(np.c_[xx.ravel(),yy.ravel()])
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx,yy,Z,cmap=cmap_light)

#Plot the training points
plt.scatter(x[:,0],x[:,1],c=y)
plt.xlim(xx.min(),xx.max())
plt.ylim(yy.min(),yy.max())
```

```
Out[120]: (1.5, 4.900000000000003)
```

如图8-5所示，散点图中，有小块区域伸入到其他决策边界之中。

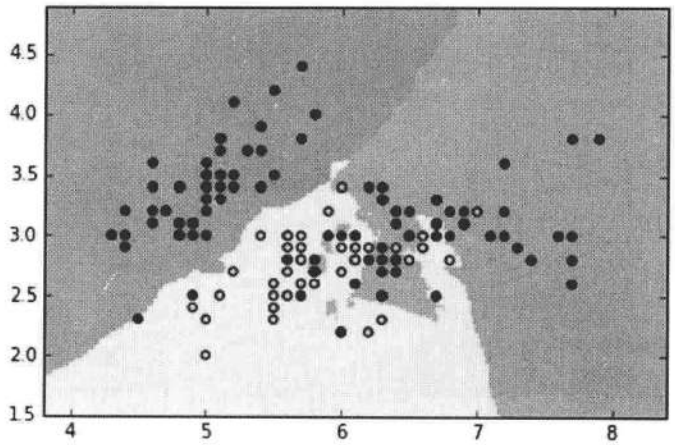


图8-5 用三种不同颜色表示的三个决策边界

在用花瓣数据绘制的散点图中，也可以画出决策边界。

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
iris = datasets.load_iris()
x = iris.data[:,2:4]      #X-Axis - petals length-width
y = iris.target           #Y-Axis - species

x_min, x_max = x[:,0].min() - .5,x[:,0].max() + .5
y_min, y_max = x[:,1].min() - .5,x[:,1].max() + .5

#MESH
cmap_light = ListedColormap(['#AAAAFF','#AAFFAA','#FFAAAA'])
h = .02
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
knn = KNeighborsClassifier()
```

```
knn.fit(x,y)
Z = knn.predict(np.c_[xx.ravel(),yy.ravel()])
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx,yy,Z,cmap=cmap_light)

#Plot the training points
plt.scatter(x[:,0],x[:,1],c=y)
plt.xlim(xx.min(),xx.max())
plt.ylim(yy.min(),yy.max())

Out[126]: (-0.40000000000000002, 2.9800000000000031)
```

如图8-6所示，我们用花瓣数据描述鸢尾花卉的特征，得到了相应的决策边界。

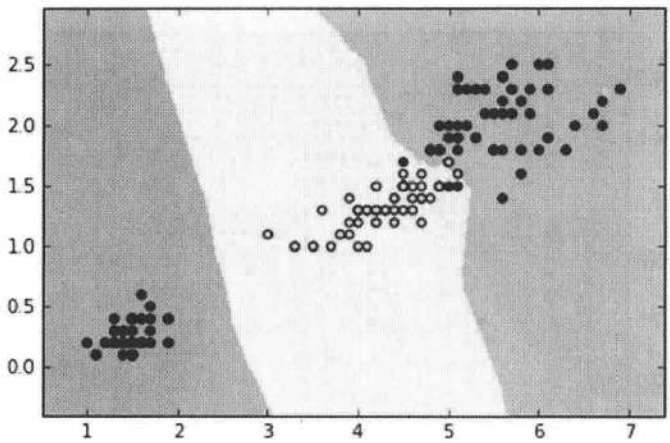


图8-6　描述花瓣大小的2D散点图中的决策边界

8.6　Diabetes 数据集

scikit-learn提供了多个数据集，其中就有Diabetes（糖尿病）数据集。人们首次使用它是在2004年（《统计年刊》，Efron、Hastie、Johnston和Tibshirani合著的一篇文章中用到了它）[①]。从那时起，人们拿它作为例子，广泛用于各种预测模型的研究和评估。

从这个数据集加载数据前，先要从scikit-learn库导入datasets模块。然后调用load_diabetes()函数加载数据集，并将其保存到diabetes变量中。

```
In [ ]: from sklearn import datasets
   ...: diabetes = datasets.load_diabetes()
```

该数据集包含442位病人的生理数据以及一年以后的病情发展情况，后者即为目标值。前10

① 《统计年刊》（*Annals of Statistics*），国际公认的统计学顶级期刊。Efron等人的文章题目为“Least Angle Regression”，见于该期刊2004年4月第32卷。

列数值为生理数据，分别表示以下特征：

- 年龄
- 性别
- 体质指数
- 血压
- S1、S2、S3、S4、S5、S6（六种血清的化验数据）

调用data属性，可以获取到测量数据。查看数据集，就会发现这些数据跟你想象中的差别很大。例如，我们来看一下第一位病人的10个数据。

```
diabetes.data[0]
Out[ ]:
array([ 0.03807591, 0.05068012, 0.06169621, 0.02187235, -0.0442235 ,
       -0.03482076, -0.04340085, -0.00259226, 0.01990842, -0.01764613])
```

这些数据其实是经过特殊处理得到的。10个数据中的每一个都做了均值中心化处理，然后又用标准差乘以个体数量调整了数值范围。验证就会发现任何一列的所有数值之和为1，比如对年龄这一列求和，所得结果非常接近于1。

```
np.sum(diabetes.data[:,0]**2)
Out[143]: 1.0000000000000746
```

即使这些数据因为经过规范化处理，所以难以读懂，但它们仍然表示10个生理特征，因而没有失去其价值或丢失统计信息。

表明疾病进展的数据，用target属性就能获取到。我们接下来得到的预测结果必须与之相符。

```
diabetes.target
Out[146]:
array([ 151.,  75., 141., 206., 135.,  97., 138.,  63., 110.,
        310., 101.,  69., 179., 185., 118., 171., 166., 144.,
         97., 168.,  68.,  49.,  68., 245., 184., 202., 137
. . .
```

我们得到了442个介于25到346之间的整数。

8.7 线性回归：最小平方回归

线性回归指的是用训练集数据创建线性模型的过程。最简单的形式则是基于下面这个用参数a和c刻画的直线方程。在计算参数a和c时，以最小化残差平方和为前提。

$$y = a * x + c$$

上述表达式中，x为训练集，y为目标值，a为斜率，c为模型所对应的直线的截距。如果要用scikit-learn库的线性回归预测模型，必须导入linear_model模块，然后用LinearRegression()构造函数创建预测模型，我们这里将其命名为linreg。

```
from sklearn import linear_model
linreg = linear_model.LinearRegression()
```

我们通过一个例子来练练手，数据集就使用前面介绍的Diabetes。首先，把包含442位病人病情的数据集分为训练集（前422位病人的数据）和测试集（剩余20位病人的数据）。

```
from sklearn import datasets
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]
```

在预测模型上调用fit()函数，使用训练集做训练。

```
linreg.fit(x_train,y_train)
Out[ ]: LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
```

训练完模型之后，调用预测模型的coef_属性，就可以得到每种（共10种）生理数据的回归系数*b*。

```
linreg.coef_
Out[164]:
array([ 3.03499549e-01, -2.37639315e+02, 5.10530605e+02,
        3.27736980e+02, -8.14131709e+02, 4.92814588e+02,
        1.02848452e+02,  1.84606489e+02, 7.43519617e+02,
        7.60951722e+01])
```

在预测模型linreg上调用predict()函数，传入测试集作为参数，将得到一列可以拿来与实际目标值相比较的预测目标值。

```
linreg.predict(x_test)
Out[ ]:
array([ 197.61846908, 155.43979328, 172.88665147, 111.53537279,
        164.80054784, 131.06954875, 259.12237761, 100.47935157,
        117.0601052 , 124.30503555, 218.36632793,  61.19831284,
        132.25046751, 120.3332925 ,  52.54458691, 194.03798088,
        102.57139702, 123.56604987, 211.0346317 ,  52.60335674])
y_test
Out[ ]:
array([ 233.,  91., 111., 152., 120., 67., 310., 94., 183.,
         66., 173.,  72.,  49.,  64., 48., 178., 104., 132.,
        220.,  57.])
```

方差是评价预测结果好坏的一个不错的指标。方差越接近于1，说明预测结果越准确。

```
linreg.score(x_test, y_test)
Out[ ]: 0.58507530226905713
```

现在我们就可以用线性回归方法分析单个生理因素与目标值之间的关系，比如我们可以从年龄开始看起。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn import datasets

diabetes = datasets.load_diabetes()
```

```
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]

x0_test = x_test[:,0]
x0_train = x_train[:,0]
x0_test = x0_test[:,np.newaxis]
x0_train = x0_train[:,np.newaxis]
linreg = linear_model.LinearRegression()
linreg.fit(x0_train,y_train)
y = linreg.predict(x0_test)
plt.scatter(x0_test,y_test,color='k')
plt.plot(x0_test,y,color='b',linewidth=3)

Out[230]: [<matplotlib.lines.Line2D at 0x380b1908>]
```

图8-7的蓝色直线表示病人的年龄和病情进展之间的相关性。

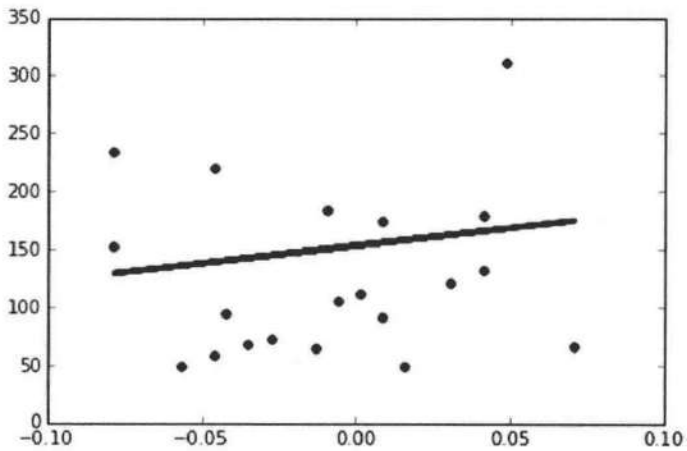


图8-7 表示一个特征和目标值之间相关性的线性回归图

Diabetes数据集实际上有10个生理因素。为了对训练集有个全面的认识，我们可以对每个生理特征进行回归分析，创建10个模型，并将其结果作成10幅图表。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn import datasets
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]
plt.figure(figsize=(8,12))
for f in range(0,10):
    xi_test = x_test[:,f]
```

```
xi_train = x_train[:,f]
xi_test = xi_test[:,np.newaxis]
xi_train = xi_train[:,np.newaxis]
linreg.fit(xi_train,y_train)
y = linreg.predict(xi_test)
plt.subplot(5,2,f+1)
plt.scatter(xi_test,y_test,color='k')
plt.plot(xi_test,y,color='b',linewidth=3)
```

图8-8为10幅线性回归图表，分别表示一个生理因素跟糖尿病病情进展之间的相关性。

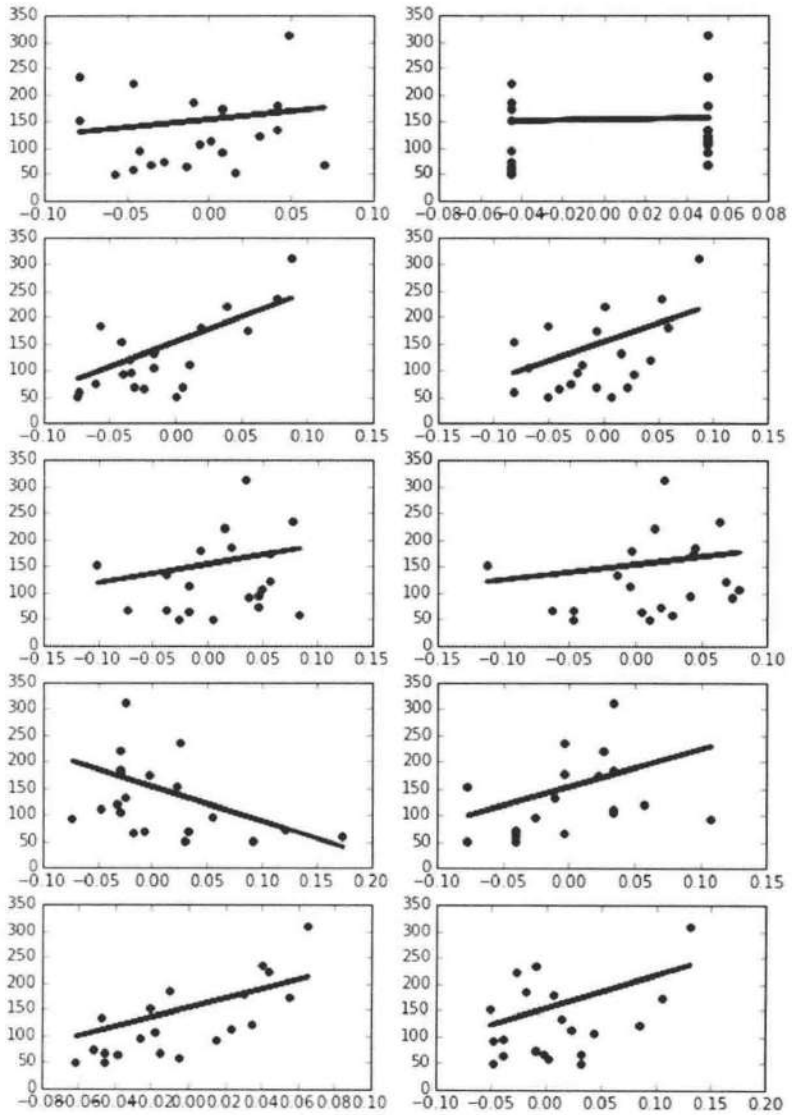


图8-8　10幅图表分别表示一个生理因素跟糖尿病病情进展之间的相关性

8.8 支持向量机

支持向量机（Support Vector Machines，SVM）指的是一系列机器学习方法，最初是在20世纪90年代初期由美国电话电报公司（AT&T）的Vapnik和同事一起开发的。这类方法的基础其实是支持向量算法，该算法是对广义肖像算法（Generalized Portrait）的扩展，后者是1963年Vapnik在苏联开发的。

简言之，SVM分类器是二元或判别模型，对两类数据进行区分。它最基础的任务是判断新观测数据属于两个类别中的哪一个。在学习阶段，这类分类器把训练数据映射到叫作决策空间（decision space）的多维空间，创建叫作决策边界的分离面，把决策空间分为两个区域。在最简单的线性可分的情况下，决策边界可以用平面（3D）或直线（2D）表示。在更复杂的情况中，分离面为曲面，形状更为复杂。

SVM算法既可用于回归问题，比如SVR（Support Vector Regression，支持向量回归）；也可用于分类，比如SVC（Support Vector Classification，支持向量分类）。

8.8.1 支持向量分类

如果想更好地理解该算法的工作原理，可以先来研究二维空间线性分类问题，其决策边界为一条直线，它把决策区域一分为二。我们用一个简单的训练集举个例子，它里面的数据点分属两个类别。训练集共包含11个数据点（观测到的数据），有取值范围均为0到4的两个不同属性。这些数据点的属性值存放在叫作x的NumPy数组中。数据点所属的类别用0或1表示，类别信息存储在数组y中。

把这些数据点绘制成散点图，观察它们在空间上的分布情况，该空间可以被称为决策空间（见图8-9）。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
Out[360]: <matplotlib.collections.PathCollection at 0x545634a8>
```

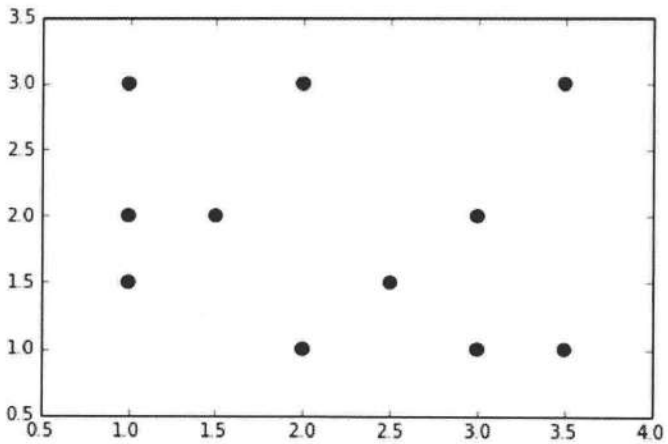


图8-9　训练集散点图展示了决策空间

既然定义好了训练集，我们就可以使用SVC算法进行训练了。该算法将会创建一条直线（决策边界），把决策区域分成两部分（见图8-10），直线所处的位置应该使得训练集中距离直线最近的几个数据点到直线的距离最大化。使用该条件，应该能够将数据点分成两部分，每一部分中所有数据点的类别相同。

用训练集训练SVC算法之前，先用SVC()构造函数定义模型，我们使用线性内核。（内核指用于模式分析的一类算法。）然后调用fit()函数，传入训练集作为参数。模型训练完成后，用decision_function()函数绘制决策边界。绘制散点图时，注意决策空间的两部分使用不同颜色。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='linear').fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k'], linestyles=['-'],levels=[0])
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)

Out[363]: <matplotlib.collections.PathCollection at 0x54acae10>
```

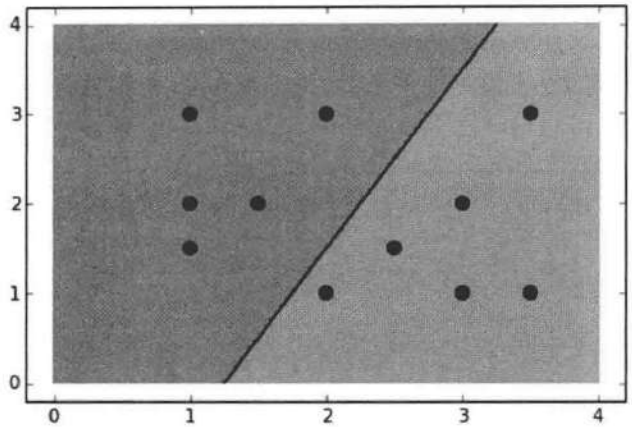


图8-10 决策区域一分为二

由图8-10可见，两块决策区域各包一个类别。可以说分类是比较成功的，除了有个蓝点被分到红色区域。

模型一旦训练完成，理解模型是如何进行预测的就很容易了。我们拿图来说，新观察到的数据点该分到哪一部分，取决于数据点在图中的位置。

反之，从偏程序设计的角度来说，predict()函数将会以数值形式返回数据点所属的类别（0为用蓝色表示的那一类，1为用红色表示的类）。

```
svc.predict([1.5,2.5])
Out[56]: array([0])

svc.predict([2.5,1])
Out[57]: array([1])
```

正则化是一个与SVC算法相关的概念，用参数C来设置：C值较小，表示计算间隔时，将分界线两侧的大量甚至全部数据点都考虑在内（泛化能力强）；C值较大，表示只考虑分界线附近的数据点（泛化能力弱）。若不指定C值，默认它的值为1。你可以通过support_vectors数组获取到参与计算间隔的数据点，为其添加高亮效果。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
    [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='linear',C=1).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

```
Out[23]: <matplotlib.collections.PathCollection at 0x177066a0>
```

这些点在散点图中用镶边的圆圈来表示。更进一步来说，它们处于分界线（图8-11中的虚线）附近的评价区域之内。

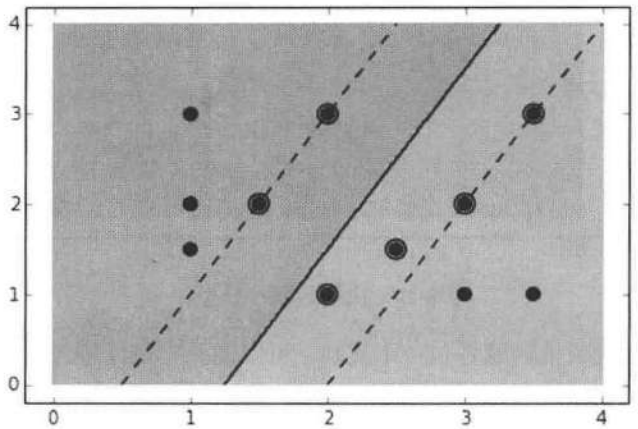


图8-11　参与计算间隔的数据点数量取决于参数C

为了理解参数C对决策边界的影响，可以给C赋一个很小的值，比如0.1。我们来看一下计算间隔用到了多少个数据点。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='linear',C=0.1).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

```
Out[24]: <matplotlib.collections.PathCollection at 0x1a01ecc0>
```

所使用的数据点数量增加了不少，分界线（决策边界）的位置也随之改变。但是现在有两个数据点处于错误的决策区域（见图8-12）。

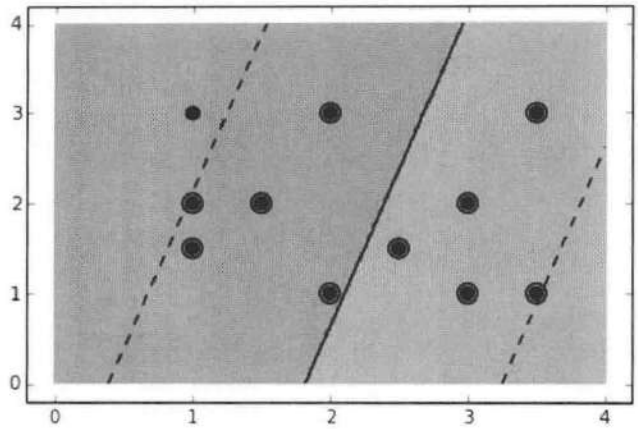


图8-12 随着C值的变小，计算间隔所考虑的数据点的数量逐渐增加

8.8.2 非线性 SVC

至此，你已经见识过SVC线性算法，它旨在定义一条把数据分为两类的分界线。还有一些更为复杂的SVC算法，它们能够建立曲线（2D）或曲面（3D），所依据的原则依旧是最大化离表面最近的数据点之间的距离。我们来看一个使用多项式内核的系统。

正如其名称所暗示的，你可以定义一条多项式曲线把决策空间分成两块。多项式的次数可用degree选项指定。即使是非线性SVC，C依旧是正则化回归系数。我们尝试使用内核为三次多项式、回归系数C取1的SVC算法。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='poly',C=1, degree=3).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)

Out[34]: <matplotlib.collections.PathCollection at 0x1b6a9198>
```

分类情况请见图8-13。

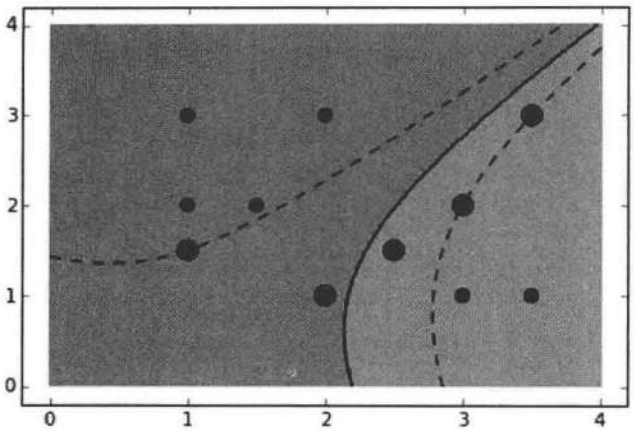


图8-13　内核为多项式的SVC算法生成的决策空间

另外一种非线性内核为径向基函数（Radial Basis Function，RBF）。这种内核生成的分隔面尝试把数据集的各个数据点分到沿径向方向分布的不同区域。、

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='rbf', C=1, gamma=3).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)

Out[43]: <matplotlib.collections.PathCollection at 0x1cb8d550>
```

从图8-14中我们可以见到两类决策区域，训练集所有数据点均处于正确的位置。

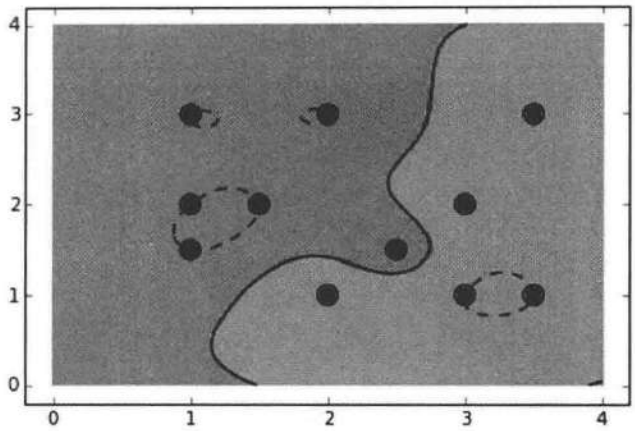


图8-14 用内核为rgb的SVC算法得到的决策区域

8.8.3 绘制 SVM 分类器对 Iris 数据集的分类效果图

前面的SVM例子使用的数据集非常简单。我们来看一下SVC算法对更复杂的数据集的分类情况。我们使用之前用过的Iris数据集。

前面用过的SVC算法从仅包含两个类别的训练集中学习。接下来这个例子中，我们把它扩展到三个类别，因为Iris数据集包含三个类别，对应三种花卉。

对于这个数据集，决策边界相互交叉，把决策空间（2D）或决策体（3D）分成多个部分。

两个线性模型均有线性决策边界（相交的超平面），而使用非线性内核的模型（多项式或高斯RBF）有非线性决策边界，后者在处理依赖于内核和参数的数据时更为灵活。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

iris = datasets.load_iris()
x = iris.data[:,:2]
y = iris.target
h = .05
svc = svm.SVC(kernel='linear',C=1.0).fit(x,y)
x_min,x_max = x[:,0].min() - .5, x[:,0].max() + .5
y_min,y_max = x[:,1].min() - .5, x[:,1].max() + .5
h = .02
X, Y = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,y_max,h))

Z = svc.predict(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z,alpha=0.4)
plt.contour(X,Y,Z,colors='k')
plt.scatter(x[:,0],x[:,1],c=y)

Out[49]: <matplotlib.collections.PathCollection at 0x1f2bd828>
```

在图8-15中，决策边界把决策空间分为三部分。

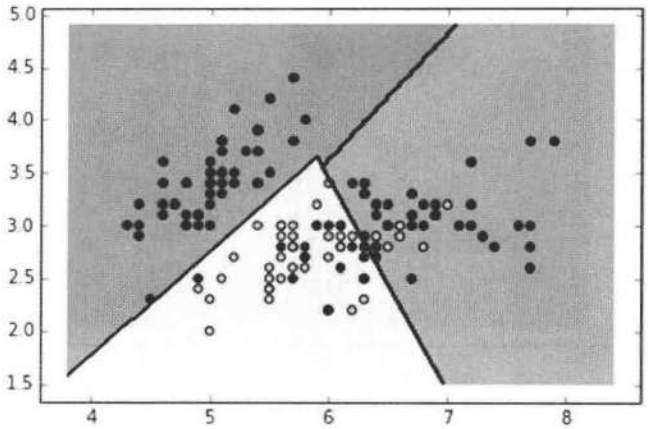


图8-15　决策边界把决策区域分为三个不同的部分

现在，我们来看一下如何用非线性内核，比如多项式内核，生成非线性决策边界。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

iris = datasets.load_iris()
x = iris.data[:,:2]
y = iris.target
h = .05
svc = svm.SVC(kernel='poly',C=1.0,degree=3).fit(x,y)
x_min,x_max = x[:,0].min() - .5, x[:,0].max() + .5
y_min,y_max = x[:,1].min() - .5, x[:,1].max() + .5

h = .02
X, Y = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,y_max,h))
Z = svc.predict(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z,alpha=0.4)
plt.contour(X,Y,Z,colors='k')
plt.scatter(x[:,0],x[:,1],c=y)

Out[50]: <matplotlib.collections.PathCollection at 0x1f4cc4e0>
```

由图8-16可见，跟之前用线性内核得到的区域相比，用多项式内核得到的决策边界划分的决策区域差别较大。

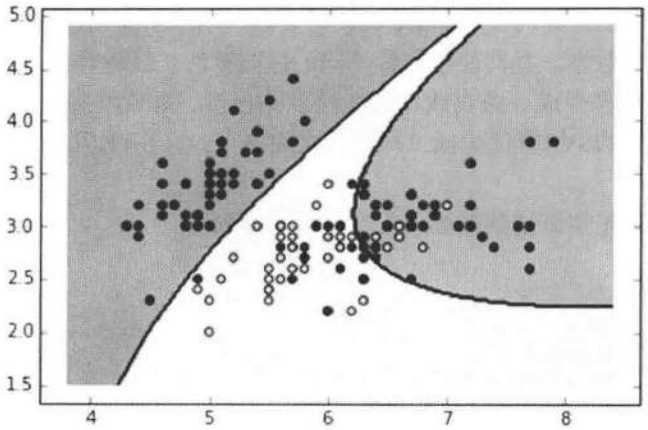


图8-16 用多项式内核得到的决策区域，蓝色部分和红色未直接连接在一起

接着可以再换用RBF内核，观察分区结果会有什么不同。

```
svc = svm.SVC(kernel='rbf', gamma=3, C=1.0).fit(x,y)
```

图8-17为用RBF内核生成的径向决策区域。

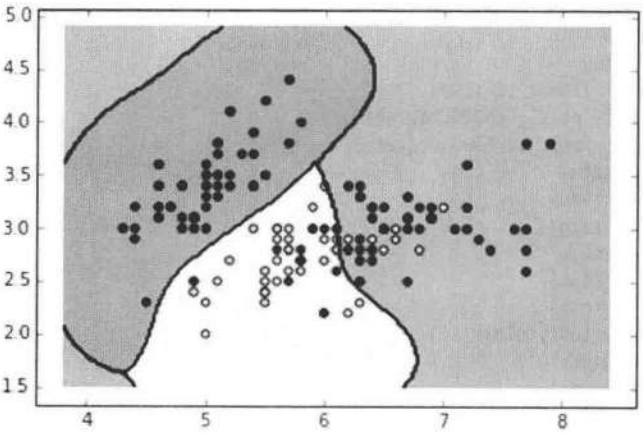


图8-17 用RBF内核得到的径向决策区域

8.8.4 支持向量回归

SVC方法经扩展甚至可用来解决回归问题，这种方法称作支持向量回归（Support Vector-Regression，SVR）。

SVC生成的模型实际上没有使用全部训练集数据，而只是使用其中一部分，也就是离决策边界最近的数据点。类似地，SVR生成的模型也只依赖于部分训练数据。

我们将介绍SVR算法是如何使用Diabetes数据集的，本章前面用过该数据集。举例起见，我们将只考虑第三个生理数据。我们使用三种不同的回归算法：线性和两个非线性（多项式）。使用线性内核的SVR算法将生成一条直线作为线性预测模型，非常类似于前面见过的线性回归算法，而使用多项式内核的SVR算法生成二次和三次曲线。SVR()函数几乎与前面见过的SVC()函数完全相同。

唯一需要考虑的就是测试集数据必须按升序形式排列。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn import datasets
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]

x0_test = x_test[:,2]
x0_train = x_train[:,2]
x0_test = x0_test[:,np.newaxis]
x0_train = x0_train[:,np.newaxis]

x0_test.sort(axis=0)
x0_test = x0_test*100
x0_train = x0_train*100
svr = svm.SVR(kernel='linear',C=1000)
svr2 = svm.SVR(kernel='poly',C=1000,degree=2)
svr3 = svm.SVR(kernel='poly',C=1000,degree=3)
svr.fit(x0_train,y_train)
svr2.fit(x0_train,y_train)
svr3.fit(x0_train,y_train)
y = svr.predict(x0_test)
y2 = svr2.predict(x0_test)
y3 = svr3.predict(x0_test)
plt.scatter(x0_test,y_test,color='k')
plt.plot(x0_test,y,color='b')
plt.plot(x0_test,y2,c='r')
plt.plot(x0_test,y3,c='g')

Out[155]: [<matplotlib.lines.Line2D at 0x262e10b8>]
```

三条回归曲线分别用三种颜色来表示。线性回归使用蓝色；二次曲线也就是抛物线，使用红色；三次曲线使用绿色（见图8-18）。

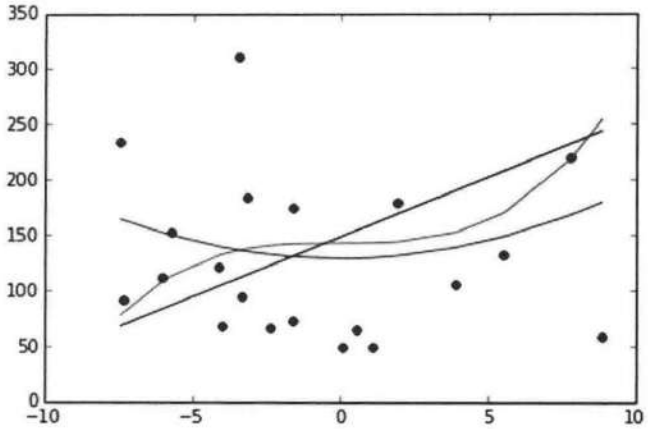


图8-18 用训练集数据生成的三条趋势极为不同的回归曲线

8.9 小结

本章讲的是如何用scikit-learn库解决最简单的回归和分类问题。我们通过几个很实用的例子，介绍了预测模型的验证过程所涉及的多个概念。

下一章，我们将借助一个简单实用的例子，全面介绍数据分析的各个步骤。所有这一切都将在集交互性和创新性于一身的IPython Notebook中实现，它非常适合以交互式文档的形式分享数据分析的每一步。这种交互式文档可用作报告，也可以用来以网页形式展示数据分析成果。

第9章

数据分析实例——气象数据

气象数据是在网上很容易找到的一类数据。很多网站都提供以往的气压、气温、湿度和降雨量等气象数据。只需指定位置和日期，就能获取一个气象数据文件。这些测量数据是由气象站收集的。气象数据这类数据源涵盖的信息范围较广。我们在第1章讲过，数据分析的目的是把原始数据转化为信息，再把信息转化为知识，因此拿气象数据作为数据分析的对象来讲解数据分析全过程再合适不过。

在这一章，我们通过一个简单的例子，讲解气象数据的使用方法。该例子有助于你从整体上了解如何应用前几章所讲的内容。

9.1 待检验的假设：靠海对气候的影响

写作本章时，虽正值夏初，却已酷热难耐，住在大城市的人感受更为强烈。于是周末很多人到山村或海滨城市去游玩，放松一下身心，远离内陆城市的闷热天气。我常常想，靠海对气候有什么影响？

这个问题可以作为数据分析的一个不错的出发点。我不想把本章写成科学类读物，只是想借助这样一种方式，让数据分析爱好者能够把所学用于实践，解决“海洋对一个地区的气候有何影响”这个问题。

研究系统：亚得里亚海和波河流域

既然已定义好问题，就需要寻找适合研究数据的系统，提供适合回答这个问题的环境。

首先，需要找到一片海域供你研究。我住在意大利，可选择的海有很多，因为意大利是一个被海洋包围的半岛国家。为什么要把自己的选择局限在意大利呢？因为我们所研究的问题刚好和意大利人的一种典型行为相关，也就是夏天我们喜欢躲在海边，以躲避内陆的酷热。我不知道在其他国家这种行为是否也很普遍，因此我只把自己熟悉的意大利作为一个系统进行研究。

但是你可能会考虑研究意大利的哪个地区呢？上面说过，意大利是半岛国家，找到可研究的海域不是问题，但是如何衡量海洋对距其远近不同的地方的影响？这就引出了一个大问题。意大利其实多山地，离海差不多远，可以彼此作为参照的内陆区域较少。为了衡量海洋对气候的影响，我排除了山地，因为山地也许会引入其他很多因素，比如海拔。

意大利波河流域这块区域就很适合研究海洋对气候的影响。这一片平原东起亚得里亚海，向内陆延伸数百公里（见图9-1）。它周边虽不乏群山环绕，但由于它很宽广，削弱了群山的影响。此外，该区域城镇密集，也便于选取一组离海远近不同的城市。我们所选的几个城市，两个城市间的最大距离约为400公里。

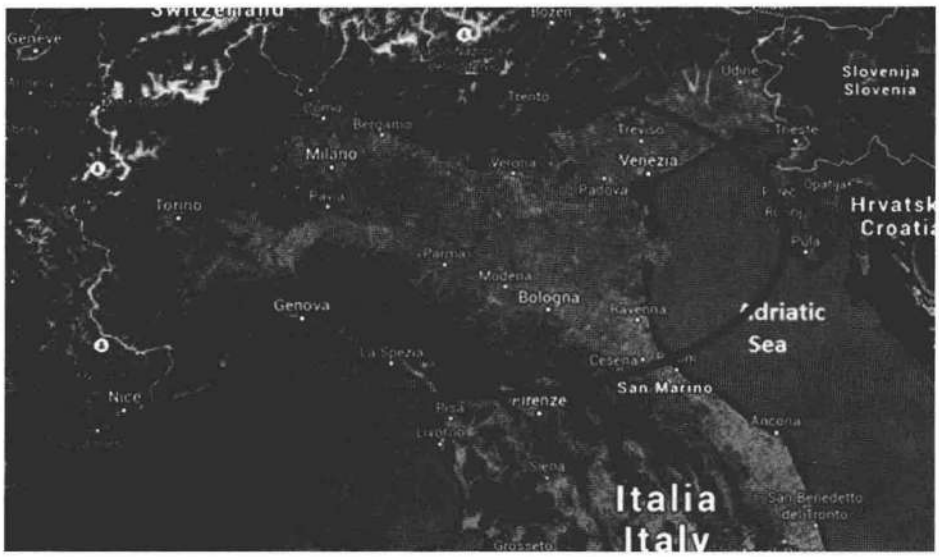


图9-1 波河流域和亚得里亚海（谷歌地图）

第一步，选10个城市作为参照组。选择城市时，注意它们要能代表整个平原地区（见图9-2）。

图9-2 作为参照组的10个城市（还有一个海滨城市，作为计算其他城市离海远近的基点[①]）

① 10个城市与该基点之间的距离即表示它们跟海洋之间的距离。

如图9-2所示，我们选取了10个城市。随后将分析它们的天气数据，其中5个城市在距海100公里范围内，其余5个距海100～400公里。

选作样本的城市列表如下：

- Ferrara（费拉拉）
- Torino（都灵）
- Mantova（曼托瓦）
- Milano（米兰）
- Ravenna（拉文纳）
- Asti（阿斯蒂）
- Bologna（博洛尼亚）
- Piacenza（皮亚琴察）
- Cesena（切塞纳）
- Faenza（法恩莎）

现在，我们需要弄清楚这些城市离海有多远。方法有多种。这里使用TheTimeNow网站（http://www.thetimenow.com/distance-calculator.php）提供的服务，它支持多种语言（见图9-3）。

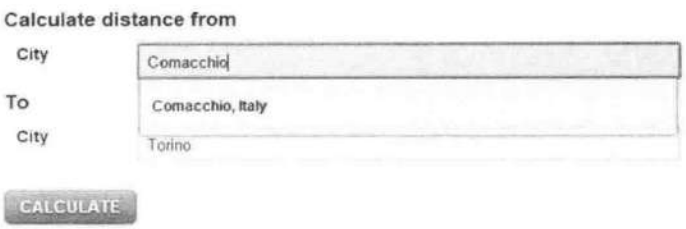


图9-3　TheTimeNow网站提供的计算两个城市间距离的服务

有了计算两城市间距离这样的服务，我们就可以计算每个城市与海之间的距离。你可以选择海滨城市Comacchio作为基点，计算其他城市与它之间的距离（见图9-2）。使用上述服务计算完所有距离后，得到的结果如表9-1所示。

表9-1　10个城市与海之间的距离

城　市	距离（公里）	备　注
Ravenna	8	用谷歌地图测量
Cesena	14	用谷歌地图测量
Faenza	37	Faenza-Ravenna 的距离，再加 8 公里
Ferrara	47	Ferrara-Comacchio 的距离
Bologna	71	Bologna-Comacchio 的距离
Mantova	121	Mantova-Comacchio 的距离
Piacenza	200	Piacenza-Comacchio 的距离

（续）

城　市	距离（公里）	备　注
Milano	250	Milano-Comacchio 的距离
Asti	315	Asti-Comacchio 的距离
Torino	357	Torino-Comacchio 的距离

9.2 数据源

定义好要研究的系统之后，我们就需要创建数据源，以获取研究所需的数据。上网浏览一番，你就会发现很多网站都提供世界各地的气象数据，其中就有Open Weather Map，它的网址是http://openweathermap.org/（见图9-4）。

图9-4　Open Weather Map网站

该网站，http://api.openweathermap.org/data/2.5/history/city?q=Atlanta,US，提供以下功能：在请求的URL中指定城市，即可获取该城市的气象数据。

该请求返回一个JSON文件，文件内容为美国亚特兰大市的气象数据（见图9-5）。稍后将把该JSON文件交由Python的pandas库处理。

图9-5　包含城市气象数据的JSON文件

9.3　用 IPython Notebook 做数据分析

本章用IPython Notebook分析数据，便于逐段输入和研究代码。

在命令行输入并运行以下代码，启动IPython Notebook应用：

```
ipython notebook
```

服务启动后，新建一个Notebook文件。

在新建的Notebook文件中，先导入以下库：

```
import numpy as np
import pandas as pd
import datetime
```

接着获取10个城市的气象数据。服务器可能会比较繁忙，因此最好一次只加载一个城市的数据。否则，由于服务器无法同时满足所有请求，你可能会得到错误信息。

```
ferrara = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Ferrara,IT')
torino = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Torino,IT')
mantova = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Mantova,IT')
milano = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Milano,IT')
ravenna = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Ravenna,IT')
asti = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Asti,IT')
bologna = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Bologna,IT')
piacenza = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Piacenza,IT')
cesena = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Cesena,IT')
faenza = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Faenza,IT')
```

我们刚刚用read_json()函数把10个城市的JSON数据读入到DataFrame数据结构中。JSON结

构已自动转换为DataFrame的列表式结构。转换为这种格式之后，研究和开发起来将更简单（见图9-6）。

```
In [153]: faenza
```

Out[153]:

	calctime	city_id	cnt	cod	list	message
0	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
1	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
2	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
3	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
4	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
5	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
6	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
7	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
8	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
9	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
10	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 18.9...	
11	0.1222	3177300	19	200	{u'clouds': {u'all': 40}, u'rain': {u'1h': 41....	
12	0.1222	3177300	19	200	{u'clouds': {u'all': 40}, u'rain': {u'1h': 41....	
13	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
14	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
15	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
16	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
17	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
18	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	

图9-6 直接把JSON文件转换为pandas DataFrame后得到的列表式结构

如上所见，转换后得到的数据还不能直接用于数据分析。所有的气象数据存储在list这一列，而该列元素还是JSON结构，因此你得编写一个函数，把这个树状数据结构转换为列表式结构，后者更适合于用pandas库做分析。

```
def prepare(city_list,city_name):
    temp = [ ]
    humidity = [ ]
    pressure = [ ]
    description = [ ]
    dt = [ ]
    wind_speed = [ ]
    wind_deg = [ ]
for row in city_list:
```

```
        temp.append(row['main']['temp']-273.15)
        humidity.append(row['main']['humidity'])
        pressure.append(row['main']['pressure'])
        description.append(row['weather'][0]['description'])
        dt.append(row['dt'])
        wind_speed.append(row['wind']['speed'])
        wind_deg.append(row['wind']['deg'])
    headings = ['temp','humidity','pressure','description','dt','wind_speed','wind_deg']
    data = [temp,humidity,pressure,description,dt,wind_speed,wind_deg]
    df = pd.DataFrame(data,index=headings)
    city = df.T
    city['city'] = city_name
    city['day'] = city['dt'].apply(datetime.datetime.fromtimestamp)
    return city
```

上述函数接收DataFrame对象和相应的城市名称两个参数，返回一个新DataFrame对象。list列的JSON结构包含多个指标，我们从中选择几个最适合研究系统的指标。

- Temperature（气温）
- Humidity（湿度）
- Pressure（气压）
- Description（详情）
- Wind Speed（风速）
- Wind Degree（风力）

所有这些指标都与dt列的观测时间有关，该列数据类型为时间戳格式，可读性较差，因此我们需要将其转换为我们更为熟悉的日期格式。转换后生成的新列为day。

```
city['day'] = city['dt'].apply(datetime.datetime.fromtimestamp)
```

气温用开氏度表示。为了将其转化为摄氏度，每个温度值都要减去273.15。

最后添加city列，为DataFrame对象的每一行数据添加城市名称。

接着使用该函数分别处理前面创建的10个DataFrame对象。

```
df_ferrara = prepare(ferrara.list,'Ferrara')
df_milano = prepare(milano.list,'Milano')
df_mantova = prepare(mantova.list,'Mantova')
df_ravenna = prepare(ravenna.list,'Ravenna')
df_torino = prepare(torino.list,'Torino')
df_asti = prepare(asti.list,'Asti')
df_bologna = prepare(bologna.list,'Bologna')
df_piacenza = prepare(piacenza.list,'Piacenza')
df_cesena = prepare(cesena.list,'Cesena')
df_faenza = prepare(faenza.list,'Faenza')
```

现在，要添加的另一个特性是城市与海洋的距离。

```
df_ravenna['dist'] = 8
df_cesena['dist'] = 14
df_faenza['dist'] = 37
df_ferrara['dist'] = 47
df_bologna['dist'] = 71
```

```
df_mantova['dist'] = 121
df_piacenza['dist'] = 200
df_milano['dist'] = 250
df_asti['dist'] = 315
df_torino['dist'] = 357
```

完成上述处理后，每个城市都表示成pandas库的DataFrame对象。该DataFrame的内部结构如图9-7所示。

```
In [154]: df_faenza
Out[154]:
```

	temp	humidity	pressure	description	dt	wind_speed	wind_deg	city	day	dist
0	25.44	69	1018	very heavy rain	1435390925	1.29	14.5002	Faenza	2015-06-27 09:42:05	37
1	26.38	73	1017	very heavy rain	1435394243	2.1	100	Faenza	2015-06-27 10:37:23	37
2	27.7	69	1017	very heavy rain	1435399019	3.1	120	Faenza	2015-06-27 11:56:59	37
3	29.04	61	1016	very heavy rain	1435402422	3.1	110	Faenza	2015-06-27 12:53:42	37
4	29.11	69	1016	very heavy rain	1435406058	3.6	110	Faenza	2015-06-27 13:54:18	37
5	29.33	65	1015	very heavy rain	1435409704	5.7	110	Faenza	2015-06-27 14:55:04	37
6	29.2	65	1014	very heavy rain	1435416898	5.1	110	Faenza	2015-06-27 16:54:58	37
7	28.88	65	1014	very heavy rain	1435420542	6.2	120	Faenza	2015-06-27 17:55:42	37
8	27.53	65	1014	very heavy rain	1435424296	6.7	120	Faenza	2015-06-27 18:58:16	37
9	26.12	73	1013	very heavy rain	1435427936	6.2	120	Faenza	2015-06-27 19:58:56	37
10	22.32	94	1015	very heavy rain	1435438357	2.6	90	Faenza	2015-06-27 22:52:37	37
11	21.38	83	1016	very heavy rain	1435442241	5.7	90	Faenza	2015-06-27 23:57:21	37
12	21.16	83	1016	very heavy rain	1435445863	2.1	210	Faenza	2015-06-28 00:57:43	37
13	20	94	1016	very heavy rain	1435453232	1.39	218.504	Faenza	2015-06-28 03:00:32	37
14	19.48	93	1016	very heavy rain	1435456487	2.1	330	Faenza	2015-06-28 03:54:47	37
15	19.16	93	1016	very heavy rain	1435460042	1.5	320	Faenza	2015-06-28 04:54:02	37
16	18.62	93	1016	very heavy rain	1435463880	0.5	300	Faenza	2015-06-28 05:58:00	37
17	19.34	88	1016	very heavy rain	1435467179	0.5	270	Faenza	2015-06-28 06:52:59	37
18	21.05	88	1016	very heavy rain	1435470851	2.1	260	Faenza	2015-06-28 07:54:11	37

图9-7 表示一个城市气象数据的DataFrame结构

留心观察就会发现，以列表式数据结构展示测量数据时，条理分明，可读性强。更进一步看，每一行数据为当天刚过去的大约20个小时中每一小时的观测结果。然而，你可能注意到了图9-7中只有18行数据。事实上，查看其他城市的话，你也会发现，由于观测系统有时会发生故障，从而导致某些时刻没有观测结果这种情况。但是，如果采集到19条数据，比如我们这里，它们也足以描述当天各气象数据的趋势。

我们最好还是查看一下10个DataFrame的大小。如果哪个城市的气象数据不足以描述当天的气象趋势，我们好换用其他城市。

```
print df_ferrara.shape
print df_milano.shape
print df_mantova.shape
print df_ravenna.shape
print df_torino.shape
print df_asti.shape
```

```
print df_bologna.shape
print df_piacenza.shape
print df_cesena.shape
print df_faenza.shape
```

输出结果为：

```
(20, 9)
(18, 9)
(20, 9)
(18, 9)
(20, 9)
(20, 9)
(20, 9)
(20, 9)
(20, 9)
(19, 9)
```

如上所见，我们选用的这10个城市很不错，因为它们为进一步分析数据提供了足够的信息。

既然得到了包含当天气象数据的10个DataFrame对象，我们需要将它们保存下来。别忘了，如果第二天（甚至是几个小时之后）再次执行代码，所有的结果会发生变化！因此最好把DataFrame数据保存到CSV文件中，以便再次使用。

```
df_ferrara.to_csv('ferrara_270615.csv')
df_milano.to_csv('milano_270615.csv')
df_mantova.to_csv('mantova_270615.csv')
df_ravenna.to_csv('ravenna_270615.csv')
df_torino.to_csv('torino_270615.csv')
df_asti.to_csv('asti_270615.csv')
df_bologna.to_csv('bologna_270615.csv')
df_piacenza.to_csv('piacenza_270615.csv')
df_cesena.to_csv('cesena_270615.csv')
df_faenza.to_csv('faenza_270615.csv')
```

运行上述命令，将会在工作目录生成10个CSV文件。

如果你想用我在本章使用的数据，需要加载写作本章时我保存的10个CSV文件。

```
df_ferrara.read_csv('ferrara_270615.csv')
df_milano.read_csv('milano_270615.csv')
df_mantova.read_csv('mantova_270615.csv')
df_ravenna.read_csv('ravenna_270615.csv')
df_torino.read_csv('torino_270615.csv')
df_asti.read_csv('asti_270615.csv')
df_bologna.read_csv('bologna_270615.csv')
df_piacenza.read_csv('piacenza_270615.csv')
df_cesena.read_csv('cesena_270615.csv')
df_faenza.read_csv('faenza_270615.csv')
```

从数据可视化入手分析收集到的数据是常见的做法。前面讲过，matplotlib库提供一系列图表生成工具，能够以可视化形式表示数据。数据可视化在数据分析阶段非常有助于发现研究系统的一些特点。

导入以下必要的库：

```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
```

举例来说，非常简单的分析方法是先分析一天中气温的变化趋势。我们以城市米兰为例。

```
y1 = df_milano['temp']
x1 = df_milano['day']
fig, ax = plt.subplots()
plt.xticks(rotation=70)
hours = mdates.DateFormatter('%H:%M')
ax.xaxis.set_major_formatter(hours)
ax.plot(x1,y1,'r')
```

执行上述代码，将得到如图9-8所示的图像。由图可见，气温走势接近正弦曲线，从早上开始气温逐渐升高，最高温出现在下午两点到六点之间，随后气温逐渐下降，在第二天早上六点时达到最低值。

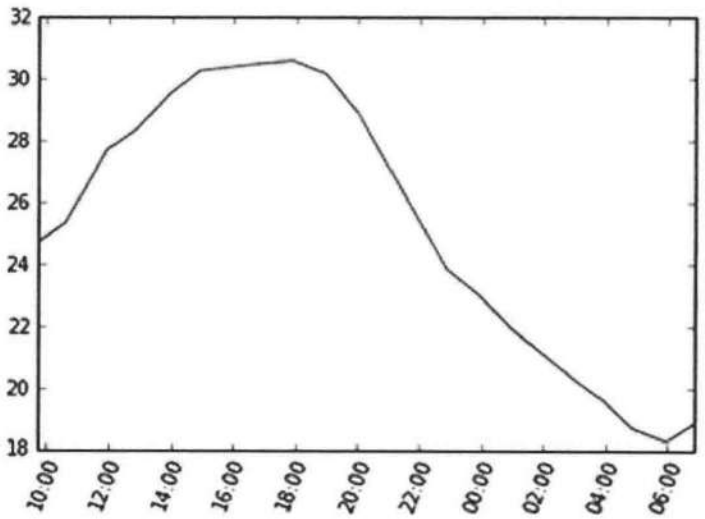


图9-8 米兰某一天的气温趋势图

我们进行数据分析的目的是尝试解释是否能够评估海洋是怎样影响气温的，以及是否能够影响气温趋势，因此我们同时来看几个不同城市的气温趋势。这是检验分析方向是否正确的唯一方式。因此，我们选择三个离海最近以及三个离海最远的城市。

```
y1 = df_ravenna['temp']
x1 = df_ravenna['day']
y2 = df_faenza['temp']
x2 = df_faenza['day']
y3 = df_cesena['temp']
x3 = df_cesena['day']
y4 = df_milano['temp']
x4 = df_milano['day']
y5 = df_asti['temp']
```

```
x5 = df_asti['day']
y6 = df_torino['temp']
x6 = df_torino['day']
fig, ax = plt.subplots()
plt.xticks(rotation=70)
hours = mdates.DateFormatter('%H:%M')
ax.xaxis.set_major_formatter(hours)
plt.plot(x1,y1,'r',x2,y2,'r',x3,y3,'r')
plt.plot(x4,y4,'g',x5,y5,'g',x6,y6,'g')
```

上述代码将生成如图9-9所示的图表。离海最近的三个城市的气温曲线使用红色，而离海最远的三个城市的曲线使用绿色。

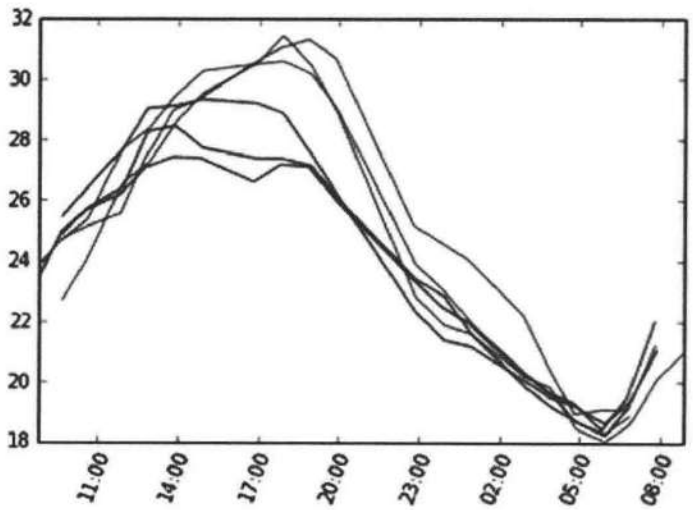


图9-9　六个城市的气温趋势（红色：离海最近的城市；绿色：离海最远的城市）

如图9-9所示，结果看起来不错。离海最近的三个城市的最高气温比离海最远的三个城市低不少，而最低气温看起来差别较小。

我们可以沿着这个方向做深入研究，收集10个城市的最高温和最低温，用线性图表示气温最值点和离海远近之间的关系。

```
dist = [df_ravenna['dist'][0],
     df_cesena['dist'][0],
     df_faenza['dist'][0],
     df_ferrara['dist'][0],
     df_bologna['dist'][0],
     df_mantova['dist'][0],
     df_piacenza['dist'][0],
     df_milano['dist'][0],
     df_asti['dist'][0],
     df_torino['dist'][0]
]
temp_max = [df_ravenna['temp'].max(),
     df_cesena['temp'].max(),
```

```
    df_faenza['temp'].max(),
    df_ferrara['temp'].max(),
    df_bologna['temp'].max(),
    df_mantova['temp'].max(),
    df_piacenza['temp'].max(),
    df_milano['temp'].max(),
    df_asti['temp'].max(),
    df_torino['temp'].max()
]
temp_min = [df_ravenna['temp'].min(),
    df_cesena['temp'].min(),
    df_faenza['temp'].min(),
    df_ferrara['temp'].min(),
    df_bologna['temp'].min(),
    df_mantova['temp'].min(),
    df_piacenza['temp'].min(),
    df_milano['temp'].min(),
    df_asti['temp'].min(),
    df_torino['temp'].min()
]
```

先把最高温画出来。

```
plt.plot(dist,temp_max,'ro')
```

结果如图9-10所示。

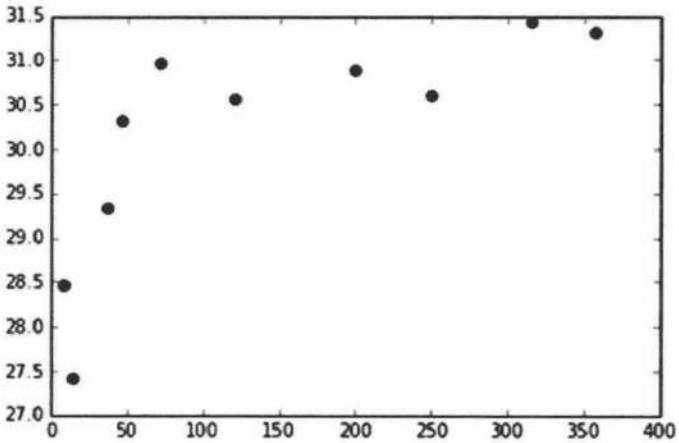


图9-10 最高温变化趋势与离海远近之间的关系

如图9-10所示，现在你可以证实，海洋对气象数据具有一定程度的影响这个假设是正确的（至少这一天如此☺）。

进一步观察上图，你会发现海洋的影响衰减得很快，离海60～70公里开外，气温就已攀升到高位。

用线性回归算法得到两条直线，分别表示两种不同的气温趋势，这样做很有趣。我们可以使用scikit-learn库的SVR方法。

```
from sklearn.svm import SVR
svr_lin1 = SVR(kernel='linear', C=1e3)
svr_lin2 = SVR(kernel='linear', C=1e3)
svr_lin1.fit(x1, y1)
svr_lin2.fit(x2, y2)
xp1 = np.arange(10,100,10).reshape((9,1))
xp2 = np.arange(50,400,50).reshape((7,1))
yp1 = svr_lin1.predict(xp1)
yp2 = svr_lin2.predict(xp2)

plt.plot(xp1, yp1, c='r', label='Strong sea effect')
plt.plot(xp2, yp2, c='b', label='Light sea effect')
plt.axis((0,400,27,32))
plt.scatter(x, y, c='k', label='data')
```

上述代码将生成如图9-11所示的图像。

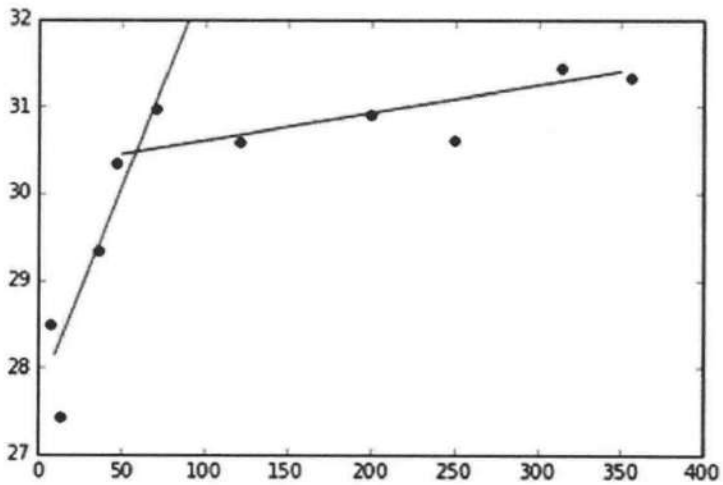


图9-11　最高气温和距离的相关性趋势图

如上所见，离海60公里以内，气温上升速度很快，从28度陡升至31度，随后增速渐趋缓和（如果还继续增长的话），更长的距离才会有小幅上升。这两种趋势可分别用两条直线来表示，直线的表达式为：

$$x = ax + b$$

其中a为斜率，b为截距。

```
print svr_lin1.coef_
print svr_lin1.intercept_
print svr_lin2.coef_
print svr_lin2.intercept_
```

```
[[-0.04794118]]
[ 27.65617647]
[[-0.00317797]]
[ 30.2854661]
```

你可能会考虑将这两条直线的交点作为受海洋影响和不受海洋影响的区域的分界点，或者至少是海洋影响较弱的分界点。

```
from scipy.optimize import fsolve

def line1(x):
    a1 = svr_lin1.coef_[0][0]
    b1 = svr_lin1.intercept_[0]
    return -a1*x + b1

def line2(x):
    a2 = svr_lin2.coef_[0][0]
    b2 = svr_lin2.intercept_[0]
    return -a2*x + b2
def findIntersection(fun1,fun2,x0):
 return fsolve(lambda x : fun1(x) - fun2(x),x0)

result = findIntersection(line1,line2,0.0)
print "[x,y] = [ %d , %d ]" % (result,line1(result))
x = numpy.linspace(0,300,31)
plt.plot(x,line1(x),x,line2(x),result,line1(result),'ro')
```

执行上述代码，将得到交点的坐标

```
[x,y] = [ 58, 30 ]
```

并得到如图9-12所示的图表。

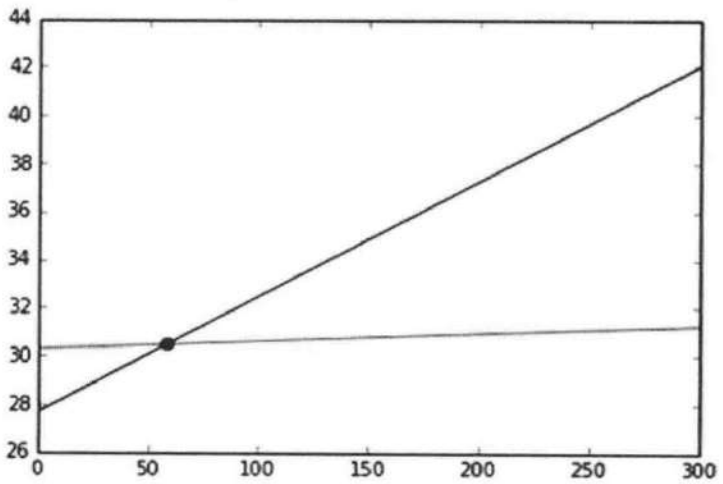


图9-12 由线性回归所得到的两条直线的交点

因此，你可以说海洋对气温产生影响的平均距离（该天的情况）为58公里。

现在，我们可以转而分析最低气温。

```
plt.axis((0,400,15,25))
plt.plot(dist,temp_min,'bo')
```

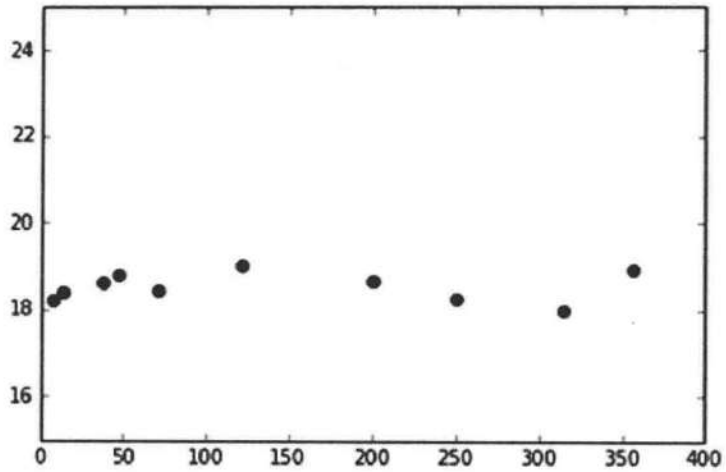


图9-13　最低气温几乎与离海远近无关

在这个例子中，很明显夜间或早上6点左右的最低温与海洋无关。如果没记错的话，小时候老师教给大家的是海洋能够缓和低温，或者说夜间海洋释放白天吸收的热量。但是从我们得到情况来看并非如此。我们刚使用的是意大利夏天的气温数据，而验证该假设在冬天或其他地方是否也成立，将会非常有趣。

10个DataFrame对象中还包含湿度这个气象数据。因此，你也可以考察当天三个近海城市和三个内陆城市的湿度趋势。

```
y1 = df_ravenna['humidity']
x1 = df_ravenna['day']
y2 = df_faenza['humidity']
x2 = df_faenza['day']
y3 = df_cesena['humidity']
x3 = df_cesena['day']
y4 = df_milano['humidity']
x4 = df_milano['day']
y5 = df_asti['humidity']
x5 = df_asti['day']
y6 = df_torino['humidity']
x6 = df_torino['day']
fig, ax = plt.subplots()
plt.xticks(rotation=70)
hours = mdates.DateFormatter('%H:%M')
ax.xaxis.set_major_formatter(hours)
plt.plot(x1,y1,'r',x2,y2,'r',x3,y3,'r')
plt.plot(x4,y4,'g',x5,y5,'g',x6,y6,'g')
```

上述代码将生成如图9-14所示的图表。

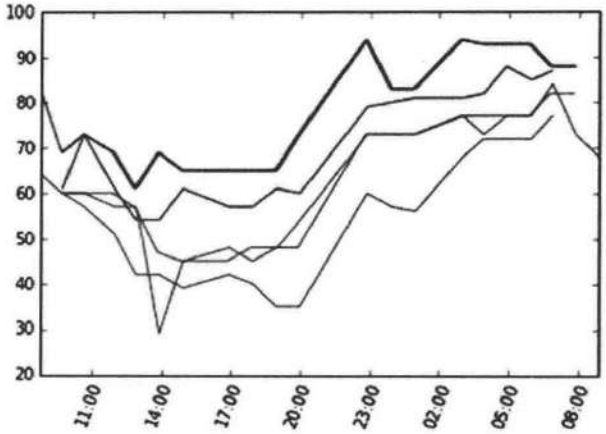


图9-14 三个近海城市（红色）和三个内陆城市（绿色）一天中的湿度趋势

乍看上去好像近海城市的湿度要大于内陆城市，全天湿度差距在20%左右。我们再来看一下湿度的极值和离海远近之间的关系，是否跟我们的第一印象相符。

```
hum_max = [df_ravenna['humidity'].max(),
     df_cesena['humidity'].max(),
     df_faenza['humidity'].max(),
     df_ferrara['humidity'].max(),
     df_bologna['humidity'].max(),
     df_mantova['humidity'].max(),
     df_piacenza['humidity'].max(),
     df_milano['humidity'].max(),
     df_asti['humidity'].max(),
     df_torino['humidity'].max()
]
plt.plot(dist,hum_max,'bo')
```

我们把10个城市的最大湿度与离海远近之间的关系做成图表，请见图9-15。

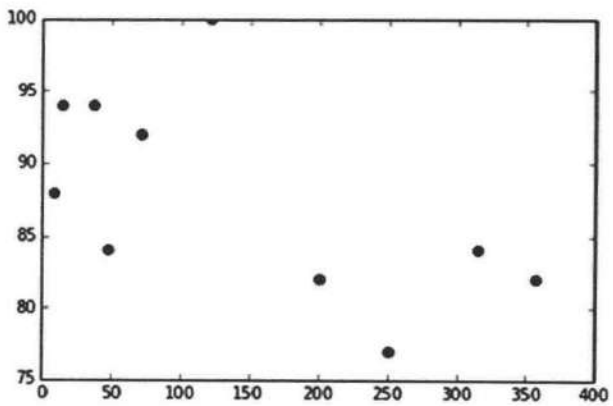


图9-15 最大湿度与离海远近之间关系的趋势图

```
hum_min = [df_ravenna['humidity'].min(),
     df_cesena['humidity'].min(),
     df_faenza['humidity'].min(),
     df_ferrara['humidity'].min(),
     df_bologna['humidity'].min(),
     df_mantova['humidity'].min(),
     df_piacenza['humidity'].min(),
     df_milano['humidity'].min(),
     df_asti['humidity'].min(),
     df_torino['humidity'].min()
]
plt.plot(dist,hum_min,'bo')
```

再来把10个城市的最小湿度与离海远近之间的关系做成图表，请见图9-16。

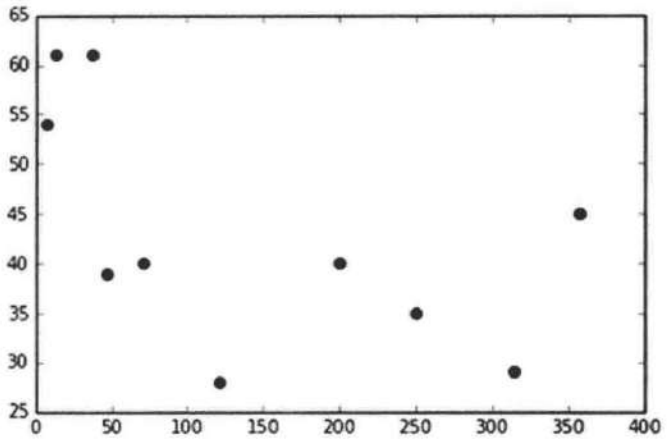


图9-16　最小湿度与离海远近之间关系的趋势图

由图9-15和图9-16可以确定，近海城市无论是最大还是最小湿度都要高于内陆城市。然而在我看来，我们还不能说湿度和距离之间存在线性关系或者其他能用曲线表示的关系。我们采集的数据点数量（10）太少，不足以描述这类趋势。

9.4　风向频率玫瑰图

在我们采集的每个城市的气象数据中，下面两个与风有关：

- 风力（风向）
- 风速

分析存放每个城市气象数据的DataFrame就会发现，风速不仅跟一天的时间段相关联，还与一个介于0~360度的方向有关。例如，每一条测量数据也包含风吹来的方向（图9-17）。

```
df_ravenna[['wind_deg','wind_speed','day']]
```

	wind_deg	wind_speed	day
0	159.5	2.01	2015-06-27 09:42:05
1	100	2.1	2015-06-27 10:37:24
2	80	4.6	2015-06-27 11:57:01
3	90	4.6	2015-06-27 12:53:43
4	80	6.2	2015-06-27 13:54:20
5	80	6.7	2015-06-27 14:55:06
6	90	6.7	2015-06-27 16:55:00
7	90	5.7	2015-06-27 17:55:43
8	90	4.6	2015-06-27 18:58:17
9	97	2.06	2015-06-27 19:58:58
10	89	2.06	2015-06-27 22:52:39
11	88.0147	2.86	2015-06-27 23:57:25
12	107.004	2.01	2015-06-28 00:57:46
13	107.004	2.01	2015-06-28 03:00:34
14	132.503	1.06	2015-06-28 03:54:49
15	132.503	1.06	2015-06-28 04:54:04
16	132.503	1.06	2015-06-28 05:58:15
17	251	1.54	2015-06-28 06:52:59

图9-17 DataFrame中与风有关的数据

为了更好地分析这类数据，有必要将其做成可视化形式，但是对于风力数据，将其制作成使用笛卡儿坐标系的线性图不再是最佳选择。

要是把一个DataFrame中的数据点做成散点图

```
plt.plot(df_ravenna['wind_deg'],df_ravenna['wind_speed'],'ro')
```

就会得到图9-18这样的图表，很显然该图的表现力也有不足。

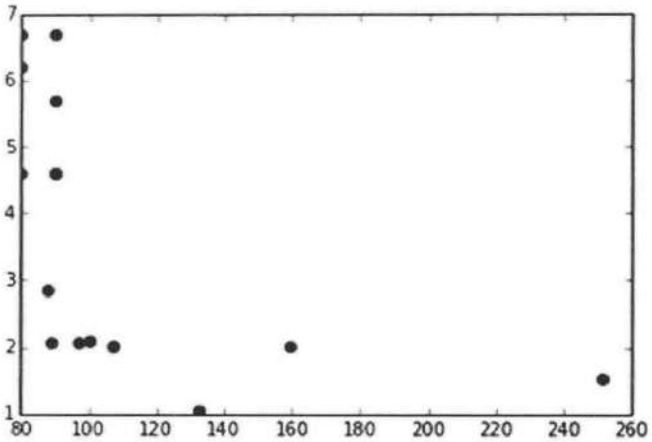


图9-18　用散点图表示呈360度分布的数据点

要表示呈360度分布的数据点，最好使用另一种可视化方法：极区图。我们在第7章讲过这类图。

首先，创建一个直方图，也就是将360度分为八个面元，每个面元为45度，把所有的数据点分到这八个面元中。

```
hist, bins = np.histogram(df_ravenna['wind_deg'],8,[0,360])
print hist
print bins
```

histogram()函数返回结果中的数组hist为落在每个面元的数据点数量。

```
[ 0  5 11  1  0  1  0  0]
```

返回结果中的数组bins定义了360度范围内各面元的边界。

```
[  0.  45.  90.  135.  180.  225.  270.  315.  360.]
```

要想正确定义极区图，离不开这两个数组。我们将创建一个函数来绘制极区图，其中部分代码在第7章已讲过。我们把这个函数定义为showRoseWind()，它有三个参数：values数组，指的是想为其作图的数据，也就是这里的hist数组；第二个参数city_name为字符串类型，指定图表标题所用的城市名称；最后一个参数max_value为整型，指定最大的蓝色值。

定义这样一个函数很有用，它既能避免多次重复编写相同的代码，还能增强代码的模块化程度，便于你把精力放到与函数内部操作相关的概念上。

```
def showRoseWind(values,city_name,max_value):
    N = 8
    theta = np.arange(0.,2 * np.pi, 2 * np.pi / N)
    radii = np.array(values)
    plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)
    colors = [(1-x/max_value, 1-x/max_value, 0.75) for x in radii]
    plt.bar(theta, radii, width=(2*np.pi/N), bottom=0.0, color=colors)
    plt.title(city_name,x=0.2, fontsize=20)
```

你需要修改变量colors存储的颜色表。这里，扇形的颜色越接近蓝色，值越大。

定义好函数之后，调用它即可：

```
showRoseWind(hist,'Ravenna',max(hist))
```

运行上述函数，将得到如图9-19所示的极区图。

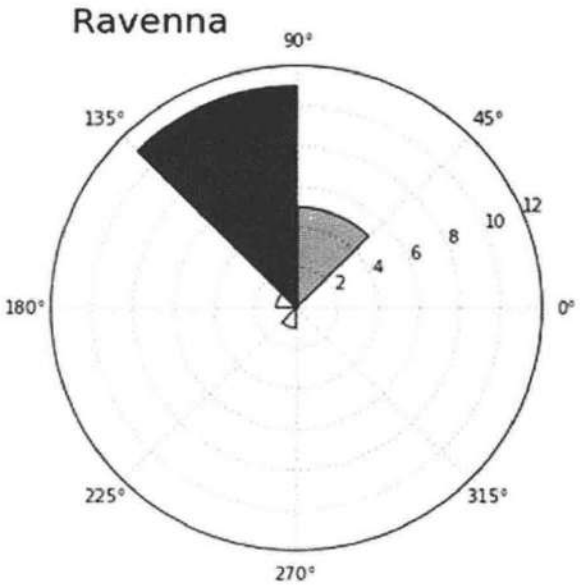


图9-19 极区图能够表示在360度范围内分布的数据点

由图9-19可见，整个360度的范围被分成八个区域（面元），每个区域弧长为45度，此外每个区域还有一列呈放射状排列的刻度值。在每个区域中，用半径长度可以改变的扇形表示一个数值，半径越长，扇形所表示的数值就越大。为了增强图表的可读性，我们使用与扇形半径相对应的颜色表。半径越长，扇形跨度越大，颜色越接近于深蓝色。

从刚得到的极区图可以得知风向在极坐标系中的分布方式。该图表示这一天大部分时间风都吹向西南和正西方向。

定义好showRoseWind()函数之后，查看10个城市的风向情况也非常简单。

```
hist, bin = np.histogram(df_ferrara['wind_deg'],8,[0,360])
print hist
showRoseWind(hist,'Ferrara', 15.0)
```

图9-20为10个城市风向的极区图。

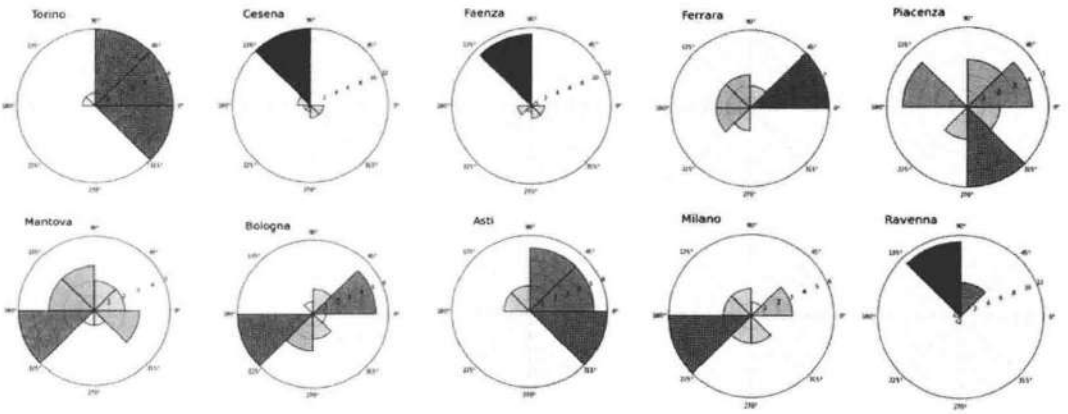


图9-20　表示风向分布的极区图

计算风速均值的分布情况

即使是跟风速相关的其他数据，也可以用极区图来表示。

定义RoseWind_Speed函数，计算将360度范围划分成的八个面元中每个面元的平均风速。

```
def RoseWind_Speed(df_city):
    degs = np.arange(45,361,45)
    tmp = []
    for deg in degs:
        tmp.append(df_city[(df_city['wind_deg']>(deg-46)) & (df_city['wind_deg']<deg)]
        ['wind_speed'].mean())
    return np.array(tmp)
```

该函数返回一个包含八个平均风速值的NumPy数组。该数组将作为先前定义的showRose-Wind()函数的第一个参数，这个函数是用来绘制极区图的。

```
showRoseWind_Speed(RoseWind_Speed(df_ravenna),'Ravenna')
```

图9-21所示的风向频率玫瑰图表示风速在360度范围内的分布情况。

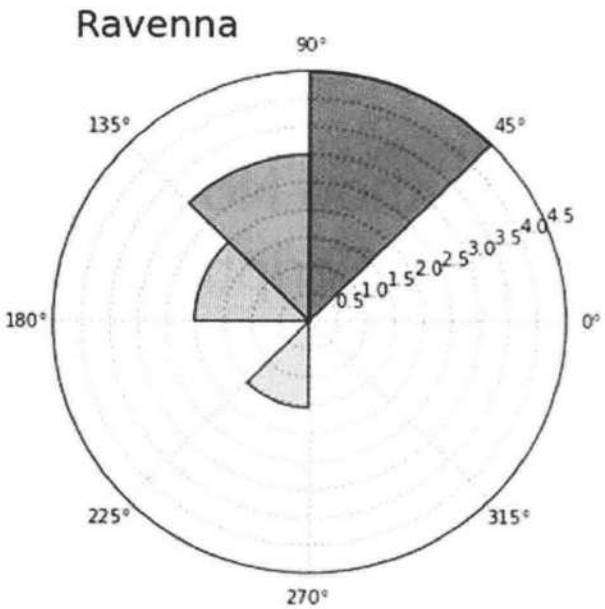


图9-21 表示风速在360度范围内分布情况的极区图

9.5 小结

本章主要目的是演示如何从原始数据获取信息。其中有些信息无法给出重要结论，而有些信息能够验证假设，增加我们对系统状态的认识，而找出这种信息也就意味着数据分析取得了成功。

下一章，你所见到的仍是真实数据。我们从开放数据源获取数据并对其进行分析。你还将学习用JavaScript库D3改善数据可视化效果。即便这个库不是用Python写的，也不用担心，因为把它整合到Python代码中很容易。

第10章 IPython Notebook内嵌JavaScript库D3

这一章将介绍如何通过在IPython Notebook中嵌入JavaScript库D3来扩展其表示图像的能力。该库具有多种图像制作功能，你甚至可以用它实现连matplotlib库无法实现的图像效果。

从这一章的多个例子中，你将学到如何在纯Python环境中实现JavaScript代码。我们将使用整合能力很强的IPython Notebook作为开发环境。你还将学会编写JavaScript代码，以多种可视化方式展示pandas DataFrame中的数据。

10.1 开放的人口数据源

本章数据分析的对象为人口数据集。从Agustin Barto在文章“Embedding Interactive Charts on an IPython Notebook”（在IPython Notebook中嵌入交互式图表）（http://www.machinalis.com/blog/embedding-interactive-charts-on-an-ipython-nb/）所提到的方法入手比较好。这篇文章用到的人口数据来自美国人口调查局网站（http://www.census.gov，见图10-1）。

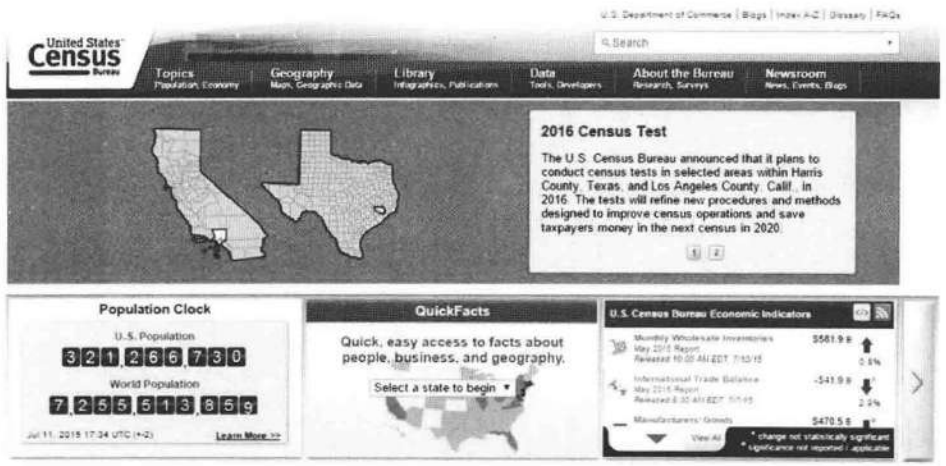


图10-1 美国人口调查局网站首页

美国人口调查局隶属于美国商务部，代表官方采集美国人口数据，并对其做统计研究。该局网站提供了大量CSV格式的数据。前几章曾讲过，将CSV格式的数据导入为pandas的DataFrame形式很容易。

本章我们感兴趣的是美国各州、郡的人口数据。这些数据存储在文件名为CO-EST2014-alldata.csv的CSV文件中。

首先，打开一个IPython Notebook文件，在第一个格子导入所有随后在IPython Notebook中可能会用到的Python库。

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

导入所有必要的库之后，接着从census.gov网站导入数据。我们需要把CO-EST2014-alldata.csv文件直接加载为pandas的DataFrame形式。前几章讲过，你可以在pd.read_csv()函数中用urllib2库指定文件的URL。该函数能够把CSV文件的列表格式数据转换为pandas的DataFrame对象，我们这里将其命名为pop2014。同时，指定dtype选项，把可能解释为数字的字段强制解释为字符串。

```
from urllib2 import urlopen

pop2014 = pd.read_csv(
    urlopen('http://www.census.gov/popest/data/counties/totals/2014/files/CO-EST2014-
    alldata.csv'),
    encoding='latin-1',
    dtype={'STATE': 'str', 'COUNTY': 'str'}
)
```

获取到数据，将其存储到DataFrame对象pop2014，然后输入下述变量名查看它的结构：

```
pop2014
```

输出结果请见图10-2。

仔细分析pop2014中数据的特点，了解该DataFrame对象中各项数据的组织形式。SUMLEV列表示行政区划级别；例如，40表示州，50表示郡。因此，对于SUMLEV列元素为40的行，其人口数据和人口估计数据分别为属于该州SUMLEV列元素为50的各郡的相应数据之和。

REGION、DIVISION、STATE和COUNTY列为美国分属于不同行政区划级别的所有区域。STNAME和CTYNAME分别为州和郡的名称。之后的各列为人口数据。CENSUS2010POP列为实际人口数据，也就是美国2010年的人口普查数据；美国每十年普查一次人口。这之后是每一年的人口估计数。上图列出了2010年的人口数据（2011、2012、2013和2014年数据也在DataFrame中，只是图10-2没有予以显示）。

本章所讲的各个例子，将以可视化形式展示美国人口数据。

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...
0	40	3	6	01	000	Alabama	Alabama	4779736	4780127	4785822	...
1	50	3	6	01	001	Alabama	Autauga County	54571	54571	54684	...
2	50	3	6	01	003	Alabama	Baldwin County	182265	182265	183216	...
3	50	3	6	01	005	Alabama	Barbour County	27457	27457	27336	...
4	50	3	6	01	007	Alabama	Bibb County	22915	22919	22879	...
5	50	3	6	01	009	Alabama	Blount County	57322	57322	57344	...
6	50	3	6	01	011	Alabama	Bullock County	10914	10915	10886	...
7	50	3	6	01	013	Alabama	Butler County	20947	20946	20945	...
8	50	3	6	01	015	Alabama	Calhoun County	118572	118586	118443	...
9	50	3	6	01	017	Alabama	Chambers County	34215	34170	34111	...
10	50	3	6	01	019	Alabama	Cherokee County	25989	25986	25968	...

图10-2　DataFrame对象pop2014包含2010年至2014年所有的人口数据

DataFrame对象pop2014包含大量我们不感兴趣的行或列，所以删除这些用不到的信息，以方便后续操作。首先，我们对每个州的人口数据很感兴趣，因此只抽取SUMLEV列元素为40的行，把它们保存到DataFrame对象pop2014_by_state之中。

```
pop2014_by_state = pop2014[pop2014.SUMLEV == 40]
```

于是，我们得到如图10-3所示的DataFrame对象。

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE
0	40	3	6	01	000	Alabama	Alabama	4779736	4780127	4785822
68	40	4	9	02	000	Alaska	Alaska	710231	710249	713856
98	40	4	8	04	000	Arizona	Arizona	6392017	6392310	6411999
114	40	3	7	05	000	Arkansas	Arkansas	2915918	2915958	2922297
190	40	4	9	06	000	California	California	37253956	37254503	37336011
249	40	4	8	08	000	Colorado	Colorado	5029196	5029324	5048575
314	40	1	1	09	000	Connecticut	Connecticut	3574097	3574096	3579345
323	40	3	5	10	000	Delaware	Delaware	897934	897936	899731
327	40	3	5	11	000	District of Columbia	District of Columbia	601723	601767	605210
329	40	3	5	12	000	Florida	Florida	18801310	18804623	18852220

图10-3　DataFrame对象pop2014_by_state包含与州相关的所有人口数据

然而，我们刚得到的这个DataFrame对象，仍包含很多用不到的列。考虑到列数很多，比起直接用drop()函数删除，仅抽取必要的列更方便。

```
states = pop2014_by_state[['STNAME','POPESTIMATE2011', 'POPESTIMATE2012', 'POPESTIMATE2013',
'POPESTIMATE2014']]
```

既然已获取到必要的信息，我们可以考虑用图形表示这些数据。例如，我们可能想找出美国人口最多的五个州。

```
states.sort(['POPESTIMATE2014'], ascending=False)[:5]
```

我们将得到如图10-4所示的DataFrame，其中各州按人口多寡降序排列。

	STNAME	POPESTIMATE2011	POPESTIMATE2012	POPESTIMATE2013	POPESTIMATE2014
190	California	37701901	38062780	38431393	38802500
2566	Texas	25657477	26094422	26505637	26956958
329	Florida	19107900	19355257	19600311	19893297
1860	New York	19521745	19607140	19695680	19746227
608	Illinois	12858725	12873763	12890552	12880580

图10-4 美国人口最多的五个州

例如，你可以考虑制作条状图，按照降序展示人口最多的五个州。用matplotlib库生成这个图表很简单，但是本章，我们要借助这个小例子，讲解如何用JavaScript库D3在IPython Notebook中实现同样的图表。

10.2 JavaScript 库 D3

JavaScript库D3可用来直接查看和操纵DOM对象（HTML5），但它完全是为数据可视化而开发的，它确实也很擅长这类工作。D3这个名字实际上来自于英文“data-driven documents”（数据驱动文档）三个单词的首字母。D3库全部是由Mike Bostock开发的。

这个库内容丰富，功能强大，因为它以JavaScript、SVG和CSS技术为基础。D3把强大的可视化组件和由数据驱动的DOM操作方法整合在一起，从而充分利用了现代浏览器的功能。

考虑到IPython Notebook同样是Web对象，且它使用的技术也是当代浏览器的基础，因此在Notebook文件中使用这个JavaScript库乍看可能觉得有点荒谬，但其实并非如此。

不熟悉JavaScript库D3以及想详细了解它的读者，我建议读读我写的另一本书：*Create Web Charts with D3*。

IPython Notebook文件确实可以用%%javascript魔术方法把JavaScript代码整合到Python代码中。

但是，跟Python代码类似的是，JavaScript代码也需要导入一些库才能执行。这些库网上就有，每次执行时必须加载它们。在HTML代码中，导入库需要使用下面这种特定的结构：

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min.js"></script>
```

由于这是一对HTML标签，要在IPython Notebook中执行导入操作，就得换成下面这种不同

的结构：

```
%%javascript
require.config({
    paths: {
        d3: '//cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min'
    }
});
```

使用require.config()函数，可以导入所有要用到的JavaScript库。

此外，熟悉HTML代码的话，你一定知道，若要增强HTML页面的表现能力，就得定义CSS样式。同理，你也可以在IPython Notebook中定义一组CSS样式。用IPython.core.display模块的HTML()函数，可以在IPython Notebook中编写HTML代码。CSS样式的正确定义方法为：

```
from IPython.core.display import display, Javascript, HTML

display(HTML("""
<style>

.bar {
   fill: steelblue;
}

.bar:hover{
   fill: brown;
}

.axis {
   font: 10px sans-serif;
}

.axis path,

.axis line {
   fill: none;
   stroke: #000;
}

.x.axis path {
   display: none;
}

</style>
<div id="chart_d3" />
"""))
```

上述代码的最后有一个id为“chart_d3”的HTML标签 ，它指定了D3图形在页面上的显示位置。

现在我们需要编写JavaScript代码，以使用D3库的函数。我们用到了Jinja2库的Template对象，这样就可以定义动态的JavaScript代码，用pandas DataFrame对象的元素替换模板中的变量。

如果系统还没有安装Jinja2库，照旧可以用Anaconda的包管理器安装。

```
conda install jinja2
```

或用pip安装：

```
pip install jinja2
```

安装该库后，就可以定义模板。

```
import jinja2

myTemplate = jinja2.Template("""

require(["d3"], function(d3){

   var data = []

   {% for row in data %}
   data.push({ 'state': '{{ row[1] }}', 'population': {{ row[5] }} });
   {% endfor %}

d3.select("#chart_d3 svg").remove()

   var margin = {top: 20, right: 20, bottom: 30, left: 40},
       width = 800 - margin.left - margin.right,
       height = 400 - margin.top - margin.bottom;

   var x = d3.scale.ordinal()
       .rangeRoundBands([0, width], .25);
   var y = d3.scale.linear()
       .range([height, 0]);

   var xAxis = d3.svg.axis()
       .scale(x)
       .orient("bottom");

   var yAxis = d3.svg.axis()
       .scale(y)
       .orient("left")
       .ticks(10)
       .tickFormat(d3.format('.1s'));

   var svg = d3.select("#chart_d3").append("svg")
       .attr("width", width + margin.left + margin.right)
       .attr("height", height + margin.top + margin.bottom)
       .append("g")
       .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

   x.domain(data.map(function(d) { return d.state; }));
   y.domain([0, d3.max(data, function(d) { return d.population; })]);

   svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + height + ")")
       .call(xAxis);

   svg.append("g")
```

```
        .attr("class", "y axis")
        .call(yAxis)
        .append("text")
        .attr("transform", "rotate(-90)")
        .attr("y", 6)
        .attr("dy", ".71em")
        .style("text-anchor", "end")
        .text("Population");

    svg.selectAll(".bar")
        .data(data)
        .enter().append("rect")
        .attr("class", "bar")
        .attr("x", function(d) { return x(d.state); })
        .attr("width", x.rangeBand())
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });
});
""");
```

但是到这里，我们还没有写完，接下来得把刚刚定义好的D3图表渲染到页面上。我们还需要编写命令，把pandas DataFrame对象中的数据传入模板，以把它们整合到上面所写的JavaScript代码中。运行JavaScript代码，显示图形或者渲染模板，需要调用render()函数。

```
display(Javascript(myTemplate.render(
    data=states.sort(['POPESTIMATE2014'], ascending=False)[:10].itertuples()
)))
```

运行上述代码后，图10-5所示的图表将出现在前面 所在的格子里。该图为2014年人口估计数据位列前五的州。

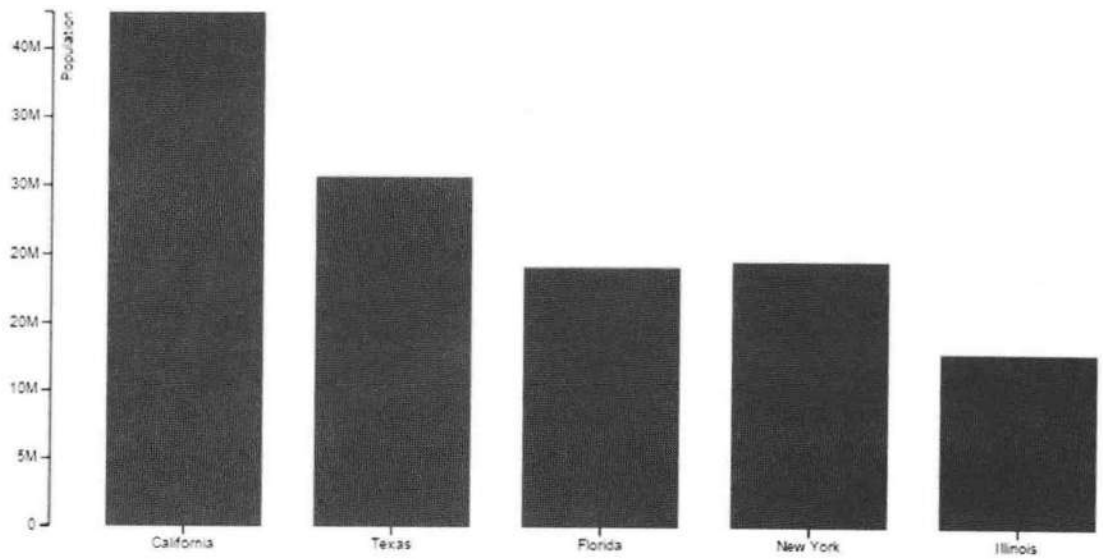


图10-5　用条状图表示2014年美国人口最多的五个州

10.3 绘制簇状条状图

至此，本章所讲的内容大体上参照了Barton那篇非常不错的文章。然而，由于我们所抽取的数据给出了美国各州过去四年的人口估计数，因此描述过去几年各州人口变化趋势的数据可视化方法更有用。

这种应用场景用簇状条状图比较好，人口最多的五个州每个州为一簇，每一簇有四块柱形区域，分别表示每一年的人口数。

可以在前面代码的基础上做修改或者在IPython Notebook中重新编写代码。

```
display(HTML("""
<style>

.bar2011 {
   fill: steelblue;
}

.bar2012 {
   fill: red;
}

.bar2013 {
   fill: yellow;
}

.bar2014 {
   fill: green;
}

.axis {
   font: 10px sans-serif;
}

.axis path,

.axis line {
   fill: none;
   stroke: #000;
}

.x.axis path {
   display: none;
}

</style>
<div id="chart_d3" />
"""))
```

你还需要修改模板，添加其他三组人口数据，以及相对应的2011、2012和2013三个年份。在簇状条状图中，这三个年份的柱形区域分别用不同的颜色来表示。

```
import jinja2


myTemplate = jinja2.Template("""

require(["d3"], function(d3){

   var data = []
   var data2 = []
   var data3 = []
   var data4 = []

   {% for row in data %}
   data.push({ 'state': '{{ row[1] }}', 'population': {{ row[2] }} });
   data2.push({ 'state': '{{ row[1] }}', 'population': {{ row[3] }} });
   data3.push({ 'state': '{{ row[1] }}', 'population': {{ row[4] }} });
   data4.push({ 'state': '{{ row[1] }}', 'population': {{ row[5] }} });
   {% endfor %}

d3.select("#chart_d3 svg").remove()

    var margin = {top: 20, right: 20, bottom: 30, left: 40},
        width = 800 - margin.left - margin.right,
        height = 400 - margin.top - margin.bottom;

    var x = d3.scale.ordinal()
        .rangeRoundBands([0, width], .25);

    var y = d3.scale.linear()
        .range([height, 0]);

    var xAxis = d3.svg.axis()
        .scale(x)
        .orient("bottom");

    var yAxis = d3.svg.axis()
        .scale(y)
        .orient("left")
        .ticks(10)
        .tickFormat(d3.format('.1s'));

    var svg = d3.select("#chart_d3").append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
        .append("g")
        .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

    x.domain(data.map(function(d) { return d.state; }));
    y.domain([0, d3.max(data, function(d) { return d.population; })]);

    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);

    svg.append("g")
```

```
        .attr("class", "y axis")
        .call(yAxis)
        .append("text")
        .attr("transform", "rotate(-90)")
        .attr("y", 6)
        .attr("dy", ".71em")
        .style("text-anchor", "end")
        .text("Population");

    svg.selectAll(".bar2011")
        .data(data)
        .enter().append("rect")
        .attr("class", "bar2011")
        .attr("x", function(d) { return x(d.state); })
        .attr("width", x.rangeBand()/4)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });
    svg.selectAll(".bar2012")
        .data(data2)
        .enter().append("rect")
        .attr("class", "bar2012")
        .attr("x", function(d) { return (x(d.state)+x.rangeBand()/4); })
        .attr("width", x.rangeBand()/4)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });

    svg.selectAll(".bar2013")
        .data(data3)
        .enter().append("rect")
        .attr("class", "bar2013")
        .attr("x", function(d) { return (x(d.state)+2*x.rangeBand()/4); })
        .attr("width", x.rangeBand()/4)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });

    svg.selectAll(".bar2014")
        .data(data4)
        .enter().append("rect")
        .attr("class", "bar2014")
        .attr("x", function(d) { return (x(d.state)+3*x.rangeBand()/4); })
        .attr("width", x.rangeBand()/4)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });
});
""");
```

我们这次从DataFrame取四个序列的数据传递给模板，因此需要更新数据，重新渲染。此外，还需要加上render()函数。

```
display(Javascript(myTemplate.render(
    data=states.sort(['POPESTIMATE2014'], ascending=False)[:5].itertuples()
)))
```

调用render()函数之后，将得到如图10-6所示的条状图。

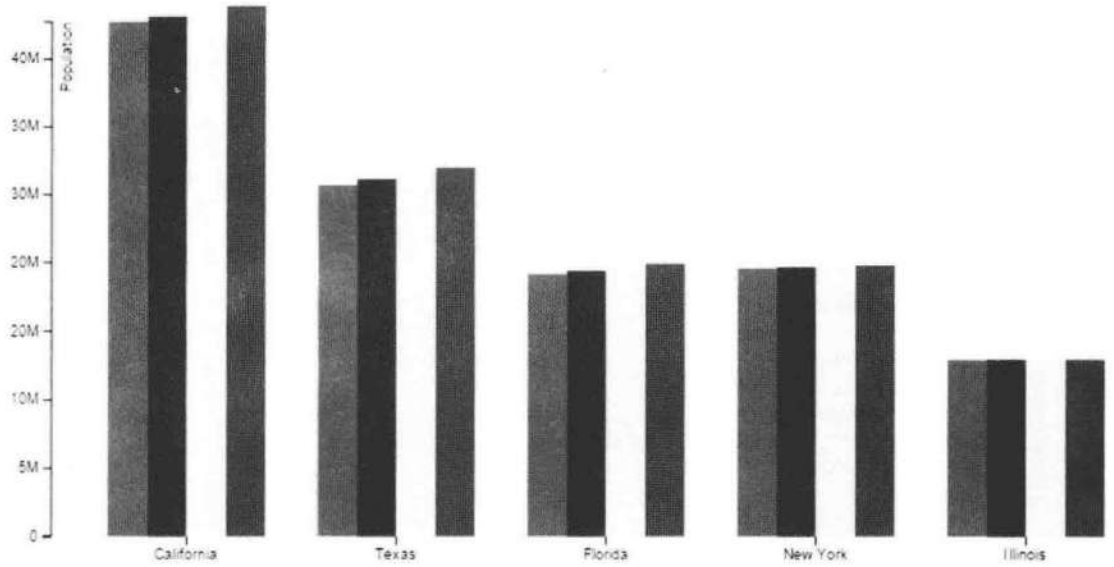


图10-6　2011到2014年间美国人口最多的五个州的簇状条状图

10.4　地区分布图

从前面几节，你了解了如何使用JavaScript代码和D3库绘制条状图。这点成果其实用matplotlib库很容易就能实现，甚至效果更好。其实，前面这些代码只是为了让你对D3有个大致的印象。

跟matplotlib库很不同的是，D3能够实现matplotlib实现不了的更为复杂的图形。因此现在我们就来尝试D3库的强大功能。它可以实现地区分布图这类非常复杂的图形。

地区分布图表示对象的地区分布情况，其中地区分成用不同颜色表示的多个部分。两块区域用不同颜色和边界进行区分，颜色和边界所表示的其实就是数据。

这种表示方法适用于人口或经济信息的数据分析结果，也适用于其他跟地理分布相关的数据。

地区分布图以JSON TopoJSON这种特殊文件为基础。该种文件包含制作诸如美国各地区这样的地区分布图所需的全部信息（见图10-7）。

US Atlas TopoJSON这个链接（https://github.com/mbostock/us-atlas）提供生成各种TopoJSON文件的相关资料，此外其他很多网站也提供类似的内容。

有了D3库，这种图像表示法不仅可以实现，甚至还可以进行个性化处理。我们可以根据DataFrame几列元素的值，为不同地理区域涂上不同颜色。

首先，我们从网上已有的例子入手，该例子在D3库http://bl.ocks.org/mbostock/4060606之中，但它全部是用HTML开发的。因此你需要学习如何把用HTML开发的D3例子移植到IPython Notebook之中。

图10-7 地区（美国领土）分布图，各郡或州无数据

查看讲解例子的网页，就能看到它引入了几个必要的JavaScript库。这一次，除了D3库，还需要导入queue和TopoJSON库。

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/queue-async/1.0.7/queue.min.js">
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/topojson/1.6.19/topojson.min.js">
</script>
```

因此你得用前几节定义的require.config()函数。

```
%%javascript
require.config({
    paths: {
        d3: '//cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min',
        queue: '//cdnjs.cloudflare.com/ajax/libs/queue-async/1.0.7/queue.min',
        topojson: '//cdnjs.cloudflare.com/ajax/libs/topojson/1.6.19/topojson.min'
    }
});
```

至于CSS部分，仍需将其全部写到HTML()函数之中。

```
from IPython.core.display import display, Javascript, HTML

display(HTML("""
<style>

.counties {
fill: none;
}
```

```
.states {
  fill: none;
  stroke: #fff;
  stroke-linejoin: round;
}

.q0-9 { fill:rgb(247,251,255); }
.q1-9 { fill:rgb(222,235,247); }
.q2-9 { fill:rgb(198,219,239); }
.q3-9 { fill:rgb(158,202,225); }
.q4-9 { fill:rgb(107,174,214); }
.q5-9 { fill:rgb(66,146,198); }
.q6-9 { fill:rgb(33,113,181); }
.q7-9 { fill:rgb(8,81,156); }
.q8-9 { fill:rgb(8,48,107); }

</style>
<div id="choropleth" />
"""))
```

下面是仿照Bostock给出的示例代码所写的模板，只不过做了些改动。

```
import jinja2

choropleth = jinja2.Template("""

require(["d3","queue","topojson"], function(d3,queue,topojson){

//   var data = []
//   {% for row in data %}
//   data.push({ 'state': '{{ row[1] }}', 'population': {{ row[2] }} });
//   {% endfor %}

d3.select("#choropleth svg").remove()

var width = 960,
    height = 600;

var rateById = d3.map();

ar quantize = d3.scale.quantize()
    .domain([0, .15])
    .range(d3.range(9).map(function(i) { return "q" + i + "-9"; }));

var projection = d3.geo.albersUsa()
    .scale(1280)
    .translate([width / 2, height / 2]);

var path = d3.geo.path()
    .projection(projection);

//row to modify
var svg = d3.select("#choropleth").append("svg")
    .attr("width", width)
```

```
    .attr("height", height);

queue()
    .defer(d3.json, "us.json")
    .defer(d3.tsv, "unemployment.tsv", function(d) { rateById.set(d.id, +d.rate); })
    .await(ready);

function ready(error, us) {
  if (error) throw error;

  svg.append("g")
      .attr("class", "counties")
    .selectAll("path")
      .data(topojson.feature(us, us.objects.counties).features)
    .enter().append("path")
      .attr("class", function(d) { return quantize(rateById.get(d.id)); })
      .attr("d", path);

  svg.append("path")
      .datum(topojson.mesh(us, us.objects.states, function(a, b) { return a !== b; }))
      .attr("class", "states")
      .attr("d", path);

}
});
""");
```

现在我们来渲染模板，这次不需要为模板指定数据，因为所有的数据都存储在JSON和TSV文件里。

```
display(Javascript(choropleth.render()))
```

结果跟Bostcok例子所示的结果相同（见图10-8）。

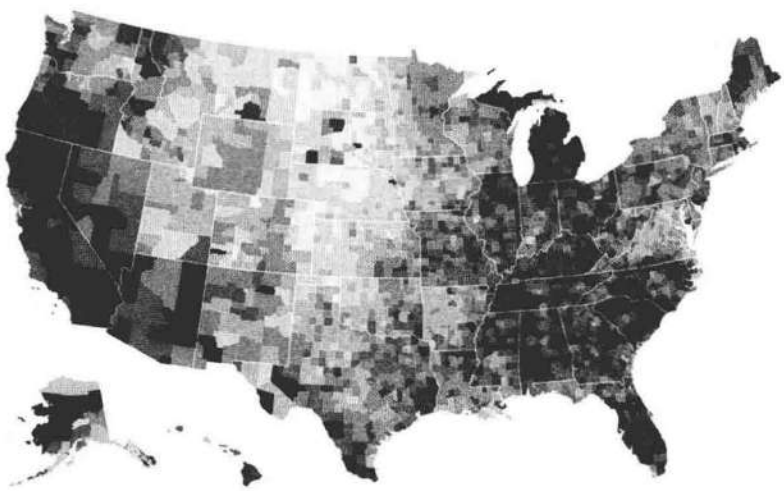


图10-8 美国人口地区分布图，根据TSV文件中的数据为各郡涂上不同颜色

10.5　2014 年美国人口地区分布图

既已了解如何从美国人口调查局网站抽取人口信息，并学会了制作地区分布图，你可以把这些知识结合在一起，做一幅地区分布图，用深浅不同的颜色表示人口数量。郡人口越多，蓝色越深；人口越少，色彩越趋向于白色。

本章第一节，我们从pop2014之中抽取了各州的人口数据。我们只选择了SUMLEV列元素为40的那些行。接下来这个例子中，我们要用到各郡的人口数据，因此只从pop2014中抽取SUMLEV列元素为50的行。

用下述代码选择行政区划级别为50的郡。

```
pop2014_by_county = pop2014[pop2014.SUMLEV == 50]
pop2014_by_county
```

我们得到了包含美国各郡信息的DataFrame，如图10-9所示。

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...
1	50	3	6	01	001	Alabama	Autauga County	54571	54571	54684	...
2	50	3	6	01	003	Alabama	Baldwin County	182265	182265	183216	...
3	50	3	6	01	005	Alabama	Barbour County	27457	27457	27336	...
4	50	3	6	01	007	Alabama	Bibb County	22915	22919	22879	...
5	50	3	6	01	009	Alabama	Blount County	57322	57322	57344	...
6	50	3	6	01	011	Alabama	Bullock County	10914	10915	10886	...
7	50	3	6	01	013	Alabama	Butler County	20947	20946	20945	...
8	50	3	6	01	015	Alabama	Calhoun County	118572	118586	118443	...
9	50	3	6	01	017	Alabama	Chambers County	34215	34170	34111	...
10	50	3	6	01	019	Alabama	Cherokee County	25989	25986	25968	...

图10-9　DataFrame对象pop2014_by_county包含美国各郡人口数据

我们必须使用刚得到的数据而不是前面TSV之中的数据。pop2014_by_county对象之中，有一列ID代码对应各郡。网上有ID代码和郡名称的对应表，有了这份文件就可以知道ID代码代表哪个郡；下载该文件，将其转换为DataFrame对象。

```
USJSONnames = pd.read_table(urlopen('http://bl.ocks.org/mbostock/raw/4090846/us-countynames.
tsv'))
USJSONnames
```

如前所述，该文件列出了ID代码对应的郡名（见图10-10）。

	id	name
0	1000	Alabama
1	1001	Autauga
2	1003	Baldwin
3	1005	Barbour
4	1007	Bibb
5	1009	Blount
6	1011	Bullock
7	1013	Butler
8	1015	Calhoun

图10-10 TSV文件的ID为各郡的代码

例如，我们来看一下Baldwin郡：

```
USJSONnames[USJSONnames['name'] == 'Baldwin']
```

你会发现有两个名为Baldwin的郡，但它们的郡代码不同（见图10-11）。

	id	name
2	1003	Baldwin
399	13009	Baldwin

图10-11 有两个名为Baldwin的郡

上表中，显示有两个郡代码不同、名称相同的郡。前面我们抽取census.gov的数据后，将其保存到一个DataFrame对象中，我们现在就来看一下该DataFrame对象都包含这两个郡的哪些信息（见图10-12）。

```
pop2014_by_county[pop2014_by_county['CTYNAME'] == 'Baldwin County']
```

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...	RINTI
2	50	3	6	01	003	Alabama	Baldwin County	182265	182265	183216	...	1.471
402	50	3	5	13	009	Georgia	Baldwin County	45720	45835	45685	...	0.971

2 rows × 84 columns

图10-12 STATE和COUNTY列的两个元素结合起来恰好是TSV文件的ID代码

我们能找到代码的对应关系。TOPOJSON中的ID代码恰好对应STATE和COUNTY列的两个元素，当STATE列的第一位为0时，要把0删除。现在可以用counties对象重新创建重现TSV那个例子所需的数据。记得把数据保存到population.csv文件中。

```
counties = pop2014_by_county[['STATE','COUNTY','POPESTIMATE2014']]
counties.is_copy = False
counties['id'] = counties['STATE'].str.lstrip('0') + "" + counties['COUNTY']
del counties['STATE']
del counties['COUNTY']
counties.columns = ['pop','id']
counties = counties[['id','pop']]
counties.to_csv('population.csv')
```

我们再次改写HTML()函数，新增一个 标签，指定其id为choropleth2。

```
from IPython.core.display import display, Javascript, HTML

display(HTML("""
<style>

.counties {
  fill: none;
}

.states {
  fill: none;
  stroke: #fff;
  stroke-linejoin: round;
}

.q0-9 { fill:rgb(247,251,255); }
.q1-9 { fill:rgb(222,235,247); }
.q2-9 { fill:rgb(198,219,239); }
.q3-9 { fill:rgb(158,202,225); }
.q4-9 { fill:rgb(107,174,214); }
.q5-9 { fill:rgb(66,146,198); }
.q6-9 { fill:rgb(33,113,181); }
.q7-9 { fill:rgb(8,81,156); }
.q8-9 { fill:rgb(8,48,107); }

</style>
<div id="choropleth2" />
"""))
```

最后，还需要定义一个新Template对象。

```
choropleth2 = jinja2.Template("""

require(["d3","queue","topojson"], function(d3,queue,topojson){

   var data = []

d3.select("#choropleth2 svg").remove()

var width = 960,
    height = 600;

var rateById = d3.map();
```

```
var quantize = d3.scale.quantize()
    .domain([0, 1000000])
    .range(d3.range(9).map(function(i) { return "q" + i + "-9"; }));

var projection = d3.geo.albersUsa()
    .scale(1280)
    .translate([width / 2, height / 2]);

var path = d3.geo.path()
    .projection(projection);

var svg = d3.select("#choropleth2").append("svg")
    .attr("width", width)
    .attr("height", height);

queue()
    .defer(d3.json, "us.json")
    .defer(d3.csv,"population.csv", function(d) { rateById.set(d.id, +d.pop); })
    .await(ready);

function ready(error, us) {
  if (error) throw error;

  svg.append("g")
      .attr("class", "counties")
    .selectAll("path")
      .data(topojson.feature(us, us.objects.counties).features)
    .enter().append("path")
      .attr("class", function(d) { return quantize(rateById.get(d.id)); })
      .attr("d", path);

  svg.append("path")
      .datum(topojson.mesh(us, us.objects.states, function(a, b) { return a !== b; }))
      .attr("class", "states")
      .attr("d", path);
}

});

""");
```

执行render()函数，生成图表。

```
display(Javascript(choropleth2.render()))
```

我们生成的地区分布图中，根据各郡人口多寡，不同区域使用深浅不一的颜色，请见图10-13。

图10-13　美国各郡人口密度地区分布图

10.6　小结

本章展示了用JavaScript库D3能够进一步扩展数据表示能力。多种高级图表都可用来表示数据，我们所讲的地区分布图只是其中一种。该例也说明IPython Notebook（Jupyter）能整合多种技术。换句话说，世界并不仅以Python为中心，但是Python允许整合更多功能以帮助我们完成工作。

下一章，也就是本书的最后一章，将看一下如何对图像进行数据分析。你将发现，建立能够识别手写体数字的模型其实很简单。

第11章 识别手写体数字

前面，我们讲解了如何对存有数字或字符串的pandas DataFrame对象做数据分析。实际上，数据分析并不仅限于此，我们还可以分析图像和声音，对它们进行分类。

这一章虽篇幅短小，但重要性丝毫不逊于前。我们将讲解手写体文本识别，特别是数字的识别。

11.1 手写体识别

手写体文本识别问题可以追溯到第一代从手写体文档中识别单个字符的自动化机器。例如，你可以想象这样一个情形：邮局里信件堆积如山，因此需要借助自动化手段识别五位邮政编码，而只有正确识别，才能实现自动化和高效地分拣邮件。面对该应用场景，你可能想到多种应用，其中也许会有OCR（Optical Character Recognition，光学字符识别）软件，它读入手写体或印刷体文本，识别其中的文字后，生成常用的电子文档。

手写体识别问题还可以再向前追溯，更准确地说可以追溯到20世纪20年代，即Emanuel Goldberg（1881~1970）开始着手研究这个问题的时候。他当时提出了统计方法可能是最佳的选择。

scikit-learn库提供了一个很棒的例子，便于我们更好地理解手写体识别技术、相关问题以及用机器学习方法识别文本的可能性。

11.2 用 scikit-learn 识别手写体数字

用scikit-learn库（http://scikit-learn.org/）分析这类数据所用的方法，跟本书前面一直在用的多少有点不同。待分析数据不仅涉及数值或字符串类型的处理，还涉及图像或声音文件的处理。

因而，你可以将本章要面对的问题看作读取和解释手写体数字图像，预测图像之中的数值。

这类数据分析问题，需要用到*估计器*（estimator）。它借助`fit()`函数进行学习，待自己的预测能力（模型足够有效）达到一定水准之后，再用`predict()`函数给出预测结果。拿到结果之后，我们还会讨论训练集和验证集。与之前不同，这两个数据集是由一系列图像组成的。

输入以下代码，从命令行运行IPython Notebook：

```
ipython notebook
```

接着依次点击New、Python 2菜单项，新建一个Notebook文件，如图11-1所示。

图11-1　IPython Notebook（Jupyter）首页

这个例子可以用sklearn.svm.SVC估计器，它使用的是SVC技术。

因此，你需要导入scikit-learn的svm模块。接着，创建SVC类型的估计器，并初始化设置。无需为C和gamma选项设置特殊值，使用一般值即可，分析过程中可再调整。

```
from sklearn import svm
svc = svm.SVC(gamma=0.001, C=100.)
```

11.3　Digits 数据集

在第8章提过，scikit-learn库提供了大量数据集，可用于测试数据分析相关的问题和结果预测问题。其中，对于这里要讲的手写体识别问题，我们可以使用它的Digits图像数据集。

该数据集包含1797张8×8像素大小的灰度图，图像的内容为一个手写体数字（见图11-2）。

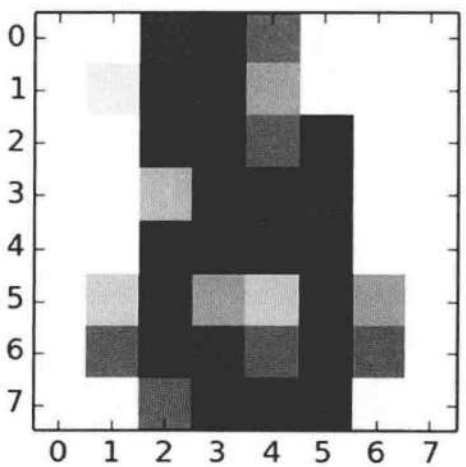


图11-2　Digits数据集1797张手写体数字图像中的一张

在Notebook中，导入Digits数据集。

```
from sklearn import datasets
digits = datasets.load_digits()
```

加载数据集后，对它里面的内容略作分析。首先，访问DESCR属性，读取数据集自带的大量说明信息。

```
print digits.DESCR
```

上述命令将输出数据集的简介、作者以及参考资料，详见图11-3。

```
print digits.DESCR

 Optical Recognition of Handwritten Digits Data Set

Notes
-----
Data Set Characteristics:
    :Number of Instances: 5620
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets
http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where
each class refers to a digit.

Preprocessing programs made available by NIST were used to extract
normalized bitmaps of handwritten digits from a preprinted form. From a
total of 43 people, 30 contributed to the training set and different 13
to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of
4x4 and the number of on pixels are counted in each block. This generates
an input matrix of 8x8 where each element is an integer in the range
0..16. This reduces dimensionality and gives invariance to small
distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.
T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.
L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,
1994.
```

图11-3 scikit-learn库的每个数据集都有一个存储说明信息的字段

手写体数字图像的数据，则存储在digits.images数组中。数组的每个元素表示一张图像，每个元素为8×8形状的矩阵，矩阵各项为数值类型，每个数值对应一种灰度等级，其中0对应白

色，15对应黑色。

```
digits.images[0]
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.],
       [ 0.,  0., 13., 15., 10., 15.,  5.,  0.],
       [ 0.,  3., 15.,  2.,  0., 11.,  8.,  0.],
       [ 0.,  4., 12.,  0.,  0.,  8.,  8.,  0.],
       [ 0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
       [ 0.,  4., 11.,  0.,  1., 12.,  7.,  0.],
       [ 0.,  2., 14.,  5., 10., 12.,  0.,  0.],
       [ 0.,  0.,  6., 13., 10.,  0.,  0.,  0.]])
```

可以借助matplotlib库为数组元素生成图像，这样看起来更直观。

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(digits.images[0], cmap=plt.cm.gray_r, interpolation='nearest')
```

运行上述命令，将得到如图11-4所示的灰度图。

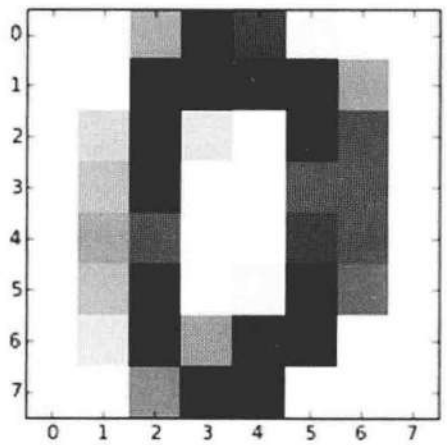


图11-4　1797个手写体数字之一

图像所表示的数字，也就是目标值，则存储在digits.targets数组之中。

```
digits.target
array([0, 1, 2, ..., 8, 9, 8])
```

该数据集据说有1797张图像，我们可以确认是否果真如此。

```
digits.target.size
1797
```

11.4　学习和预测

既已在IPython Notebook中加载了Digits数据集，且已定义好一个SVC估计器，估计器的学习步骤可就此开始。

在第8章讲过，定义好预测模型之后，必须用已知各条数据类别的训练集调教它。考虑到Digits数据集的数据量很大，用它进行训练得到的模型效果肯定非常好。也就是说，模型识别手写体数字准确率高。

Digits数据集由1797个元素组成，可以考虑用前1791个作为训练集，用剩余6个作为验证集。我们可再次用matplotlib生成这6个手写体数字的图像，以便查看其细节。

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.subplot(321)
plt.imshow(digits.images[1791], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(322)
plt.imshow(digits.images[1792], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(323)
plt.imshow(digits.images[1793], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(324)
plt.imshow(digits.images[1794], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(325)
plt.imshow(digits.images[1795], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(326)
plt.imshow(digits.images[1796], cmap=plt.cm.gray_r, interpolation='nearest')
```

上述代码将生成6个数字的图像，见图11-5。

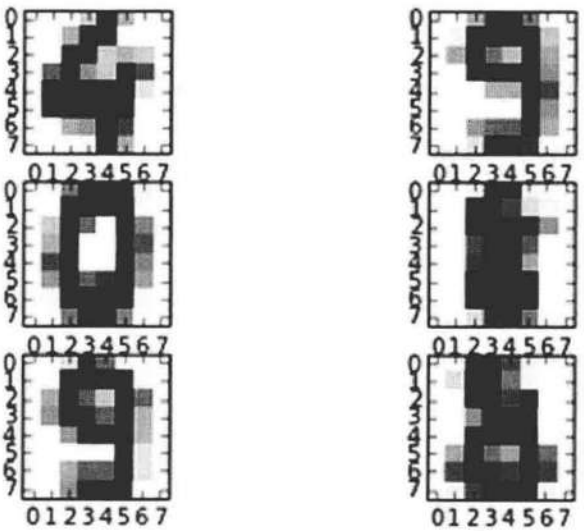


图11-5 验证集6个数字的图像

现在，你可以让先前定义的svc估计器进行学习。

```
svc.fit(digits.data[1:1790], digits.target[1:1790])
```

上述命令运行一小段时间后，输出训练得到的估计器。

```
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
  random_state=None, shrinking=True, tol=0.001, verbose=False)
```

接着，用估计器预测验证集的6个数字，以测试估计器的效果。

```
svc.predict(digits.data[1791:1976])
```

得到的结果如下：

```
array([4, 9, 0, 8, 9, 8])
```

与验证集各图像实际表示的数字相比：

```
digits.target[1791:1976]
```

```
array([4, 9, 0, 8, 9, 8])
```

你会发现svc估计器能够正确学习，识别手写体数字。对于验证集的6个数字，它全部预测正确。

11.5　小结

本章内容虽简短，但你也能从中领会到这种数据分析方法应用的机会其实很多。它不仅可以分析数值或文本数据，还可以用来分析图像，甚至是由相机或扫描仪生成的手写体图像。

此外，你还见识了用机器学习方法创建的预测模型能够给出完美的预测结果，而借助scikit-learn库，实现机器学习方法也较为简单。

附录 A

用LaTeX编写数学表达式

Python世界大量使用LaTeX。本附录给出了很多实用的例子，介绍在Python应用中如何使用LaTeX表达式。同样的内容也可以从matplotlib官网的相关页面http://matplotlib.org/users/mathtext.html找到。

A.1 matplotlib

若函数能接收LaTeX表达式，就可以直接以参数的形式将表达式传入。例如，绘制图表标题的title()函数。

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.title(r'$\alpha > \beta$')
```

A.2 IPython Notebook 文件 Markdown 格子

将LaTeX表达式置于两个$$符号之间。

```
$$c = \sqrt{a^2 + b^2}$$
```

$$c = \sqrt{a^2 + b^2}$$

A.3 IPython Notebook 文件 Python 2 格子

在Math()函数中输入LaTeX表达式。

```
from IPython.display import display, Math, Latex
display(Math(r'F(k) = \int_{-\infty}^{\infty} f(x) e^{2\pi i k} dx'))
```

A.4 下标和上标

用_和^符号定义下标和上标格式。

```
r'$\alpha_i > \beta_i$'
```

$\alpha_i > \beta_i$

编写求和表达式时，上下标定义方法的威力立马显现出来。

```
r'$\sum_{i=0}^\infty x_i$'
```

$$\sum_{i=0}^{\infty} x_i$$

A.5　分数、二项式和数字堆叠

分数、二项式和数字堆叠可分别用\frac{}{}、\binom{}{}和\stackrel{}{}命令实现：

```
r'$\frac{3}{4} \binom{3}{4} \stackrel{3}{4}$'
```

$$\frac{3}{4}\binom{3}{4}^{3}$$

分数可任意嵌套[①]：

$$\frac{5-\frac{1}{x}}{4}$$

分数外要加小括号和方括号时，需特别注意。括号前要分别加上\left和\right，以告知解析器括号是将整个对象包裹在里面[②]：

$$\left(\frac{5-\frac{1}{x}}{4}\right)$$

A.6　根数

用\sqrt[]{}命令生成根数。

```
r'$\sqrt{2}$'
```

$$\sqrt{2}$$

A.7　字体

数学符号默认使用斜体。如要改变字体，比如三角函数*sin*的字体：

① 下面的分数对应的LaTeX表达式为r'$\frac{5-\frac{1}{x}}{4} $'。

② 下面的表达式对应的LaTeX表达式为r'$\left[\frac{5-\frac{1}{x}}{4} \right]$'。

$s\ (t) = Asin(2wt\ )$

可用的字体有[①]：

```
from IPython.display import display, Math, Latex
display(Math(r'\mathrm{Roman}'))
display(Math(r'\mathit{Italic}'))
display(Math(r'\mathtt{Typewriter}'))
display(Math(r'\mathcal{CALLIGRAPHY}'))
```

Roman

Italic

$\mathtt{Typewriter}$

$\mathcal{CALLIGRAPHY}$

A.8　强调符号

强调命令置于任何符号前，可实现在符号上添加表示强调标识的效果，其中有些标识有长短之分。

\acute a或\'a[②]	á
\bar a	ā
\breve a	ă
\ddot a或 \"a	ä
\dot a或 \.a	ȧ
\grave a或 \ ` a	à
\hat a或 \^a	â
\tilde a或 \~a	ã
\vec a	$\vec{a}$
\overline{abc}	$\overline{abc}$

符号

还可以使用大量TeX符号。

① 把三角函数表达式放入字体函数的公式中即可，如r'\mathtt{s(t)=Asin(2wt)}'。

② 在Python语句中使用“\'a”时，要注意对斜线进行转义。

小写希腊字母

α \alpha	β \beta	χ \chi	δ \delta	F \digamma
ϵ \epsilon	η \eta	γ \gamma	ι \iota	κ \kappa
λ \lambda	μ \mu	ν \nu	ω \omega	ϕ \phi
π \pi	ψ \psi	ρ \rho	σ \sigma	τ \tau
θ \theta	υ \upsilon	ε \varepsilon	$\varkappa$ \varkappa	φ \varphi
ϖ \varpi	ϱ \varrho	ς \varsigma	ϑ \vartheta	ξ \xi
ζ \zeta				

大写希腊字母

Δ \Delta	Γ \Gamma	Λ \Lambda	Ω \Omega	Φ \Phi	Π \Pi
Ψ \Psi	Σ \Sigma	Θ \Theta	Υ \Upsilon	Ξ \Xi	$\mho$ \mho
∇ \nabla					

希伯来语字母

$\aleph$ \aleph	$\beth$ \beth	$\daleth$ \daleth	$\gimel$ \gimel

定界符

$/$ /	$[$ [	$\Downarrow$ \Downarrow	$\Uparrow$ \Uparrow	$\Vert$ \Vert	$\backslash$ \backslash
$\downarrow$ \downarrow	$\langle$ \langle	$\lceil$ \lceil	$\lfloor$ \lfloor	$\llcorner$ \llcorner	$\lrcorner$ \lrcorner
$\rangle$ \rangle	$\rceil$ \rceil	$\rfloor$ \rfloor	$\ulcorner$ \ulcorner	$\uparrow$ \uparrow	$\urcorner$ \urcorner
$\vert$ \vert	$\{$ \{	$\Vert$ \Vert	$\}$ \}	$]$]	$\vert$ \vert

大符号

$\bigcap$ \bigcap	$\bigcup$ \bigcup	$\bigodot$ \bigodot	$\bigoplus$ \bigoplus	$\bigotimes$ \bigotimes
$\biguplus$ \biguplus	$\bigvee$ \bigvee	$\bigwedge$ \bigwedge	$\coprod$ \coprod	$\int$ \int
$\oint$ \oint	$\prod$ \prod	$\sum$ \sum		

标准函数名

$\Pr$ \Pr	$\arccos$ \arccos	$\arcsin$ \arcsin	$\arctan$ \arctan
$\arg$ \arg	$\cos$ \cos	$\cosh$ \cosh	$\cot$ \cot
$\coth$ \coth	$\csc$ \csc	$\deg$ \deg	$\det$ \det
$\dim$ \dim	$\exp$ \exp	$\gcd$ \gcd	$\hom$ \hom
$\inf$ \inf	$\ker$ \ker	$\lg$ \lg	$\lim$ \lim
$\liminf$ \liminf	$\limsup$ \limsup	$\ln$ \ln	$\log$ \log
$\max$ \max	$\min$ \min	$\sec$ \sec	$\sin$ \sin
$\sinh$ \sinh	$\sup$ \sup	$\tan$ \tan	$\tanh$ \tanh

二元运算和关系符号

$\Bumpeq$ \Bumpeq	$\Cap$ \Cap	$\Cup$ \Cup
$\Doteq$ \Doteq	$\bowtie$ \Join	$\Subset$ \Subset
$\Supset$ \Supset	$\Vdash$ \Vdash	$\Vvdash$ \Vvdash
$\approx$ \approx	$\approxeq$ \approxeq	$\ast$ \ast
$\asymp$ \asymp	$\backepsilon$ \backepsilon	$\backsim$ \backsim
$\backsimeq$ \backsimeq	$\barwedge$ \barwedge	$\because$ \because
$\between$ \between	$\bigcirc$ \bigcirc	$\bigtriangledown$ \bigtriangledown

（续）

△\bigtriangleup	◀\blacktriangleleft	▶\blacktriangleright
⊥\bot	⋈\bowtie	⊡\boxdot
⊟\boxminus	⊞\boxplus	⊠\boxtimes
•\bullet	≏\bumpeq	∩\cap
·\cdot	∘\circ	≗\circeq
≔\coloneq	≅\cong	∪\cup
⋞\curlyeqprec	⋟\curlyeqsucc	⋎\curlyvee
⋏\curlywedge	†\dag	⊣\dashv
‡\ddag	◇\diamond	÷\div
⋇\divideontimes	$\dot{e}q$\doteq	$\dot{e}q\dot{d}ot$\doteqdot
$\dot{p}lus$\dotplus	⩛\doublebarwedge	⧈\eqcirc
=:\eqcolon	≂\eqsim	⪖\eqslantgtr
⋜\eqslantless	≡\equiv	⩽̇\fallingdotseq
⌢\frown	≥\geq	⧧\geqq
⩾\geqslant	≫\gg	⋙\ggg
⪊\gnapprox	⪈\gneqq	⋧\gnsim
⪆\gtrapprox	⋗\gtrdot	⩼\gtreqless
⪌\gtreqqless	≷\gtrless	≳\gtrsim
∈\in	⊺\intercal	⋋\leftthreetimes
≤\leq	≦\leqq	⩽\leqslant
⪅\lessapprox	⋖\lessdot	⋚\lesseqgtr
⪋\lesseqqgtr	≶\lessgtr	≲\lesssim
≪\ll	⋘\lll	⪉\lnapprox
⪇\lneqq	⋦\lnsim	⋉\ltimes

（续）

$\mid$ \mid	$\models$ \models	$\mp$ \mp
$\nVDash$ \nVDash	$\nVdash$ \nVdash	$\napprox$ \napprox
$\ncong$ \ncong	$\ne$ \ne	$\neq$ \neq
$\neq$ \neq	$\nequiv$ \nequiv	$\ngeq$ \ngeq
$\ngtr$ \ngtr	$\ni$ \ni	$\nleq$ \nleq
$\nless$ \nless	$\nmid$ \nmid	$\notin$ \notin
$\nparallel$ \nparallel	$\nprec$ \nprec	$\nsim$ \nsim
$\nsubset$ \nsubset	$\nsubseteq$ \nsubseteq	$\nsucc$ \nsucc
$\nsupset$ \nsupset	$\nsupseteq$ \nsupseteq	$\ntriangleleft$ \ntriangleleft
$\ntrianglelefteq$ \ntrianglelefteq	$\ntriangleright$ \ntriangleright	$\ntrianglerighteq$ \ntrianglerighteq
$\nvDash$ \nvDash	$\nvdash$ \nvdash	$\odot$ \odot
$\ominus$ \ominus	$\oplus$ \oplus	$\oslash$ \oslash
$\otimes$ \otimes	$\parallel$ \parallel	$\perp$ \perp
$\pitchfork$ \pitchfork	$\pm$ \pm	$\prec$ \prec
$\precapprox$ \precapprox	$\preccurlyeq$ \preccurlyeq	$\preceq$ \preceq
$\precnapprox$ \precnapprox	$\precnsim$ \precnsim	$\precsim$ \precsim
$\propto$ \propto	$\rightthreetimes$ \rightthreetimes	$\risingdotseq$ \risingdotseq
$\rtimes$ \rtimes	$\sim$ \sim	$\simeq$ \simeq
$/$ \slash	$\smile$ \smile	$\sqcap$ \sqcap
$\sqcup$ \sqcup	$\sqsubset$ \sqsubset	$\sqsubset$ \sqsubset
$\sqsubseteq$ \sqsubseteq	$\sqsupset$ \sqsupset	$\sqsupset$ \sqsupset
$\sqsupseteq$ \sqsupseteq	$\star$ \star	$\subset$ \subset
$\subseteq$ \subseteq	$\subseteqq$ \subseteqq	$\subsetneq$ \subsetneq
$\subsetneqq$ \subsetneqq	$\succ$ \succ	$\succapprox$ \succapprox
$\succcurlyeq$ \succcurlyeq	$\succeq$ \succeq	$\succnapprox$ \succnapprox
$\succnsim$ \succnsim	$\succsim$ \succsim	$\supset$ \supset

（续）

⊇ \supseteq	⫆ \supseteqq	⊉ \supsetneq
⫌ \supsetneqq	∴ \therefore	× \times
⊤ \top	◁ \triangleleft	⊴ \trianglelefteq
≜ \triangleq	▷ \triangleright	⊵ \trianglerighteq
⊎ \uplus	⊨ \vDash	∝ \varpropto
⊲ \vartriangleleft	⊳ \vartriangleright	⊢ \vdash
∨ \vee	⩒ \veebar	∧ \wedge
≀ \wr		

箭头符号

⇓ \Downarrow	⇐ \Leftarrow
⇔ \Leftrightarrow	⇚ \Lleftarrow
⟸ \Longleftarrow	⟺ \Longleftrightarrow
⟹ \Longrightarrow	↰ \Lsh
⇗ \Nearrow	⇖ \Nwarrow
⇒ \Rightarrow	⇛ \Rrightarrow
↱ \Rsh	⇘ \Searrow
⇙ \Swarrow	⇑ \Uparrow
⇕ \Updownarrow	↺ \circlearrowleft
↻ \circlearrowright	↶ \curvearrowleft
↷ \curvearrowright	⇠ \dashleftarrow
⇢ \dashrightarrow	↓ \downarrow
⇊ \downdownarrows	⇃ \downharpoonleft
⇂ \downharpoonright	↩ \hookleftarrow
↪ \hookrightarrow	⇝ \leadsto
← \leftarrow	↢ \leftarrowtail
↽ \leftharpoondown	↼ \leftharpoonup

（续）

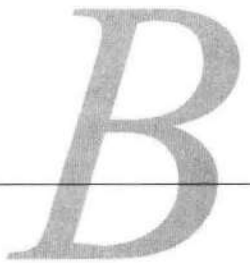
⇇ \leftleftarrows	↔ \leftrightarrow
⇆ \leftrightarrows	⇋ \leftrightharpoons
↭ \leftrightsquigarrow	↼ \leftsquigarrow
⟵ \longleftarrow	⟷ \longleftrightarrow
⟼ \longmapsto	⟶ \longrightarrow
↫ \looparrowleft	↬ \looparrowright
↦ \mapsto	⊸ \multimap
⇍ \nLeftarrow	⇎ \nLeftrightarrow
⇏ \nRightarrow	↗ \nearrow
↚ \nleftarrow	↮ \nleftrightarrow
↛ \nrightarrow	↖ \nwarrow
→ \rightarrow	↣ \rightarrowtail
⇁ \rightharpoondown	⇀ \rightharpoonup
⇄ \rightleftarrows	⇄ \rightleftarrows
⇌ \rightleftharpoons	⇌ \rightleftharpoons
⇉ \rightrightarrows	⇉ \rightrightarrows
⇝ \rightsquigarrow	↘ \searrow
↙ \swarrow	→ \to
↞ \twoheadleftarrow	↠ \twoheadrightarrow
↑ \uparrow	↕ \updownarrow
↕ \updownarrow	↿ \upharpoonleft
↾ \upharpoonright	⇈ \upuparrows

其他符号

$ \$	Å \AA	ⅎ \Finv
⅁ \Game	ℑ \Im	¶ \P
ℜ \Re	§ \S	∠ \angle
‵ \backprime	★ \bigstar	■ \blacksquare
▲ \blacktriangle	▼ \blacktriangledown	… \cdots
✓ \checkmark	® \circledR	Ⓢ \circledS
♣ \clubsuit	∁ \complement	© \copyright
$\ddot{s}$ \ddots	♢ \diamondsuit	ℓ \ell
∅ \emptyset	ð \eth	∃ \exists
♭ \flat	∀ \forall	ℏ \hbar
♡ \heartsuit	ħ \hslash	⫌ \iiint
∭ \iiint	∭ \iiint	𝚤 \imath
∞ \infty	𝚥 \jmath	… \ldots
∡ \measuredangle	♮ \natural	¬ \neg
∄ \nexists	⧌ \oiiint	∂ \partial
′ \prime	♯ \sharp	♠ \spadesuit
∢ \sphericalangle	ß \ss	▽ \triangledown
∅ \varnothing	△ \vartriangle	⋮ \vdots
℘ \wp	¥ \yen	

附录 B

开放数据源



B.1 政治和政府数据

data.gov网站

http://data.gov

该网站数据多与政府相关。

Socrata 网站

http://www.socrata.com/resources/

Socrata网站是探索政府相关数据的好去处。它提供了几种可视化工具，可帮助用户探索数据。

美国人口调查局

http://www.census.gov/data.html

该网站提供人口信息、地区分布和教育情况等美国公民相关的数据。

UN3ta

https://data.un.org/

UNdata是基于互联网的数据服务，提供UN统计数据库。

欧盟开放数据平台

http://open-data.europa.eu/en/data/

欧盟开放数据平台（European Union Open Data Portal）提供欧盟各机构的大量数据。

data.gov.uk

http://data.gov.uk/

英国政府网站，收录英国国家书目（British National Bibliography）：自1950年以来，英国出版的所有图书和其他出版物的元数据。

中情局世界概况

https://www.cia.gov/library/publications/the-world-factbook/

中情局世界概况（the CIA World Factbook）网站隶属美国中央情报局，提供了267个国家的历史、人口、经济、政府、基础设施和军事信息。

B.2　健康数据

healthdata.gov网站

https://www.healthdata.gov/

该网站提供流行病学、人口统计数据等医学相关的数据。

英国国民医疗服务体系和社会福利信息中心

http://www.hscic.gov.uk/home

该网站收录英国国民医疗服务体系（National Health Service）所提供的健康数据。

B.3　社会数据

Facebook Graph

https://developers.facebook.com/docs/graph-api

Facebook官方提供的API，用于查询该网站用户公开的海量信息。

Topsy网站

http://topsy.com/

Topsy网站维护了一个数据库，收录了Twitter用户发表的消息（推文），并开放检索功能，其中所存储的最早的消息可追溯至2006年。它还提供了几种对话分析工具。

谷歌趋势

http://www.google.com/trends/explore

谷歌趋势提供自2004年以来任意词语的搜索量（与全部搜索的占比）。

Likebutton网站

http://likebutton.com/

挖掘Facebook公开的数据——来自全球用户或你自己的朋友圈——了解当前人们喜欢（“Like”）什么。

B.4　其他开放数据集

亚马逊网络服务开放数据集

http://aws.amazon.com/datasets

亚马逊网络服务提供了一个开放数据集中心仓库，它包含多个数据集。其中一个非常有趣的数据集是1000 Genome Project（全球千人基因组计划），该计划尝试建立最全面的人类基因信息数据库。该仓库还存储了NASA的地球卫星图像。

DBPedia项目

http://wiki.dbpedia.org

维基百科提供了上千万条数据，主题多种多样，既有结构化数据，也有非结构化数据。DBPedia

项目雄心勃勃，意在为维基数据编制目录，并创建开放和可自由发布的数据库，便于每个人分析维基数据。

Freebase网站

http://www.freebase.com/

该社区数据库提供四千五百多万条涵盖多个主题的信息。

Gapminder网站

http://www.gapminder.org/data/

该网站数据来自世界卫生组织和世界银行，包括全球经济、医疗和社会统计数据。

B.5　金融数据

谷歌金融

https://www.google.com/finance

收录40年以来的股票数据，实时更新。

B.6　气候数据

美国国家气候数据中心

http://www.ncdc.noaa.gov/data-access/quick-links#loc-clim

美国国家气候数据中心提供了大量环境、气象和气候数据集，是世界最大的气象数据档案。

WeatherBase网站

http://www.weatherbase.com/

该网站提供全球四万多个城市的气候平均值、天气预报和当前天气状况数据。

Wunderground网站

http://www.wunderground.com/

该网站提供由卫星和气象观测站收集的温度、风力和其他气候测量数据。

B.7　体育数据

Pro-Football-Reference网站

http://www.pro-football-reference.com/

该网站提供足球及其他几种体育活动的数据。

B.8　报纸、图书及其他出版物

《纽约时报》

http://developer.nytimes.com/docs

该网站提供《纽约时报》自1851年以来的新闻文章，并为其编制了索引，开放查询服务。

Google Books Ngrams项目

http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

该项目为谷歌图书项目的一部分，可对几千万本电子书的全文进行查询和分析。

B.9　音乐数据

百万歌曲数据集

http://aws.amazon.com/datasets/6468931156960467

百万歌曲数据集（Million Song Data Set）为亚马逊网络服务的一部分，收录了超过一百万首歌曲和乐曲的元数据。

Amazon读者推荐

“学习pandas和matplotlib的很好的专业入门材料。”

“书如其名，内容充实、实用，例子有趣吸引人。如果你想利用Python进行数据分析的话，这本书很合适。”

Python数据分析实战

- 了解Python在信息处理、管理和检索方面的强大功能
- 学会如何利用Python及其衍生工具处理、分析数据
- 详尽探究三个真实Python数据分析案例，将理论付诸实践

图灵Python相关图书阅读路线图

Apress®
图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/数据分析
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-43220-9
ISBN 978-7-115-43220-9
定价：59.00元