Mastering pandas

Master the features and capabilities of pandas, a data analysis toolkit for Python

Femi Anthony

[PACKT] open source*
PUBLISHING   community experience distilled

Mastering pandas

Master the features and capabilities of pandas,
a data analysis toolkit for Python

Femi Anthony

[PACKT] open source*

PUBLISHING

community experience distilled

BIRMINGHAM - MUMBAI

Mastering pandas

Credits

Author
Femi Anthony

Reviewers
Opeyemi Akinjayeju

Louis Hénault

Carlos Marin

Commissioning Editor
Karthikey Pandey

Acquisition Editor
Kevin Colaco

Content Development Editor
Arun Nadar

Technical Editor
Mohita Vyas

Copy Editors
Tani Kothari

Jasmine Nadar

Vikrant Phadke

Project Coordinator
Neha Bhatnagar

Proofreader
Safis Editing

Indexer
Tejal Soni

Graphics
Jason Monteiro

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Femi Anthony is a seasoned and knowledgeable software programmer, with over 15 years experience in a vast array of languages, including Perl, C, C++, Java, and Python. He has worked in both the Internet space and financial services space for many years and is now working for a well-known financial data company. He holds a bachelor's degree in mathematics with computer science from MIT and a master's degree from the University of Pennsylvania. His pet interests include data science, machine learning, and Python. Femi is working on a few side projects in these areas. His hobbies include reading, soccer, and road cycling. You can follow him at `@dataphanatik`, and for any queries, contact him at `femibyte@gmail.com`.

First and foremost, I would like to thank my wife, Ene, for her support throughout my career and in writing this book. She has been my inspiration and motivation for continuing to improve my knowledge and helping me move ahead in my career. She is my rock, and I dedicate this book to her. I also thank my wonderful children, Femi, Lara, and our new addition, Temi, for always making me smile and for understanding on those days when I was writing this book instead of playing games with them.

I would also like to thank my book reviewers—Opeyemi Akinjayeju, who is a dear friend of mine, as well as Louis Hénault and Carlos Marin—for their invaluable feedback and input toward the completion of this book. Lastly, I would like to thank my parents, George and Katie Anthony, for instilling a strong work ethic in me from an early age.

About the Reviewers

Opeyemi Akinjayeju is risk management professional. He holds graduate degrees in statistics (Penn State University) and economics (Georgia Southern University), and has built predictive models for insurance companies, banks, captive automotive finance lenders, and consulting firms. He enjoys analyzing data and solving complex business problems using SAS, R, EViews/Gretl, Minitab, SQL, and Python. Opeyemi is also an adjunct at Northwood University where he designs and teaches undergraduate courses in microeconomics and macroeconomics.

Louis Hénault is a data scientist at OgilvyOne Paris. He loves combining mathematics and computer science to solve real-world problems in an innovative way. After getting a master's degree in engineering with a major in data sciences and another degree in applied mathematics in France, he entered into the French start-up ecosystem, working on several projects. Louis has gained experience in various industries, including geophysics, application performance management, online music platforms, e-commerce, and digital advertising. He is now working for a leading customer engagement agency, where he helps clients unlock the complete value of customers using big data.

> I've met many outstanding people in my life who have helped me become what I am today. A great thank you goes to the professors, authors, and colleagues who taught me many fantastic things. Of course, I can't end this without a special thought for my friends and family.

Carlos Marin is a software engineer at Rackspace, where he maintains and develops a suite of applications that manage networking devices in Rackspace's data centers. He has made contributions to OpenStack, and has worked with multiple teams and on multiple projects within Rackspace, from the Identity API to big data and analytics.

Carlos graduated with a degree in computer engineering from the National Autonomous University of Mexico. Prior to joining Rackspace, he worked as a consultant, developing software for multiple financial enterprises in programming languages. In Austin, Texas, he regularly attends local technology events and user groups. He also spends time volunteering and pursuing outdoor adventures.

> I'm grateful to my parents and family, who have always believed in me.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

[i]

Preface

Welcome to *Mastering pandas*. This book will teach you how to effectively use pandas, which is a one of the most popular Python packages today for performing data analysis. The first half of this book starts off with the rationale for performing data analysis. Then it introduces Python and pandas in particular, taking you through the installation steps, what pandas is all about, what it can be used for, data structures in pandas, and how to select, merge and group data in pandas. Then it covers handling missing data and time series data, as well as plotting for data visualization.

The second half of this book shows you how to use pandas to perform inferential statistics using the classical and Bayesian approaches, followed by a chapter on pandas architecture, before rounding off with a whirlwind tour of machine learning, which introduces the scikit-learn library. The aim of this book is to immerse you into pandas through the use of illustrative examples on real-world datasets.

What this book covers

Chapter 1, *Introduction to pandas and Data Analysis*, explains the motivation for doing data analysis, introduces the Python language and the pandas library, and discusses how they can be used for data analysis. It also describes the benefits of using pandas for data analysis.

Chapter 2, *Installation of pandas and the Supporting Software*, gives a detailed description on how to install pandas. It gives installation instructions across multiple operating system platforms: Unix, MacOS X, and Windows. It also describes how to install supporting software, such as NumPy and IPython.

Chapter 3, *The pandas Data Structures*, introduces the data structures that form the bedrock of the pandas library. The `numpy.ndarray` data structure is first introduced and discussed as it forms the basis for the pandas.Series and `pandas.DataFrame` data structures, which are the foundation data structures used in pandas. This chapter may be the most important on in the book, as knowledge of these data structures is absolutely necessary to do data analysis using pandas.

Chapter 4, *Operations in pandas, Part I – Indexing and Selecting*, focuses on how to access and select data from the pandas data structures. It discusses the various ways of selecting data via Basic, Label, Integer, and Mixed Indexing. It explains more advanced indexing concepts such as MultiIndex, Boolean indexing, and operations on Index types.

Chapter 5, *Operations in pandas, Part II – Grouping, Merging, and Reshaping of Data*, tackles the problem of rearranging data in pandas' data structures. The various functions in pandas that enable the user to rearrange data are examined by utilizing them on real-world datasets. This chapter examines the different ways in which data can be rearranged: by aggregation/grouping, merging, concatenating, and reshaping.

Chapter 6, *Missing Data, Time Series, and Plotting using Matplotlib*, discusses topics that are necessary for the pre-processing of data that is to be used as input for data analysis, prediction, and visualization. These topics include how to handle missing values in the input data, how to handle time series data, and how to use the matplotlib library to plot data for visualization purposes.

Chapter 7, *A Tour of Statistics – The Classical Approach*, takes you on a brief tour of classical statistics and shows how pandas can be used together with Python's statistical packages to conduct statistical analyses. Various statistical topics are addressed, including statistical inference, measures of central tendency, hypothesis testing, Z- and T-tests, analysis of variance, confidence intervals, and correlation and regression.

Chapter 8, *A Brief Tour of Bayesian Statistics*, discusses an alternative approach to performing statistical analysis, known as Bayesian analysis. This chapter introduces Bayesian statistics and discusses the underlying mathematical framework. It examines the various probability distributions used in Bayesian analysis and shows how to generate and visualize them using matplotlib and scipy.stats. It also introduces the PyMC library for performing Monte Carlo simulations, and provides a real-world example of conducting a Bayesian inference using online data.

Chapter 9, *The pandas Library Architecture*, provides a fairly detailed description of the code underlying pandas. It gives a breakdown of how the pandas library code is organized and describes the various modules that make up pandas, with some details. It also has a section that shows the user how to improve Python and pandas's performance using extensions.

Chapter 10, *R and pandas Compared*, focuses on comparing pandas with R, the stats package on which much of pandas's functionality is based. This chapter compares R data types and their pandas equivalents, and shows how the various operations compare in both libraries. Operations such as slicing, selection, arithmetic operations, aggregation, group-by, matching, split-apply-combine, and melting are compared.

Chapter 11, *Brief Tour of Machine Learning*, takes you on a whirlwind tour of machine learning, with focus on using the pandas library as a tool to preprocess input data into machine learning programs. It also introduces the scikit-learn library, which is the most widely used machine learning toolkit in Python. Various machine learning techniques and algorithms are introduced by applying them to a well-known machine learning classification problem: which passengers survived the sinking of the Titanic?

What you need for this book

This software applies to all the chapters of the book:

- Windows/Mac OS/Linux
- Python 2.7.x
- pandas
- IPython
- R
- scikit-learn

For hardware, there are no specific requirements, since Python and pandas can run on any PC that has Mac, Linux, or Windows.

Who this book is for

This book is intended for Python programmers, mathematicians, and analysts who already have a basic understanding of Python and wish to learn about its data analysis capabilities in depth. Maybe your appetite has been whetted after using Python for a few months, or maybe you are an R user who wishes to investigate what Python has to offer with regards to data analysis. In either case, this book will help you master the core features and capabilities of pandas for data analysis. It would be helpful for the user to have some experience using Python or experience with a data analysis package such as R.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: " Upon installation, the following folders should be added to the `PATH` environment variable: `C:\Python27\` and `C:\Python27\Tools\Scripts`."

Any command-line input or output is written as follows:

```
brew install readline
brew install zeromq
pip install ipython pyzmq tornado pygments
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The preceding image of **PYMC pandas Example** is taken from `http://healthyalgorithms.files.wordpress.com/2012/01/pymc-pandas-example.png`."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

> You can also download the code from the GitHub repository at:
> `https://github.com/femibyte/mastering_pandas`

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introduction to pandas and Data Analysis

In this chapter, we address the following:

- Motivation for data analysis
- How Python and pandas can be used for data analysis
- Description of the pandas library
- Benefits of using pandas

Motivation for data analysis

In this section, we will discuss the trends that are making data analysis an increasingly important field of endeavor in today's fast-moving technological landscape.

We live in a big data world

The term **big data** has become one of the hottest technology buzzwords in the past two years. We now increasingly hear about big data in various media outlets, and big data startup companies have increasingly been attracting venture capital. A good example in the area of retail would be Target Corporation, which has invested substantially in big data and is now able to identify potential customers by using big data to analyze people's shopping habits online; refer to a related article at `http://nyti.ms/19LT8ic`.

Loosely speaking, big data refers to the phenomenon wherein the amount of data exceeds the capability of the recipients of the data to process it. Here is a Wikipedia entry on big data that sums it up nicely: `http://en.wikipedia.org/wiki/Big_data`.

4 V's of big data

A good way to start thinking about the complexities of big data is along what are called the 4 dimensions, or **4 V's of big data**. This model was first introduced as the 3V's by Gartner analyst Doug Laney in 2001. The 3V's stood for Volume, Velocity, and Variety, and the 4th V, Veracity, was added later by IBM. Gartner's official definition is as follows:

> *"Big data is high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization."*

> *Laney, Douglas. "The Importance of 'Big Data': A Definition", Gartner*

Volume of big data

The **volume** of data in the big data age is simply mind-boggling. According to IBM, by 2020, the total amount of data on the planet would have ballooned to 40 zettabytes. You heard that right-40 zettabytes is 43 trillion gigabytes, which is about 4×1021 bytes. For more information on this refer to the Wikipedia page on Zettabyte - `http://en.wikipedia.org/wiki/Zettabyte`.

To get a handle of how much data this would be, let me refer to an EMC press release published in 2010, which stated what 1 zettabyte was approximately equal to:

> *"The digital information created by every man, woman and child on Earth 'Tweeting' continuously for 100 years " or "75 billion fully-loaded 16 GB Apple iPads, which would fill the entire area of Wembley Stadium to the brim 41 times, the Mont Blanc Tunnel 84 times, CERN's Large Hadron Collider tunnel 151 times, Beijing National Stadium 15.5 times or the Taipei 101 Tower 23 times..."*

> *EMC study projects 45× data growth by 2020*

The growth rate of data has been fuelled largely by a few factors, such as the following:

- The rapid growth of the Internet.
- The conversion from analog to digital media coupled with an increased capability to capture and store data, which in turn has been made possible with cheaper and more capable storage technology. There has been a proliferation of digital data input devices such as cameras and wearables, and the cost of huge data storage has fallen rapidly. Amazon Web Services is a prime example of the trend toward much cheaper storage.

The *Internetification* of devices, or rather *Internet of Things*, is the phenomenon wherein common household devices, such as our refrigerators and cars, will be connected to the Internet. This phenomenon will only accelerate the above trend.

Velocity of big data

From a purely technological point of view, **velocity** refers to the throughput of big data, or how fast the data is coming in and is being processed. This has ramifications on how fast the recipient of the data needs to process it to keep up. Real-time analytics is one attempt to handle this characteristic. Tools that can help enable this include Amazon Web Services Elastic Map Reduce.

At a more macro level, the velocity of data can also be regarded as the increased speed at which data and information can now be transferred and processed faster and at greater distances than ever before.

The proliferation of high-speed data and communication networks coupled with the advent of cell phones, tablets, and other connected devices, are primary factors driving information velocity. Some measures of velocity include the number of tweets per second and the number of emails per minute.

Variety of big data

The **variety** of big data comes from having a multiplicity of data sources that generate the data, and the different formats of the data that are produced.

This results in a technological challenge for the recipients of the data who have to process it. Digital cameras, sensors, the web, cell phones, and so on are some of the data generators that produce data in differing formats, and the challenge comes in being able to handle all these formats and extract meaningful information from the data. The ever-changing nature of the data formats with the dawn of the big data era has led to a revolution in the database technology industry, with the rise of NoSQL databases to handle what is known as *unstructured* data or rather data whose format is fungible or constantly changing. For more information on Couchbase, refer to "Why NoSQL- `http://bit.ly/1c3iVEc`.

Veracity of big data

The 4th characteristic of big data – **veracity**, which was added later, refers to the need to validate or confirm the *correctness* of the data or the fact that the data represents the truth. The sources of data must be verified and the errors kept to a minimum. According to an estimate by IBM, poor data quality costs the US economy about $3.1 trillion dollars a year. For example, medical errors cost the United States $19.5 billion in 2008; for more information you can refer to a related article at `http://bit.ly/1CTah5r`. Here is an info-graphic by IBM that summarizes the 4V's of big data:
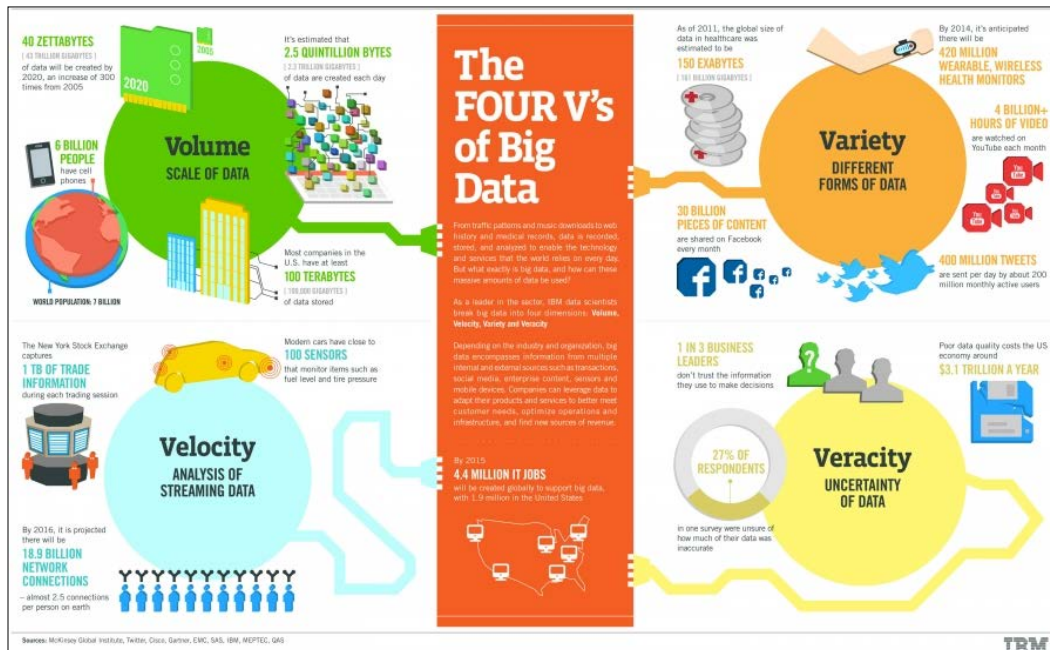
IBM on the 4 V's of big data

So much data, so little time for analysis

Data analytics has been described by Eric Schmidt, the former CEO of Google, as the *Future of Everything*. For reference, you can check out a YouTube video called *Why Data Analytics is the Future of Everything* at `http://bit.ly/1KmqGCP`.

The volume and velocity of data will continue to increase in the big data age. Companies that can efficiently collect, filter, and analyze data results in information that allows them to better meet the needs of their customers in a much quicker timeframe will gain a significant competitive advantage over their competitors. For example, data analytics (*Culture of Metrics*) plays a very key role in the business strategy of `http://www.amazon.com/`. For more information refer to Amazon.com Case Study, Smart Insights at `http://bit.ly/1glnA1u`.

The move towards real-time analytics

As technologies and tools have evolved, to meet the ever-increasing demands of business, there has been a move towards what is known as real-time analytics. More information on Insight Everywhere, Intel available at `http://intel.ly/1899xqo`.

In the big data Internet era, here are some examples:

- Online businesses demand instantaneous insights into how the new products/features they have introduced in their online market are doing and how they can adjust their online product mix accordingly. Amazon is a prime example of this with their *Customers Who Viewed This Item Also Viewed* feature.
- In finance, risk management and trading systems demand almost instantaneous analysis in order to make effective decisions based on data-driven insights.

How Python and pandas fit into the data analytics mix

The **Python** programming language is one of the fastest growing languages today in the emerging field of data science and analytics. Python was created by Guido von Russom in 1991, and its key features include the following:

- Interpreted rather than compiled
- Dynamic type system
- Pass by value with object references
- Modular capability
- Comprehensive libraries
- Extensibility with respect to other languages

- Object orientation
- Most of the major programming paradigms-procedural, object-oriented, and to a lesser extent, functional.

> For more information, refer the Wikipedia page on Python at `http://en.wikipedia.org/wiki/Python_%28programming_language%29`.
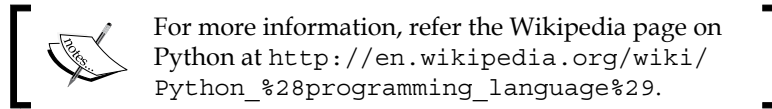
Among the characteristics that make Python popular for data science are its very user-friendly (human-readable) syntax, the fact that it is interpreted rather than compiled (leading to faster development time), and its very comprehensive library for parsing and analyzing data, as well as its capacity for doing numerical and statistical computations. Python has libraries that provide a complete toolkit for data science and analysis. The major ones are as follows:

- **NumPy**: The general-purpose array functionality with emphasis on numeric computation
- **SciPy**: Numerical computing
- **Matplotlib**: Graphics
- **pandas**: Series and data frames (1D and 2D array-like types)
- **Scikit-Learn**: Machine learning
- **NLTK**: Natural language processing
- **Statstool**: Statistical analysis

For this book, we will be focusing on the 4th library listed in the preceding list, pandas.

What is pandas?

The **pandas** is a high-performance open source library for data analysis in Python developed by Wes McKinney in 2008. Over the years, it has become the de-facto standard library for data analysis using Python. There's been great adoption of the tool, a large community behind it, (220+ contributors and 9000+ commits by 03/2014), rapid iteration, features, and enhancements continuously made.

Some key features of pandas include the following:

- It can process a variety of data sets in different formats: time series, tabular heterogeneous, and matrix data.
- It facilitates loading/importing data from varied sources such as CSV and DB/SQL.

- It can handle a myriad of operations on data sets: subsetting, slicing, filtering, merging, groupBy, re-ordering, and re-shaping.

- It can deal with missing data according to rules defined by the user/ developer: ignore, convert to 0, and so on.

- It can be used for parsing and munging (conversion) of data as well as modeling and statistical analysis.

- It integrates well with other Python libraries such as statsmodels, SciPy, and scikit-learn.

- It delivers fast performance and can be speeded up even more by making use of **Cython** (C extensions to Python).

For more information go through the official pandas documentation available at `http://pandas.pydata.org/pandas-docs/stable/`.

Benefits of using pandas

The pandas forms a core component of the Python data analysis corpus. The distinguishing feature of pandas is the suite of data structures that it provides, which is naturally suited to data analysis, primarily the DataFrame and to a lesser extent Series (1-D vectors) and Panel (3D tables).

Simply put, pandas and statstools can be described as Python's answer to R, the data analysis and statistical programming language that provides both the data structures, such as R-data frames, and a rich statistical library for data analysis.

The benefits of pandas over using a language such as Java, C, or C++ for data analysis are manifold:

- **Data representation**: It can easily represent data in a form naturally suited for data analysis via its DataFrame and Series data structures in a concise manner. Doing the equivalent in Java/C/C++ would require many lines of custom code, as these languages were not built for data analysis but rather networking and kernel development.

- **Data subsetting and filtering**: It provides for easy subsetting and filtering of data, procedures that are a staple of doing data analysis.

- **Concise and clear code**: Its concise and clear API allows the user to focus more on the core goal at hand, rather than have to write a lot of scaffolding code in order to perform routine tasks. For example, reading a CSV file into a DataFrame data structure in memory takes two lines of code, while doing the same task in Java/C/C++ would require many more lines of code or calls to non-standard libraries, as illustrated in the following table. Here, let's suppose that we had the following data:

Country	Year	CO2 Emissions	Power Consumption	Fertility Rate	Internet Usage Per 1000 People	Life Expectancy	Population
Belarus	2000	5.91	2988.71	1.29	18.69	68.01	1.00E+07
Belarus	2001	5.87	2996.81		43.15		9970260
Belarus	2002	6.03	2982.77	1.25	89.8	68.21	9925000
Belarus	2003	6.33	3039.1	1.25	162.76		9873968
Belarus	2004		3143.58	1.24	250.51	68.39	9824469
Belarus	2005			1.24	347.23	68.48	9775591

In a CSV file, this data that we wish to read would look like the following:

```
Country,Year,CO2Emissions,PowerConsumption,FertilityRate,
InternetUsagePer1000, LifeExpectancy, Population
Belarus,2000,5.91,2988.71,1.29,18.69,68.01,1.00E+07
Belarus,2001,5.87,2996.81,,43.15,,9970260
Belarus,2002,6.03,2982.77,1.25,89.8,68.21,9925000
...
Philippines,2000,1.03,514.02,,20.33,69.53,7.58E+07
Philippines,2001,0.99,535.18,,25.89,,7.72E+07
Philippines,2002,0.99,539.74,3.5,44.47,70.19,7.87E+07
...
Morocco,2000,1.2,489.04,2.62,7.03,68.81,2.85E+07
Morocco,2001,1.32,508.1,2.5,13.87,,2.88E+07
Morocco,2002,1.32,526.4,2.5,23.99,69.48,2.92E+07
..
```

The data here is taken from World Bank Economic data available at: `http://data.worldbank.org`.

In Java, we would have to write the following code:

```
public class CSVReader {
public static void main(String[] args) {
```

```
        String[] csvFile=args[1];
        CSVReader csvReader = new csvReader();
        List<Map>dataTable=csvReader.readCSV(csvFile);
    }
public void readCSV(String[] csvFile)
{
  BufferedReader bReader=null;
  String line="";
  String delim=",";
  //Initialize List of maps, each representing a line of the csv file
  List<Map> data=new ArrayList<Map>();
  try {
        bufferedReader = new BufferedReader(new
FileReader(csvFile));
        // Read the csv file, line by line
          while ((line = br.readLine()) != null){
            String[] row = line.split(delim);
            Map<String,String> csvRow=new HashMap<String,String>();
            csvRow.put('Country')=row[0];
            csvRow.put('Year')=row[1];
        csvRow.put('CO2Emissions')=row[2]; csvRow.
put('PowerConsumption')=row[3];
              csvRow.put('FertilityRate')=row[4];
              csvRow.put('InternetUsage')=row[1];
              csvRow.put('LifeExpectancy')=row[6];
              csvRow.put('Population')=row[7];
              data.add(csvRow);
          }
      } catch (FileNotFoundException e) {
  e.printStackTrace();
      } catch (IOException e) {
    e.printStackTrace();
      }
 return data;
}
```

But, using pandas, it would take just two lines of code:

```
import pandas as pd
worldBankDF=pd.read_csv('worldbank.csv')
```

In addition, pandas is built upon the NumPy libraries and hence, inherits many of the performance benefits of this package, especially when it comes to numerical and scientific computing. One oft-touted drawback of using Python is that as a scripting language, its performance relative to languages like Java/C/C++ has been rather slow. However, this is not really the case for pandas.

Summary

We live in a big data era characterized by the 4V's- volume, velocity, variety, and veracity. The volume and velocity of data are ever increasing for the foreseeable future. Companies that can harness and analyze big data to extract information and take actionable decisions based on this information will be the winners in the marketplace. Python is a fast-growing, user-friendly, extensible language that is very popular for data analysis.

The pandas is a core library of the Python toolkit for data analysis. It provides features and capabilities that make it much easier and faster for data analysis than many other popular languages such as Java, C, C++, and Ruby.

Thus, given the strengths of Python listed in the preceding section as a choice for the analysis of data, the data analysis practitioner utilizing Python should become quite adept at pandas in order to become more effective. This book aims to assist the user in achieving this goal.

2
Installation of pandas and the Supporting Software

Before we can start work on pandas for doing data analysis, we need to make sure that the software is installed and the environment is in proper working order. This section deals with the installation of Python (if necessary), the pandas library, and all necessary dependencies for the Windows, Mac OS X, and Linux platforms. The topics we address include the following:

- Selecting a version of Python
- Installing Python
- Installing pandas (0.16.0)
- Installing IPython and Virtualenv

The steps outlined in the following section should work for the most part, but your mileage may vary depending upon the setup. On different operating system versions, the scripts may not always work perfectly, and the third-party software packages already in the system may sometimes conflict with the provided instructions.

Selecting a version of Python to use

Before proceeding with the installation and download of Python and pandas, we need to consider the version of Python we're going to use. Currently, there are two versions flavors of Python in current use, namely Python 2.7.x and Python 3. If the reader is new to Python as well as pandas, the question becomes which version of the language he/she should adopt.

On the surface, Python 3.x would appear to be the better choice since Python 2.7.x is supposed to be the legacy, and Python 3.x is supposed to be the future of the language.

> For reference, you can go through the documentation on this with the title *Python2orPython3* at `https://wiki.python.org/moin/Python2orPython3`.

The main differences between Python 2.x and 3 include better Unicode support in Python 3, print and exec changed to functions, and integer division. For more details, see *What's New in Python 3.0* at `http://docs.python.org/3/whatsnew/3.0.html`.

However, for scientific, numeric, or data analysis work, Python 2.7 is recommended over Python 3 for the following reason: Python 2.7 is the preferred version for most current distributions and the support for Python 3.x was not as strong for some libraries, although that is increasingly becoming less of an issue.

For reference, have a look at the documentation titled *Will Scientists Ever Move to Python 3?* at `http://bit.ly/1DOgNuX`. Hence, this book will use Python 2.7. It does not preclude the use of Python 3, and developers using Python 3 can easily make the necessary code changes to the examples by referring to the following documentation: *Porting Python 2 Code to Python 3* at `http://docs.python.org/2/howto/pyporting.html`.

Python installation

Here, we detail the installation of Python on multiple platforms – Linux, Windows, and Mac OS X.

Linux

If you're using Linux, Python most probably came pre-installed. If you're not sure, type the following at the command prompt:

```
which python
```

Python is likely to be found in one of the following folders on Linux depending upon your distribution and particular installation:

- `/usr/bin/python`
- `/bin/python`
- `/usr/local/bin/python`
- `/opt/local/bin/python`

You can determine which particular version of Python is installed, by typing the following in the command prompt:

```
python --version
```

In the rare event that Python isn't already installed, you need to figure out which flavor of Linux you're using, then download and install it. Here are the install commands as well as links to the various Linux Python distributions:

1. Debian/Ubuntu (14.04)

   ```
   sudo apt-get install python2.7
   sudo apt-get install python2.7-devel
   ```

   Debian Python page at `https://wiki.debian.org/Python`.

2. Redhat Fedora/Centos/RHEL

   ```
   sudo yum install python
   sudo yum install python-devel
   ```

   Fedora software installs at `http://bit.ly/1B2RpCj`.

3. Open Suse

   ```
   sudo zypper install python
   sudo zypper install python-devel
   ```

   More information on installing software can be found at `http://en.opensuse.org/YaST_Software_Management`.

4. `Slackware`: For this distribution of Linux, it may be best to download a compressed tarball and install it from the source as described in the following section.

Installing Python from compressed tarball

If none of the preceding methods work for you, you can also download a compressed tarball (XZ or Gzip) and get it installed. Here is a brief synopsis on the steps:

```
#Install dependencies
sudo apt-get install build-essential
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev
libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
#Download the tarball
```

```
mkdir /tmp/downloads
cd /tmp/downloads
wget http://python.org/ftp/python/2.7.5/Python-2.7.5.tgz
tar xvfz Python-2.7.5.tgz
cd Python-2.7.5
# Configure, build and install
./configure --prefix=/opt/python2.7 --enable-shared
make
make test
sudo make install
echo "/opt/python2.7/lib" >> /etc/ld.so.conf.d/opt-python2.7.conf
ldconfig
cd ..
rm -rf /tmp/downloads
```

Information on this can be found at the Python download page at `http://www.python.org/download/`.

Windows

Unlike Linux and Mac distributions, Python does not come pre-installed on Windows.

Core Python installation

The standard method is to use the Windows installers from CPython's team, which are MSI packages. The MSI packages can be downloaded from here: `http://www.python.org/download/releases/2.7.6/`.

Select the appropriate Windows package depending upon whether your Windows version is 32-bit or 64-bit. Python by default gets installed to a folder containing the version number, so in this case, it will be installed to the following location: `C:\Python27`.

This enables you to have multiple versions of Python running without problems. Upon installation, the following folders should be added to the PATH environment variable: `C:\Python27\` and `C:\Python27\Tools\Scripts`.

Third-party Python software installation

There are a couple of Python tools that need to be installed in order to make the installation of other packages such as pandas easier. Install **Setuptools** and **pip**. Setuptools is very useful for installing other Python packages such as pandas. It adds to the packaging and installation functionality that is provided by the `distutils` tool in the standard Python distribution.

To install Setuptools, download the `ez_setup.py` script from the following link: `https://bitbucket.org/pypa/setuptools/raw/bootstrap`.

Then, save it to `C:\Python27\Tools\Scripts`.

Then, run `ez_setup.py`: `C:\Python27\Tools\Scripts\ez_setup.py`.

The associated command `pip` provides the developer with an easy-to-use command that enables a quick and easy installation of Python modules. Download the `get-pip` script from the following link: `http://www.pip-installer.org/en/latest/`.

Then, run it from the following location: `C:\Python27\Tools\Scripts\get-pip.py`.

For reference, you can also go through the documentation titled *Installing Python on Windows* at `http://docs.python-guide.org/en/latest/starting/install/win/`.

There are also third-party providers of Python on Windows that make the task of installation even easier. They are listed as follows:

- **Enthought**: `https://enthought.com/`
- **Continuum Analytics**: `http://www.continuum.io/`
- **Active State Python**: `http://www.activestate.com/activepython`

Mac OS X

Python 2.7 comes pre-installed on the current and recent releases (past 5 years) of Mac OS X. The pre-installed Apple-provided build can be found in the following folders on the Mac:

- `/System/Library/Frameworks/Python.framework`
- `/usr/bin/python`

However, you can install your own version from `http://www.python.org/download/`. The one caveat to this is that you will now have two installations of Python, and you have to be careful to make sure the paths and environments are cleanly separated.

Installation using a package manager

Python can also be installed using a package manager on the Mac such as Macports or Homebrew. I will discuss installation using Homebrew here as it seems to be the most user-friendly. For reference, you can go through the documentation titled *Installing Python on Mac OS X* at `http://docs.python-guide.org/en/latest/starting/install/osx/`. Here are the steps:

1. Install Homebrew and run:

    ```
    ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go)"
    ```

    You then need to add the Homebrew folder at the top of your `PATH` environment variable.

2. Install Python 2.7 at the Unix prompt:

    ```
    brew install python
    ```

3. Install third-party software: Distribute and pip. Installation of Homebrew automatically installs these packages. Distribute and pip enable one to easily download and install/uninstall Python packages.

Installation of Python and pandas from a third-party vendor

The most straightforward way to install Python, pandas, and their associated dependencies would be to install a packaged distribution by using a third-party vendor such as Enthought or Continuum Analytics.

I used to prefer Continuum Analytics Anaconda over Enthought because Anaconda was given away free while Enthought used to charge a subscription for full access to all its numerical modules. However, with the latest release of Enthought Canopy, there is little to separate the two distributions. Nevertheless, my personal preference is for Anaconda, so it is the distribution whose installation I will describe.

For reference, see *Anaconda Python Distribution* at `http://bit.ly/1aBhmgH`. I will now give a brief description about the Anaconda package and how to install it.

Continuum Analytics Anaconda

Anaconda is a free Python distribution focused on large-scale data processing, analytics, and numeric computing. The following are the key features of Anaconda:

- It includes the most popular Python packages for scientific, engineering, numerical, and data analysis.
- It is completely free and available on Linux, Windows, and Mac OS X platforms.
- Installations do not require root or local admin privileges, and the entire package installs in a single folder.
- Multiple installations can coexist, and the installation does not affect preexisting Python installations on the system.
- It includes modules such as Cython, NumPy, SciPy, pandas, IPython, matplotlib, and homegrown Continuum packages such as Numba, Blaze, and Bokeh.

> For more information on this, refer to the link at
> `https://store.continuum.io/cshop/anaconda`.

Installing Anaconda

The following instructions detail how to install Anaconda on all three platforms. The download location is `http://continuum.io/downloads`. The version of Python is Python 2.7 in Anaconda by default.

Linux

Perform the following steps for installation:

1. Download the Linux installer (32/64-bit) from the download location.
2. In a terminal, run the following command:

   `bash <Linux installer file>`

   For example, `bash Anaconda-1.8.0-Linux-x86_64.sh`.

3. Accept the license terms.
4. Specify the install location. I tend to use `$HOME/local` for my local third-party software installations.

[**17**]

Mac OS X

Perform the following steps for installation:

1. Download the Mac installer (`.pkg file - 64-bit`) from the download location.

2. Double click on the `.pkg` file to install and follow the instructions on the window that pops up. For example, package file name: `Anaconda-1.8.0-MacOSX-x86_64.pkg`.

Windows

Perform the following steps for the Windows environment:

1. Download the Windows installer (`.exe file - 32/64-bit`) from the download location.

2. Double click on the `.pkg` file to install and follow the instructions on the window that pops up. For example, package file name: `Anaconda-1.8.0-MacOSX-x86_64.pkg`.

Final step for all platforms

As a shortcut, you can define `ANACONDA_HOME` to be the folder into which Anaconda was installed. For example, on my Linux and Mac OS X installations, I have the following environment variable setting:

```
ANACONDA_HOME=$HOME/local/anaconda
```

On Windows, it would be as follows:

```
set ANACONDA_HOME=C:\Anaconda
```

Add the Anaconda `bin` folder to your `PATH` environment variable. If you wish to use the Python Anaconda by default, you can do this by making sure that `$ANACONDA_HOME/bin` is at the head of the `PATH` variable before the folder containing System Python. If you don't want to use the Anaconda Python by default, you have the following two options:

1. Activate the Anaconda environment each time as needed. This can be done as follows:
   ```
   source $HOME/local/anaconda/bin/activate $ANACONDA_HOME
   ```

2. Create a separate environment for Anaconda. This can be done by using the built-in `conda` command as described here: `https://github.com/pydata/conda`.

> For more information, read the Conda documentation at `http://docs.`
> `continuum.io/conda/index.html`. More detailed instructions on
> installing Anaconda can be obtained from the Anaconda Installation page
> at `http://docs.continuum.io/anaconda/install.html`.

Other numeric or analytics-focused Python distributions

The following is a synopsis of various third-party data analysis-related Python distributions. All of the following distributions include pandas:

- **Continuum Analytics Anaconda**: Free enterprise-ready Python distribution focused on large-scale data processing, analytics, and numeric computing. For details, refer to `https://store.continuum.io/cshop/anaconda/`.

- **Enthought Canopy**: Comprehensive Python data analysis environment. For more information, refer to `https://www.enthought.com/products/canopy/`.

- **Python(x,y)**: Free scientific and engineering-oriented Python distribution for numerical computing, data analysis, and visualization. It is based on the Qt GUI package and Spyder interactive scientific development environment. For more information, refer to `https://code.google.com/p/pythonxy/`.

- **WinPython**: Free open source distribution of Python for the Windows platform focused on scientific computing. For more information, refer to `http://winpython.sourceforge.net/`.

For more information on Python distributions, go to `http://bit.ly/1yOzB7o`.

Downloading and installing pandas

The pandas library is part of the Python language, so we can now proceed to install pandas. At the time of writing this book, the latest stable version of pandas available is version 0.12. The various dependencies along with the associated download locations are as follows:

Package	Required	Description	Download location
NumPy : 1.6.1 or higher	Required	NumPy library for numerical operations	`http://www.numpy.org/`
python-dateutil 1.5	Required	Date manipulation and utility library	`http://labix.org/`

Package	Required	Description	Download location
`Pytz`	Required	Time zone support	`http://sourceforge.net/`
`numexpr`	Optional, recommended	Speeding up of numerical operations	`https://code.google.com/`
`bottleneck`	Optional, recommended	Performance-related	`http://berkeleyanalytics.com/`
`Cython`	Optional, recommended	C-extensions for Python used for optimization	`http://cython.org/`
`SciPy`	Optional, recommended	Scientific toolset for Python	`http://scipy.org/`
`PyTables`	Optional	Library for HDF5-based storage	`http://pytables.github.io/`
`matplotlib`	Optional, recommended	Matlab-like Python plotting library	`http://sourceforge.net/`
`statsmodels`	Optional	Statistics module for Python	`http://sourceforge.net/`
`openpyxl`	Optional	Library to read/write Excel files	`https://www.python.org/`
`xlrd/xlwt`	Optional	Libraries to read/write Excel files	`http://python-excel.org/`
`boto`	Optional	Library to access Amazon S3	`https://www.python.org/`
`BeautifulSoup` and one of `html5lib`, `lxml`	Optional	Libraries needed for the read_html() function to work	`http://www.crummy.com/`
`html5lib`	Optional	Library for parsing HTML	`https://pypi.python.org/pypi/html5lib`
`lmxl`	Optional	Python library for processing XML and HTML	`http://lxml.de/`

Linux

Installing pandas is fairly straightforward for popular flavors of Linux. First, make sure that the Python `.dev` files are installed. If not, then install them as explained in the following section.

Ubuntu/Debian

For the Ubantu/Debian environment, run the following command:

```
sudo apt-get install python-dev
```

Red Hat

For the Red Hat environment, run the following command:

```
yum install python-dev
```

Now, I will show you how to install pandas.

Ubuntu/Debian

For installing pandas in the Ubuntu/Debian environment, run the following command:

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

Fedora

For Fedora, run the following command:

```
sudo yum install python-pandas
```

OpenSuse

Install Python-pandas via YaST Software Management or use the following command:

```
sudo zypper install python-pandas
```

Sometimes, additional dependencies may be needed for the preceding installation, particularly in the case of Fedora. In this case, you can try installing additional dependences:

```
sudo yum install gcc-gfortran gcc44-gfortran libgfortran lapack blas python-devel
```
```
sudo python-pip install numpy
```

Mac

There are a variety of ways to install pandas on Mac OS X. They are explained in the following sections.

Source installation

The pandas have a few dependencies for it to work properly, some are required and the others are optional, although needed for certain desirable features to work properly. This installs all the required dependencies:

1. Install the `easy_install` program:

    wget http://python-distribute.org/distribute_setup.pysudo python distribute_setup.py

2. Install Cython

    sudo easy_install -U Cython

3. You can then install from the source code as follows:

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

Binary installation

If you have installed pip as described in the *Python installation* section, installing pandas is as simple as the following:

pip install pandas

Windows

The following methods describe the installation in the Windows environment.

Binary Installation

Make sure that `numpy`, `python-dateutil`, and `pytz` are installed first. The following commands need to be run for each of these modules:

*   For python-dateutil:

    C:\Python27\Scripts\pip install python-dateutil

*   For pytz:

    C:\Python27\Scripts\pip install pytz

Install from the binary download, and run the binary for your version of Windows from `https://pypi.python.org/pypi/pandas`. For example, if your processor is an AMD64, you can download and install pandas by using the following commands:

1. Download the following file: (applies to pandas 0.16)

   ```
   pandas-0.16.1-cp26-none-win_amd64.whl (md5)
   ```

2. Install the downloaded file via pip:

   ```
   pip install
   pandas-0.16.1-cp26-none-win_amd64.whl
   ```

To test the install, run Python and type the following on the command prompt:

import pandas

If it returns with no errors then the installation was successful.

Source installation

The steps here explain the installation completely:

1. Install the `MinGW` compiler by following the instructions in the documentation titled *Appendix: Installing MinGW on Windows* at `http://docs.cython.org/src/tutorial/appendix.html`.

2. Make sure that the `MingW` binary location is added to the `PATH` variable, that has `C:\MingW\bin` appended to it.

3. Install `Cython` and `Numpy`.

   `Numpy` can be downloaded and installed from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy`.

   `Cython` can be downloaded and installed from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#cython`

The steps to install `Cython` are as follows:

- Installation via Pip:

  ```
  C:\Python27\Scripts\pip install Cython
  ```

- Direct Download:

  1. Download and install the pandas source from GitHub:
     `http://github.com/pydata/pandas`.

  2. You can simply download and extract the zip file to a suitable folder.

  3. Change to the folder containing the pandas download `to` `C:\python27\python` and run `setup.py` install.

4. Sometimes, you may obtain the following error when running `setup.py`:

   distutils.errors.DistutilsError: Setup script exited with error:

   Unable to find vcvarsall.bat

This may have to do with not properly specifying `mingw` as the compiler. Check that you have followed all the steps again.

> Installing pandas on Windows from the source is prone to many bugs and errors and is not really recommended.

IPython

Interactive Python (**IPython**) is a tool that is very useful for using Python for data analysis, and a brief description of the installation steps is provided here. IPython provides an interactive environment that is much more useful than the standard Python prompt. Its features include the following:

- Tab completion to help the user do data exploration.
- Comprehensive Help functionality using `object_name?` to print details about objects.
- Magic functions that enable the user to run operating system commands within IPython, and run a Python script and load its data into the IPython environment by using the `%run` magic command.
- History functionality via the `_`, `__`, and `__` variables, the `%history` and other magic functions, and the up and down arrow keys.

> For more information, see the documentation at `http://bit.ly/1Is4zIW`.

IPython Notebook

IPython Notebook is the web-enabled version of IPython. It enables the user to combine code, numerical computation, and display graphics and rich media in a single document, the notebook. Notebooks can be shared with colleagues and converted to the HTML/PDF formats. For more information, refer to the documentation titled *The IPython Notebook* at `http://ipython.org/notebook.html`. Here is an illustration:

PyMC Pandas Example

This example project shows how to fit a fixed effects Poisson model with PyMC. It uses pandas Series and DataFrame objects to store data in a classy way.

```
In [1]: import pylab as pl
        import pymc as mc
        import pandas
```

1. Simulate Noisy Data

```
In [2]: # simulate data with known distribution

        N = 100
        X = pandas.DataFrame({'constant': pl.ones(N), 'cov_1': pl.randn(N)})

        beta_true = pandas.Series(dict(constant=100., cov_1=20.))
        mu_true = pl.dot(X, beta_true)

        Y = mc.rpoisson(mu_true)
```

```
In [3]: # explore the data a little bit graphically

        pl.figure(figsize=(11,4.25))

        pl.subplot(1,2,1)
        pl.hist(Y)
        pl.xlabel('Observed Count')

        pl.subplot(1,2,2)
        pl.plot(X['cov_1'], Y, '.')
        pl.xlabel('Covariate 1')
        pl.ylabel('Observed Count')
```

```
Out[3]: <matplotlib.text.Text at 0xb8fddec>
```

The preceding image of **PYMC Pandas Example** is taken from `http://healthyalgorithms.files.wordpress.com/2012/01/pymc-pandas-example.png`.
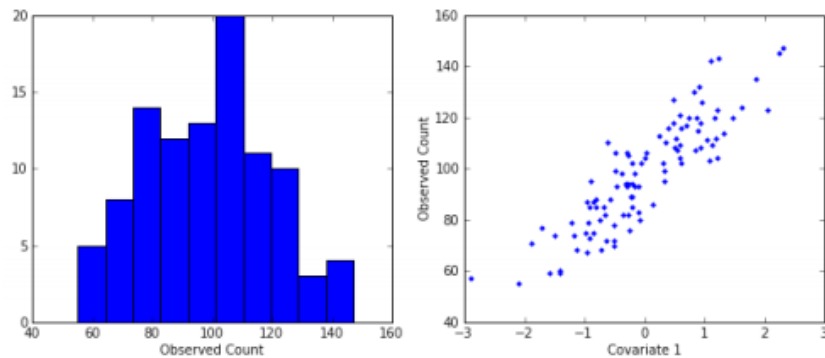
IPython installation

The recommended method to install IPython would be to use a third-party package such as Continuum's Anaconda or Enthought Canopy.

Linux

Assuming that pandas and other tools for scientific computing have been installed as per the instructions, the following one-line commands should suffice:

For Ubuntu/Debian, use

```
sudo apt-get install ipython-notebook
```

For Fedora, use

```
sudo yum install python-ipython-notebook
```

If you have `pip` and `setuptools` installed, you can also install it via the following command for Linux/Mac platforms:

```
sudo pip install ipython
```

Windows

IPython requires `setuptools` on Windows, and the `PyReadline` library. `PyReadline` is a Python implementation of the GNU `readline` library. To install IPython on Windows, perform the following steps:

1. Install `setuptools` as detailed in the preceding section.
2. Install `pyreadline` by downloading the MS Windows installer from PyPI Readline package page at `https://pypi.python.org/pypi/pyreadline`.
3. Download and run the IPython Installer from the GitHub IPython download location: `https://github.com/ipython/ipython/downloads`.

For more information, see the IPython installation page at `http://bit.ly/1MkCZhC`.

Mac OS X

IPython can be installed on Mac OS X by using `pip` or `setuptools`. It also needs the `readline` and `zeromq` library, which are best installed by using Homebrew. The steps are as follows:

```
brew install readline
brew install zeromq
pip install ipython pyzmq tornado pygments
```

The `pyzmq`, `tornado`, and `pygments` modules are necessary to obtain the full graphical functionality of IPython Notebook. For more information, see the documentation titled *Setup IPython Notebook and Pandas for OSX* at `http://bit.ly/1JG0wKA`.

Install via Anaconda (for Linux/Mac OS X)

Assuming that Anaconda is already installed, simply run the following commands to update IPython to the latest version:

```
conda update conda
conda update ipython
```

Wakari by Continuum Analytics

If the user is not quite ready to install IPython, an alternative would be to use IPython in the cloud. Enter **Wakari**, a cloud-based analytics solution that provides full support for IPython notebooks hosted on Continuum's servers. It allows the user to create, edit, save, and share IPython notebooks all within a browser on the cloud. More details can be found at `http://continuum.io/wakari`.

Virtualenv

Virtualenv is a tool that is used to create isolated Python environments. It can be useful if you wish to work in an environment to test out the latest version of pandas without affecting the standard Python build.

Virtualenv installation and usage

I would only recommend installing Virtualenv if you decide not to install and use the `Anaconda` package, as this already provides the Virtualenv functionality. The brief steps are as follows:

1. Install via `pip`:

    ```
    pip install virtualenv
    ```

2. Use of Virtualenv

    ° Create a virtual environment by using the following command:

    ```
    virtualenv newEnv
    ```

    ° Activate the virtual environment by using the following command:

    ```
    source newEnv/bin/activate
    ```

    ° Deactivate the virtual environment and go back to the standard Python environment by using the following command:

    ```
    deactivate
    ```

For more information on this, you can go through the documentation titled *Virtual Environments* at `http://docs.python-guide.org/en/latest/dev/virtualenvs/`.

Summary

There are two main versions of Python available: Python 2.7.x and Python 3.x. At the moment, Python 2.7.x is preferable for data analysis and numerical computing as it is more mature. The pandas library requires a few dependencies in order to be setup correctly – NumPy, SciPy, and matplotlib to name a few. There are a myriad number of ways to install pandas – the recommended method is to install one of the third-party distributions that include pandas. Distributions include Anaconda by Continuum, Enthough Canopy, WinPython, and Python(x,y). Installation of the IPython package is highly recommended as it provides a rich, highly interactive environment for data analysis.

Thus, setting up our environment for learning pandas involves installing a suitable version of Python, installing pandas and its dependent modules, and setting up some useful tools such as IPython. To re-emphasize, I strongly advise readers to do themselves a favor and make their task easier by installing a third-party distribution, such as Anaconda or Enthought, so as to get their environment up and running trouble-free in the shortest possible timeframe. In our next chapter, we will start diving into pandas directly as we take a look at its key features.

3
The pandas Data Structures

This chapter is one of the most important ones in this book. We will now begin to dive into the meat and bones of pandas. We start by taking a tour of NumPy `ndarrays`, a data structure not in pandas but NumPy. Knowledge of NumPy `ndarrays` is useful as it forms the foundation for the pandas data structures. Another key benefit of NumPy arrays is that they execute what is known as *vectorized* operations, which are operations that require traversing/looping on a Python array, much faster.

The topics we will cover in this chapter include the following:

- Tour of `numpy.ndarray` data structure.
- The `pandas.Series` **1-dimensional** (**1D**) pandas data structure
- The `pandas.DatcaFrame` **2-dimensional** (**2D**) pandas tabular data structure
- The `pandas.Panel` **3-dimensional** (**3D**) pandas data structure

In this chapter, I will present the material via numerous examples using IPython, a browser-based interface that allows the user to type in commands interactively to the Python interpreter. Instructions for installing IPython are provided in the previous chapter.

NumPy ndarrays

The NumPy library is a very important package used for numerical computing with Python. Its primary features include the following:

- The type `numpy.ndarray`, a homogenous multidimensional array
- Access to numerous mathematical functions – linear algebra, statistics, and so on
- Ability to integrate C, C++, and Fortran code

For more information about NumPy, see `http://www.numpy.org`.

The primary data structure in NumPy is the array class `ndarray`. It is a homogeneous multi-dimensional (n-dimensional) table of elements, which are indexed by integers just as a normal array. However, `numpy.ndarray` (also known as `numpy.array`) is different from the standard Python `array.array` class, which offers much less functionality. More information on the various operations is provided at `http://scipy-lectures.github.io/intro/numpy/array_object.html`.

NumPy array creation

NumPy arrays can be created in a number of ways via calls to various NumPy methods.

NumPy arrays via numpy.array

NumPy arrays can be created via the `numpy.array` constructor directly:

```
In [1]: import numpy as np
In [2]: ar1=np.array([0,1,2,3])# 1 dimensional array
In [3]: ar2=np.array ([[0,3,5],[2,8,7]]) # 2D array
In [4]: ar1
Out[4]: array([0, 1, 2, 3])
In [5]: ar2
Out[5]: array([[0, 3, 5],
               [2, 8, 7]])
```

The shape of the array is given via `ndarray.shape`:

```
In [5]: ar2.shape
Out[5]: (2, 3)
```

The number of dimensions is obtained using `ndarray.ndim`:

```
In [7]: ar2.ndim
Out[7]: 2
```

NumPy array via numpy.arange

`ndarray.arange` is the NumPy version of Python's range function:
In [10]: # produces the integers from 0 to 11, not inclusive of 12

```
        ar3=np.arange(12); ar3
Out[10]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
In [11]: # start, end (exclusive), step size
        ar4=np.arange(3,10,3); ar4
Out[11]: array([3, 6, 9])
```

NumPy array via numpy.linspace

`ndarray.linspace` generates linear evenly spaced elements between the start and the end:

```
In [13]:# args - start element,end element, number of elements
        ar5=np.linspace(0,2.0/3,4); ar5
Out[13]:array([ 0.,  0.22222222,  0.44444444,  0.66666667])
```

NumPy array via various other functions

These functions include `numpy.zeros`, `numpy.ones`, `numpy.eye`, `nrandom.rand`, `numpy.random.randn`, and `numpy.empty`.

The argument must be a tuple in each case. For the 1D array, you can just specify the number of elements, no need for a tuple.

numpy.ones

The following command line explains the function:

```
In [14]:# Produces 2x3x2 array of 1's.
        ar7=np.ones((2,3,2)); ar7
Out[14]: array([[[ 1.,  1.],
                [ 1.,  1.],
                [ 1.,  1.]],
               [[ 1.,  1.],
                [ 1.,  1.],
                [ 1.,  1.]]])
```

numpy.zeros

The following command line explains the function:

```
In [15]:# Produce 4x2 array of zeros.
            ar8=np.zeros((4,2));ar8
```

```
Out[15]: array([[ 0.,   0.],
          [ 0.,   0.],
           [ 0.,   0.],
           [ 0.,   0.]])
```

numpy.eye

The following command line explains the function:

```
In [17]:# Produces identity matrix
          ar9 = np.eye(3);ar9
Out[17]: array([[ 1.,   0.,   0.],
           [ 0.,   1.,   0.],
           [ 0.,   0.,   1.]])
```

numpy.diag

The following command line explains the function:

```
In [18]: # Create diagonal array
        ar10=np.diag((2,1,4,6));ar10
Out[18]: array([[2, 0, 0, 0],
           [0, 1, 0, 0],
          [0, 0, 4, 0],
          [0, 0, 0, 6]])
```

numpy.random.rand

The following command line explains the function:

```
In [19]: # Using the rand, randn functions
        # rand(m) produces uniformly distributed random numbers with
range 0 to m
         np.random.seed(100)    # Set seed
         ar11=np.random.rand(3); ar11
Out[19]: array([ 0.54340494,  0.27836939,  0.42451759])
In [20]: # randn(m) produces m normally distributed (Gaussian) random
numbers
         ar12=np.random.rand(5); ar12
Out[20]: array([ 0.35467445, -0.78606433, -0.2318722 ,    0.20797568,
0.93580797])
```

numpy.empty

Using `np.empty` to create an uninitialized array is a cheaper and faster way to allocate an array, rather than using `np.ones` or `np.zeros` (`malloc` versus. `cmalloc`). However, you should only use it if you're sure that all the elements will be initialized later:

```
In [21]: ar13=np.empty((3,2)); ar13
Out[21]: array([[ -2.68156159e+154,   1.28822983e-231],
                [  4.22764845e-307,   2.78310358e-309],
                [  2.68156175e+154,   4.17201483e-309]])
```

numpy.tile

The `np.tile` function allows one to construct an array from a smaller array by repeating it several times on the basis of a parameter:

```
In [334]: np.array([[1,2],[6,7]])
Out[334]: array([[1, 2],
                 [6, 7]])
In [335]: np.tile(np.array([[1,2],[6,7]]),3)
Out[335]: array([[1, 2, 1, 2, 1, 2],
                 [6, 7, 6, 7, 6, 7]])
In [336]: np.tile(np.array([[1,2],[6,7]]),(2,2))
Out[336]: array([[1, 2, 1, 2],
                 [6, 7, 6, 7],
                 [1, 2, 1, 2],
                 [6, 7, 6, 7]])
```

NumPy datatypes

We can specify the type of contents of a numeric array by using the `dtype` parameter:

```
In [50]: ar=np.array([2,-1,6,3],dtype='float'); ar
Out[50]: array([ 2., -1.,  6.,  3.])
In [51]: ar.dtype
Out[51]: dtype('float64')
In [52]: ar=np.array([2,4,6,8]); ar.dtype
Out[52]: dtype('int64')
In [53]: ar=np.array([2.,4,6,8]); ar.dtype
Out[53]: dtype('float64')
```

The default `dtype` in NumPy is `float`. In the case of strings, `dtype` is the length of the longest string in the array:

```
In [56]: sar=np.array(['Goodbye','Welcome','Tata','Goodnight']); sar.
dtype
Out[56]: dtype('S9')
```

You cannot create variable-length strings in NumPy, since NumPy needs to know how much space to allocate for the string. `dtypes` can also be Boolean values, complex numbers, and so on:

```
In [57]: bar=np.array([True, False, True]); bar.dtype
Out[57]: dtype('bool')
```

The datatype of `ndarray` can be changed in much the same way as we cast in other languages such as Java or C/C++. For example, `float` to `int` and so on. The mechanism to do this is to use the `numpy.ndarray.astype()` function. Here is an example:

```
In [3]: f_ar = np.array([3,-2,8.18])
        f_ar
Out[3]: array([ 3.  , -2.  ,  8.18])
In [4]: f_ar.astype(int)
Out[4]: array([ 3, -2,  8])
```

More information on casting can be found in the official documentation at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.astype.html`.

NumPy indexing and slicing

Array indices in NumPy start at `0`, as in languages such as Python, Java, and C++ and unlike in Fortran, Matlab, and Octave, which start at `1`. Arrays can be indexed in the standard way as we would index into any other Python sequences:

```
# print entire array, element 0, element 1, last element.
In [36]: ar = np.arange(5); print ar; ar[0], ar[1], ar[-1]
[0 1 2 3 4]
Out[36]: (0, 1, 4)
# 2nd, last and 1st elements
In [65]: ar=np.arange(5); ar[1], ar[-1], ar[0]
Out[65]: (1, 4, 0)
```

Arrays can be reversed using the `::-1` idiom as follows:

```
In [24]: ar=np.arange(5); ar[::-1]
Out[24]: array([4, 3, 2, 1, 0])
```

Multi-dimensional arrays are indexed using tuples of integers:

```
In [71]: ar = np.array([[2,3,4],[9,8,7],[11,12,13]]); ar
Out[71]: array([[ 2,  3,  4],
                [ 9,  8,  7],
                [11, 12, 13]])
In [72]: ar[1,1]
Out[72]: 8
```

Here, we set the entry at `row1` and `column1` to `5`:

```
In [75]: ar[1,1]=5; ar
Out[75]: array([[ 2,  3,  4],
                [ 9,  5,  7],
                [11, 12, 13]])
```

Retrieve row 2:

```
In [76]:  ar[2]
Out[76]: array([11, 12, 13])
In [77]: ar[2,:]
Out[77]: array([11, 12, 13])
```

Retrieve column 1:

```
In [78]: ar[:,1]
Out[78]: array([ 3,  5, 12])
```

If an index is specified that is out of bounds of the range of an array, `IndexError` will be raised:

```
In [6]: ar = np.array([0,1,2])
In [7]: ar[5]
   ---------------------------------------------------------------------
-----
   IndexError                       Traceback (most recent call last)
  <ipython-input-7-8ef7e0800b7a> in <module>()
   ----> 1 ar[5]
      IndexError: index 5 is out of bounds for axis 0 with size 3
```

Thus, for 2D arrays, the first dimension denotes rows and the second dimension, the columns. The colon (`:`) denotes selection across all elements of the dimension.

Array slicing

Arrays can be sliced using the following syntax:
`ar[startIndex: endIndex: stepValue]`.

```
In [82]: ar=2*np.arange(6); ar
Out[82]: array([ 0,  2,  4,  6,  8, 10])
In [85]: ar[1:5:2]
Out[85]: array([2, 6])
```

Note that if we wish to include the `endIndex` value, we need to go above it, as follows:

```
In [86]: ar[1:6:2]
Out[86]: array([ 2,  6, 10])
```

Obtain the first n-elements using `ar[:n]`:

```
In [91]: ar[:4]
Out[91]: array([0, 2, 4, 6])
```

The implicit assumption here is that `startIndex=0, step=1`.

Start at element 4 until the end:

```
In [92]: ar[4:]
Out[92]: array([ 8, 10])
```

Slice array with `stepValue=3`:

```
In [94]: ar[::3]
Out[94]: array([0, 6])
```

To illustrate the scope of indexing in NumPy, let us refer to this illustration, which is taken from a NumPy lecture given at SciPy 2013 and can be found at `http://bit.ly/1GxCDpC`:

```
>>> a[0,3:5]
array([3,4])


>>> a[4:,4:]
array([[44, 45],
       [54, 55]])


>>> a[:,2]
array([2,12,22,32,42,52])


>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Let us now examine the meanings of the expressions in the preceding image:

- The expression a[0,3:5] indicates the start at row 0, and columns 3-5, where column 5 is not included.

- In the expression a[4:,4:], the first 4 indicates the start at row 4 and will give all columns, that is, the array [[40, 41,42,43,44,45] [50,51,52,53,54,55]]. The second 4 shows the cutoff at the start of column 4 to produce the array [[44, 45], [54, 55]].

- The expression a[:,2] gives all rows from column 2.

- Now, in the last expression a[2::2,::2], 2::2 indicates that the start is at row 2 and the step value here is also 2. This would give us the array [[20, 21, 22, 23, 24, 25], [40, 41, 42, 43, 44, 45]]. Further, ::2 specifies that we retrieve columns in steps of 2, producing the end result array ([[20, 22, 24], [40, 42, 44]]).

Assignment and slicing can be combined as shown in the following code snippet:

```
In [96]: ar
Out[96]: array([ 0,  2,  4,  6,  8, 10])
In [100]: ar[:3]=1; ar
Out[100]: array([ 1,  1,  1,  6,  8, 10])
In [110]: ar[2:]=np.ones(4);ar
Out[110]: array([1, 1, 1, 1, 1, 1])
```

Array masking

Here, NumPy arrays can be used as masks to select or filter out elements of the original array. For example, see the following snippet:

```
In [146]: np.random.seed(10)
          ar=np.random.random_integers(0,25,10); ar
Out[146]: array([ 9,  4, 15,  0, 17, 25, 16, 17,  8,  9])
In [147]: evenMask=(ar % 2==0); evenMask
Out[147]: array([False,  True, False,  True, False, False,  True, False,
True, False], dtype=bool)
In [148]: evenNums=ar[evenMask]; evenNums
Out[148]: array([ 4,  0, 16,  8])
```

In the following example, we randomly generate an array of 10 integers between 0 and 25. Then, we create a Boolean mask array that is used to filter out only the even numbers. This masking feature can be very useful, say for example, if we wished to eliminate missing values, by replacing them with a default value. Here, the missing value '' is replaced by 'USA' as the default country. Note that '' is also an empty string:

```
In [149]: ar=np.array(['Hungary','Nigeria',
                       'Guatemala','','Poland',
                       '','Japan']); ar
Out[149]: array(['Hungary', 'Nigeria', 'Guatemala',
                 '', 'Poland', '', 'Japan'],
                 dtype='|S9')
In [150]: ar[ar=='']='USA'; ar
Out[150]: array(['Hungary', 'Nigeria', 'Guatemala',
  'USA', 'Poland', 'USA', 'Japan'], dtype='|S9')
```

Arrays of integers can also be used to index an array to produce another array. Note that this produces multiple values; hence, the output must be an array of type `ndarray`. This is illustrated in the following snippet:

```
In [173]: ar=11*np.arange(0,10); ar
Out[173]: array([ 0, 11, 22, 33, 44, 55, 66, 77, 88, 99])
In [174]: ar[[1,3,4,2,7]]
Out[174]: array([11, 33, 44, 22, 77])
```

In the preceding code, the selection object is a list and elements at indices 1, 3, 4, 2, and 7 are selected. Now, assume that we change it to the following:

```
In [175]: ar[1,3,4,2,7]
```

We get an `IndexError` error since the array is 1D and we're specifying too many indices to access it.

```
IndexError                Traceback (most recent call last)
<ipython-input-175-adbcbe3b3cdc> in <module>()
----> 1 ar[1,3,4,2,7]


IndexError: too many indices
```

This assignment is also possible with array indexing, as follows:

```
In [176]: ar[[1,3]]=50; ar
Out[176]: array([ 0, 50, 22, 50, 44, 55, 66, 77, 88, 99])
```

When a new array is created from another array by using a list of array indices, the new array has the same shape.

Complex indexing

Here, we illustrate the use of complex indexing to assign values from a smaller array into a larger one:

```
In [188]: ar=np.arange(15); ar
Out[188]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
14])


In [193]: ar2=np.arange(0,-10,-1)[::-1]; ar2
Out[193]: array([-9, -8, -7, -6, -5, -4, -3, -2, -1,  0])
```

Slice out the first 10 elements of `ar`, and replace them with elements from `ar2`, as follows:

```
In [194]: ar[:10]=ar2; ar
Out[194]: array([-9, -8, -7, -6, -5, -4, -3, -2, -1,  0, 10, 11, 12, 13,
14])
```

Copies and views

A view on a NumPy array is just a particular way of portraying the data it contains. Creating a view does not result in a new copy of the array, rather the data it contains may be arranged in a specific order, or only certain data rows may be shown. Thus, if data is replaced on the underlying array's data, this will be reflected in the view whenever the data is accessed via indexing.

The initial array is not copied into the memory during slicing and is thus more efficient. The `np.may_share_memory` method can be used to see if two arrays share the same memory block. However, it should be used with caution as it may produce false positives. Modifying a view modifies the original array:

```
In [118]:ar1=np.arange(12); ar1
Out[118]:array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])


In [119]:ar2=ar1[::2]; ar2
Out[119]: array([ 0,  2,  4,  6,  8, 10])


In [120]: ar2[1]=-1; ar1
Out[120]: array([ 0,  1, -1,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

To force NumPy to copy an array, we use the `np.copy` function. As we can see in the following array, the original array remains unaffected when the copied array is modified:

```
In [124]: ar=np.arange(8);ar
Out[124]: array([0, 1, 2, 3, 4, 5, 6, 7])


In [126]: arc=ar[:3].copy(); arc
Out[126]: array([0, 1, 2])


In [127]: arc[0]=-1; arc
Out[127]: array([-1,  1,  2])


In [128]: ar
Out[128]: array([0, 1, 2, 3, 4, 5, 6, 7])
```

Operations

Here, we present various operations in NumPy.

Basic operations

Basic arithmetic operations work element-wise with scalar operands. They are - +, -, *, /, and **.

```
In [196]: ar=np.arange(0,7)*5; ar
Out[196]: array([ 0,  5, 10, 15, 20, 25, 30])


In [198]: ar=np.arange(5) ** 4 ; ar
Out[198]: array([  0,   1,  16,  81, 256])


In [199]: ar ** 0.5
Out[199]: array([ 0.,  1.,  4.,  9., 16.])
```

Operations also work element-wise when another array is the second operand as follows:

```
In [209]: ar=3+np.arange(0, 30,3); ar
Out[209]: array([ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30])


In [210]: ar2=np.arange(1,11); ar2
Out[210]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Here, in the following snippet, we see element-wise subtraction, division, and multiplication:

```
In [211]: ar-ar2
Out[211]: array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20])


In [212]: ar/ar2
Out[212]: array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3])


In [213]: ar*ar2
Out[213]: array([  3,  12,  27,  48,  75, 108, 147, 192, 243, 300])
```

It is much faster to do this using NumPy rather than pure Python. The `%timeit` function in IPython is known as a magic function and uses the Python `timeit` module to time the execution of a Python statement or expression, explained as follows:

```
In [214]: ar=np.arange(1000)
          %timeit ar**3
```

```
           100000 loops, best of 3: 5.4 µs per loop


In [215]:ar=range(1000)
         %timeit [ar[i]**3 for i in ar]
         1000 loops, best of 3: 199 µs per loop
```

Array multiplication is not the same as matrix multiplication; it is element-wise, meaning that the corresponding elements are multiplied together. For matrix multiplication, use the dot operator. For more information refer to `http://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html`.

```
In [228]: ar=np.array([[1,1],[1,1]]); ar
Out[228]: array([[1, 1],
                 [1, 1]])


In [230]: ar2=np.array([[2,2],[2,2]]); ar2
Out[230]: array([[2, 2],
                 [2, 2]])


In [232]: ar.dot(ar2)
Out[232]: array([[4, 4],
                 [4, 4]])
```

Comparisons and logical operations are also element-wise:

```
In [235]: ar=np.arange(1,5); ar
Out[235]: array([1, 2, 3, 4])


In [238]: ar2=np.arange(5,1,-1);ar2
Out[238]: array([5, 4, 3, 2])


In [241]: ar < ar2
Out[241]: array([ True,  True, False, False], dtype=bool)


In [242]: l1 = np.array([True,False,True,False])
          l2 = np.array([False,False,True, False])
          np.logical_and(l1,l2)
Out[242]: array([False, False,  True, False], dtype=bool)
```

Other NumPy operations such as log, sin, cos, and exp are also element-wise:

```
In [244]: ar=np.array([np.pi, np.pi/2]); np.sin(ar)
Out[244]: array([  1.22464680e-16,   1.00000000e+00])
```

Note that for element-wise operations on two NumPy arrays, the two arrays *must* have the same shape, else an error will result since the arguments of the operation must be the corresponding elements in the two arrays:

```
In [245]: ar=np.arange(0,6); ar
Out[245]: array([0, 1, 2, 3, 4, 5])


In [246]: ar2=np.arange(0,8); ar2
Out[246]: array([0, 1, 2, 3, 4, 5, 6, 7])


In [247]: ar*ar2

        ------------------------------------------------------------
------------

        ValueError                                Traceback (most
recent call last)
        <ipython-input-247-2c3240f67b63> in <module>()
        ----> 1 ar*ar2
        ValueError: operands could not be broadcast together with
shapes (6) (8)
```

Further, NumPy arrays can be transposed as follows:

```
In [249]: ar=np.array([[1,2,3],[4,5,6]]); ar
Out[249]: array([[1, 2, 3],
                 [4, 5, 6]])


In [250]:ar.T
Out[250]:array([[1, 4],
                [2, 5],
                [3, 6]])


In [251]: np.transpose(ar)
Out[251]: array([[1, 4],
                 [2, 5],
                 [3, 6]])
```

Suppose we wish to compare arrays not element-wise, but array-wise. We could achieve this as follows by using the `np.array_equal` operator:

```
In [254]: ar=np.arange(0,6)
          ar2=np.array([0,1,2,3,4,5])
          np.array_equal(ar, ar2)
Out[254]: True
```

Here, we see that a single Boolean value is returned instead of a Boolean array. The value is `True` only if *all* the corresponding elements in the two arrays match. The preceding expression is equivalent to the following:

```
In [24]: np.all(ar==ar2)
Out[24]: True
```

Reduction operations

Operators such as `np.sum` and `np.prod` perform reduces on arrays; that is, they combine several elements into a single value:

```
In [257]: ar=np.arange(1,5)
          ar.prod()
Out[257]: 24
```

In the case of multi-dimensional arrays, we can specify whether we want the reduction operator to be applied row-wise or column-wise by using the axis parameter:

```
In [259]: ar=np.array([np.arange(1,6),np.arange(1,6)]);ar
Out[259]: array([[1, 2, 3, 4, 5],
                 [1, 2, 3, 4, 5]])
# Columns
In [261]: np.prod(ar,axis=0)
Out[261]: array([ 1,  4,  9, 16, 25])
# Rows
In [262]: np.prod(ar,axis=1)
Out[262]: array([120, 120])
```

In the case of multi-dimensional arrays, not specifying an axis results in the operation being applied to all elements of the array as explained in the following example:

```
In [268]: ar=np.array([[2,3,4],[5,6,7],[8,9,10]]); ar.sum()
```

```
Out[268]: 54


In [269]: ar.mean()
Out[269]: 6.0
In [271]: np.median(ar)
Out[271]: 6.0
```

Statistical operators

These operators are used to apply standard statistical operations to a NumPy array. The names are self-explanatory: `np.std()`, `np.mean()`, `np.median()`, and `np.cumsum()`.

```
In [309]: np.random.seed(10)
          ar=np.random.randint(0,10, size=(4,5));ar
Out[309]: array([[9, 4, 0, 1, 9],
                 [0, 1, 8, 9, 0],
                 [8, 6, 4, 3, 0],
                 [4, 6, 8, 1, 8]])
In [310]: ar.mean()
Out[310]: 4.4500000000000002


In [311]: ar.std()
Out[311]: 3.4274626183227732


In [312]: ar.var(axis=0)  # across rows
Out[312]: array([ 12.6875,   4.1875,  11.   ,  10.75 ,  18.1875])


In [313]: ar.cumsum()
Out[313]: array([ 9, 13, 13, 14, 23, 23, 24, 32, 41, 41, 49, 55,
                 59, 62, 62, 66, 72, 80, 81, 89])
```

Logical operators

Logical operators can be used for array comparison/checking. They are as follows:

- `np.all()`: This is used for element-wise and all of the elements
- `np.any()`: This is used for element-wise or all of the elements

Generate a random 4 × 4 array of `ints` and check if any element is divisible by 7 and if all elements are less than 11:

```
In [320]: np.random.seed(100)
          ar=np.random.randint(1,10, size=(4,4));ar
Out[320]: array([[9, 9, 4, 8],
                 [8, 1, 5, 3],
                 [6, 3, 3, 3],
                 [2, 1, 9, 5]])


In [318]: np.any((ar%7)==0)
Out[318]: False


In [319]: np.all(ar<11)
Out[319]: True
```

Broadcasting

In broadcasting, we make use of NumPy's ability to combine arrays that don't have the same exact shape. Here is an example:

```
In [357]: ar=np.ones([3,2]); ar
Out[357]: array([[ 1.,  1.],
                 [ 1.,  1.],
                 [ 1.,  1.]])


In [358]: ar2=np.array([2,3]); ar2
Out[358]: array([2, 3])


In [359]: ar+ar2
Out[359]: array([[ 3.,  4.],
                 [ 3.,  4.],
                 [ 3.,  4.]])
```

Thus, we can see that `ar2` is *broadcasted* across the rows of `ar` by adding it to each row of `ar` producing the preceding result. Here is another example, showing that broadcasting works across dimensions:

```
In [369]: ar=np.array([[23,24,25]]); ar
Out[369]: array([[23, 24, 25]])
```

```
In [368]: ar.T
Out[368]: array([[23],
                 [24],
                 [25]])
In [370]: ar.T+ar
Out[370]: array([[46, 47, 48],
                 [47, 48, 49],
                 [48, 49, 50]])
```

Here, both row and column arrays were broadcasted and we ended up with a
3 × 3 array.

Array shape manipulation

There are a number of steps for the shape manipulation of arrays.

Flattening a multi-dimensional array

The `np.ravel()` function allows you to flatten a multi-dimensional array as follows:

```
In [385]: ar=np.array([np.arange(1,6), np.arange(10,15)]); ar
Out[385]: array([[ 1,  2,  3,  4,  5],
                 [10, 11, 12, 13, 14]])


In [386]: ar.ravel()
Out[386]: array([ 1,  2,  3,  4,  5, 10, 11, 12, 13, 14])


In [387]: ar.T.ravel()
Out[387]: array([ 1, 10,  2, 11,  3, 12,  4, 13,  5, 14])
```

You can also use `np.flatten`, which does the same thing, except that it returns a
copy while `np.ravel` returns a view.

Reshaping

The reshape function can be used to change the shape of or unflatten an array:

```
In [389]: ar=np.arange(1,16);ar
Out[389]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
15])
In [390]: ar.reshape(3,5)
```

```
Out[390]: array([[ 1,  2,  3,  4,  5],
                 [ 6,  7,  8,  9, 10],
                 [11, 12, 13, 14, 15]])
```

The `np.reshape` function returns a view of the data, meaning that the underlying array remains unchanged. In special cases, however, the shape cannot be changed without the data being copied. For more details on this, see the documentation at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html`.

Resizing

There are two resize operators, `numpy.ndarray.resize`, which is an `ndarray` operator that resizes in place, and `numpy.resize`, which returns a new array with the specified shape. Here, we illustrate the `numpy.ndarray.resize` function:

```
In [408]: ar=np.arange(5); ar.resize((8,));ar
Out[408]: array([0, 1, 2, 3, 4, 0, 0, 0])
```

Note that this function only works if there are no other references to this array; else, `ValueError` results:

```
In [34]: ar=np.arange(5);
         ar
Out[34]: array([0, 1, 2, 3, 4])
In [35]: ar2=ar
In [36]: ar.resize((8,));
------------------------------------------------------------------------
---
ValueError                                Traceback (most recent call
last)
<ipython-input-36-394f7795e2d1> in <module>()
----> 1 ar.resize((8,));


ValueError: cannot resize an array that references or is referenced by
another array in this way.  Use the resize function
```

The way around this is to use the `numpy.resize` function instead:

```
In [38]: np.resize(ar,(8,))
Out[38]: array([0, 1, 2, 3, 4, 0, 1, 2])
```

Adding a dimension

The `np.newaxis` function adds an additional dimension to an array:

```
In [377]: ar=np.array([14,15,16]); ar.shape
Out[377]: (3,)
In [378]: ar
Out[378]: array([14, 15, 16])
In [379]: ar=ar[:, np.newaxis]; ar.shape
Out[379]: (3, 1)
In [380]: ar
Out[380]: array([[14],
                 [15],
                 [16]])
```

Array sorting

Arrays can be sorted in various ways.

1. Sort the array along an axis; first, let's discuss this along the y-axis:

```
In [43]: ar=np.array([[3,2],[10,-1]])
         ar
Out[43]: array([[ 3,  2],
                [10, -1]])
In [44]: ar.sort(axis=1)
         ar
Out[44]: array([[ 2,  3],
                [-1, 10]])
```

2. Here, we will explain the sorting along the x-axis:

```
In [45]: ar=np.array([[3,2],[10,-1]])
         ar
Out[45]: array([[ 3,  2],
                [10, -1]])
In [46]: ar.sort(axis=0)
         ar
Out[46]: array([[ 3, -1],
                [10,  2]])
```

3. Sorting by in-place (`np.array.sort`) and out-of-place (`np.sort`) functions.

4. Other operations that are available for array sorting include the following:

    ° `np.min()`: It returns the minimum element in the array

    ° `np.max()`: It returns the maximum element in the array

    ° `np.std()`: It returns the standard deviation of the elements in the array

    ° `np.var()`: It returns the variance of elements in the array

    ° `np.argmin()`: It indices of minimum

    ° `np.argmax()`: It indices of maximum

    ° `np.all()`: It returns element-wise and all of the elements

    ° `np.any()`: It returns element-wise or all of the elements

Data structures in pandas

The pandas was created by Wed McKinney in 2008 as a result of frustrations he encountered while working on time series data in R. It is built on top of NumPy and provides features not available in it. It provides fast, easy-to-understand data structures and helps fill the gap between Python and a language such as R.

A key reference for the various operations I demonstrate here is the official pandas data structure documentation: `http://pandas.pydata.org/pandas-docs/dev/dsintro.html`.

There are three main data structures in pandas:

*   Series
*   DataFrame
*   Panel

Series

Series is really a 1D NumPy array under the hood. It consists of a NumPy array coupled with an array of labels.

Series creation

The general construct for creating a Series data structure is as follows:

```
import pandas as pd

ser=pd.Series(data, index=idx)
```

where data can be one of the following:

- An ndarray
- A Python dictionary
- A scalar value

Using numpy.ndarray

In this case, the index must be the same length as the data. If an index is not specified, the following default index `[0,... n-1]` will be created, where `n` is the length of the data. The following example creates a Series structure of seven random numbers between 0 and 1; the index is not specified:

```
In [466]: import numpy as np
          np.random.seed(100)
          ser=pd.Series(np.random.rand(7)); ser
Out[466]:0    0.543405
         1    0.278369
         2    0.424518
         3    0.844776
         4    0.004719
         5    0.121569
         6    0.670749
         dtype: float64
```

The following example creates a Series structure of the first 5 months of the year with a specified index of month names:

```
In [481]: import calendar as cal
      monthNames=[cal.month_name[i] for i in np.arange(1,6)]
   months=pd.Series(np.arange(1,6),index=monthNames);months

Out[481]: January    1
    February    2
    March       3
    April       4
    May         5
    dtype: int64
```

```
In [482]: months.index
Out[482]: Index([u'January', u'February', u'March', u'April', u'May'],
dtype=object)
```

Using Python dictionary

If the data is a dictionary and an index is provided, the labels will be constructed from it; else, the keys of the dictionary will be used for the labels. The values of the dictionary are used to populate the Series structure.

```
In [486]: currDict={'US' : 'dollar', 'UK' : 'pound',
                    'Germany': 'euro', 'Mexico':'peso',
                    'Nigeria':'naira',
                    'China':'yuan', 'Japan':'yen'}
          currSeries=pd.Series(currDict); currSeries
Out[486]: China        yuan
          Germany      euro
          Japan         yen
          Mexico       peso
          Nigeria     naira
          UK          pound
          US         dollar
          dtype: object
```

The index of a pandas Series structure is of type `pandas.core.index.Index` and can be viewed as an ordered multiset.

In the following case, we specify an index, but the index contains one entry that isn't a key in the corresponding `dict`. The result is that the value for the key is assigned as `NaN`, indicating that it is missing. We will deal with handling missing values in a later section.

```
In [488]: stockPrices = {'GOOG':1180.97,'FB':62.57,
                         'TWTR': 64.50, 'AMZN':358.69,
                         'AAPL':500.6}
          stockPriceSeries=pd.Series(stockPrices,
                         index=['GOOG','FB','YHOO',
                               'TWTR','AMZN','AAPL'],
                         name='stockPrices')
          stockPriceSeries
```

```
Out[488]: GOOG      1180.97

          FB          62.57

          YHOO          NaN

          TWTR        64.50

          AMZN       358.69

          AAPL       500.60

          Name: stockPrices, dtype: float64
```

Note that Series also has a name attribute that can be set as shown in the preceding snippet. The name attribute is useful in tasks such as combining Series objects into a DataFrame structure.

Using scalar values

For scalar data, an index must be provided. The value will be repeated for as many index values as possible. One possible use of this method is to provide a quick and dirty method of initialization, with the Series structure to be filled in later. Let us see how to create a Series using scalar values:

```
In [491]: dogSeries=pd.Series('chihuahua',
                  index=['breed','countryOfOrigin',
                  'name', 'gender'])
          dogSeries
Out[491]: breed              chihuahua

          countryOfOrigin    chihuahua

          name               chihuahua

          gender             chihuahua

          dtype: object
```

Failure to provide an index just results in a scalar value being returned as follows:

```
In [494]: dogSeries=pd.Series('pekingese'); dogSeries

Out[494]: 'pekingese'


In [495]: type(dogSeries)
Out[495]: str
```

Operations on Series

The behavior of Series is very similar to that of `numpy` arrays discussed in a previous section, with one caveat being that an operation such as slicing also slices the index.

Assignment

Values can be set and accessed using the index label in a dictionary-like manner:

```
In [503]: currDict['China']
Out[503]: 'yuan'


In [505]: stockPriceSeries['GOOG']=1200.0
          stockPriceSeries
Out[505]: GOOG     1200.00
          FB          62.57
          YHOO          NaN
          TWTR        64.50
          AMZN       358.69
          AAPL       500.60
          dtype: float64
```

Just as in the case of dict, KeyError is raised if you try to retrieve a missing label:

```
In [506]: stockPriceSeries['MSFT']
KeyError: 'MSFT'
```

This error can be avoided by explicitly using get as follows:

```
In [507]: stockPriceSeries.get('MSFT',np.NaN)
Out[507]: nan
```

In this case, the default value of np.NaN is specified as the value to return when the key does not exist in the Series structure.

Slicing

The slice operation behaves the same way as a NumPy array:

```
In [498]: stockPriceSeries[:4]
Out[498]: GOOG     1180.97
          FB          62.57
          YHOO          NaN
          TWTR        64.50
          dtype: float64
```

Logical slicing also works as follows:

```
In [500]: stockPriceSeries[stockPriceSeries > 100]
Out[500]: GOOG     1180.97
          AMZN      358.69
          AAPL      500.60
          dtype: float64
```

Other operations

Arithmetic and statistical operations can be applied, just as with a NumPy array:

```
In [501]: np.mean(stockPriceSeries)
Out[501]: 433.46600000000001
In [502]: np.std(stockPriceSeries)
Out[502]: 410.50223047384287
```

Element-wise operations can also be performed on series:

```
In [506]: ser
Out[506]: 0     0.543405
          1     0.278369
          2     0.424518
          3     0.844776
          4     0.004719
          5     0.121569
          6     0.670749
          dtype: float64
In [508]: ser*ser
Out[508]: 0     0.295289
          1     0.077490
          2     0.180215
          3     0.713647
          4     0.000022
          5     0.014779
          6     0.449904
          dtype: float64
In [510]: np.sqrt(ser)
Out[510]: 0     0.737160
```

```
       1      0.527607

       2      0.651550

       3      0.919117

       4      0.068694

       5      0.348668

       6      0.818993
dtype: float64
```

An important feature of Series is that the data is automatically aligned on the basis of the label:

```
In [514]: ser[1:]
Out[514]: 1    0.278369

       2      0.424518

       3      0.844776

       4      0.004719

       5      0.121569

       6      0.670749
dtype: float64
In [516]:ser[1:] + ser[:-2]
Out[516]: 0          NaN

       1      0.556739

       2      0.849035

       3      1.689552

       4      0.009438

       5          NaN

       6          NaN
dtype: float64
```

Thus, we can see that for non-matching labels, NaN is inserted. The default behavior is that the union of the indexes is produced for unaligned Series structures. This is preferable as information is preserved rather than lost. We will handle missing values in pandas in a later chapter of the book.

DataFrame

DataFrame is an 2-dimensional labeled array. Its column types can be heterogeneous: that is, of varying types. It is similar to structured arrays in NumPy with mutability added. It has the following properties:

- Conceptually analogous to a table or spreadsheet of data.
- Similar to a NumPy `ndarray` but not a subclass of `np.ndarray`.
- Columns can be of heterogeneous types: float64, int, bool, and so on.
- A DataFrame column is a Series structure.
- It can be thought of as a dictionary of Series structures where both the columns and the rows are indexed, denoted as 'index' in the case of rows and 'columns' in the case of columns.
- It is size mutable: columns can be inserted and deleted.

Every axis in a Series/DataFrame has an index, whether default or not. Indexes are needed for fast lookups as well as proper aligning and joining of data in pandas. The axes can also be named-for example in the form of month for the array of columns Jan Feb Mar... Dec. Here is a representation of an indexed DataFrame, with named columns across and an index column of characters V, W, X, Y, Z:

```
columns nums strs bools decs
index
V           11      cat   True   1.4
W           -6      hat   False  6.9
X           25      bat   False  -0.6
Y            8      mat   True   3.7
Z          -17      sat   False  18.
```

DataFrame Creation

DataFrame is the most commonly used data structure in pandas. The constructor accepts many different types of arguments:

- Dictionary of 1D ndarrays, lists, dictionaries, or Series structures
- 2D NumPy array
- Structured or record ndarray
- Series structures
- Another DataFrame structure

Row label indexes and column labels can be specified along with the data. If they're not specified, they will be generated from the input data in an intuitive fashion, for example, from the keys of `dict.` (in case of column labels) or by using `np.range(n)` in the case of row labels, where `n` corresponds to the number of rows.

Using dictionaries of Series

Here, we create a DataFrame structure by using a dictionary of Series objects.

```
In [97]:stockSummaries={
'AMZN': pd.Series([346.15,0.59,459,0.52,589.8,158.88],
        index=['Closing price','EPS',
               'Shares Outstanding(M)',
               'Beta', 'P/E','Market Cap(B)']),
'GOOG': pd.Series([1133.43,36.05,335.83,0.87,31.44,380.64],
        index=['Closing price','EPS','Shares Outstanding(M)',
               'Beta','P/E','Market Cap(B)']),
'FB': pd.Series([61.48,0.59,2450,104.93,150.92],
      index=['Closing price','EPS','Shares Outstanding(M)',
             'P/E', 'Market Cap(B)']),
'YHOO': pd.Series([34.90,1.27,1010,27.48,0.66,35.36],
        index=['Closing price','EPS','Shares Outstanding(M)',
               'P/E','Beta', 'Market Cap(B)']),
'TWTR':pd.Series([65.25,-0.3,555.2,36.23],
       index=['Closing price','EPS','Shares Outstanding(M)',
              'Market Cap(B)']),
'AAPL':pd.Series([501.53,40.32,892.45,12.44,447.59,0.84],
       index=['Closing price','EPS','Shares Outstanding(M)','P/E',
              'Market Cap(B)','Beta'])}


In [99]: stockDF=pd.DataFrame(stockSummaries); stockDF
Out[99]:
```

	AAPL	AMZN	FB	GOOG	TWTR	YHOO
Beta	0.84	0.52	NaN	0.87	NaN	0.66
Closing price	501.53	346.15	61.48	1133.43	65.25	34.9
EPS	40.32	0.59	0.59	36.05	-0.3	1.27
Market Cap(B)	447.59	158.88	150.92	380.64	36.23	35.36
P/E	12.44	589.8	104.93	31.44	NaN	27.48
Shares Outstanding(M)	892.45	459	2450	335.83	555.2	1010

```
In [100]:stockDF=pd.DataFrame(stockSummaries,
               index=['Closing price','EPS',
                   'Shares Outstanding(M)',
                   'P/E', 'Market Cap(B)','Beta']);stockDF
```

Out [100]:

	AAPL	AMZN	FB	GOOG	TWTR	YHOO
Closing price	501.53	346.15	61.48	1133.43	65.25	34.9
EPS	40.32	0.59	0.59	36.05	-0.3	1.27
Shares Outstanding(M)	892.45	459	2450	NaN	555.2	1010
P/E	12.44	589.8	104.93	31.44	NaN	27.48
Market Cap(B)	447.59	158.88	150.92	380.64	36.23	35.36
Beta	0.84	0.52	NaN	0.87	NaN	0.66

```
In [102]:stockDF=pd.DataFrame(stockSummaries,
               index=['Closing price','EPS',
                   'Shares Outstanding(M)',
                   'P/E', 'Market Cap(B)','Beta'],
               columns=['FB','TWTR','SCNW'])
        stockDF
```

Out [102]:

	FB	TWTR	SCNW
Closing price	61.48	65.25	NaN
EPS	0.59	-0.3	NaN
Shares Outstanding(M)	2450	555.2	NaN
P/E	104.93	NaN	NaN
Market Cap(B)	150.92	36.23	NaN
Beta	NaN	NaN	NaN

The row index labels and column labels can be accessed via the index and column attributes:

```
In [527]: stockDF.index
Out[527]: Index([u'Closing price', u'EPS',
```

```
          u'Shares        Outstanding(M)',
          u'P/E', u'Market Cap(B)', u'Beta'], dtype=object)
In [528]: stockDF.columns
Out[528]: Index([u'AAPL', u'AMZN', u'FB', u'GOOG', u'TWTR',
          u'YHOO'], dtype=object)
```

The source for the preceding data is Google Finance, accessed on 2/3/2014:
`http://finance.google.com`.

Using a dictionary of ndarrays/lists

Here, we create a DataFrame structure from a dictionary of lists. The keys become
the column labels in the DataFrame structure and the data in the list becomes the
column values. Note how the row label indexes are generated using `np.range(n)`.

```
In [529]:algos={'search':['DFS','BFS','Binary Search',
                 'Linear','ShortestPath (Djikstra)'],
      'sorting': ['Quicksort','Mergesort', 'Heapsort',
                  'Bubble Sort', 'Insertion Sort'],
      'machine learning':['RandomForest',
                          'K Nearest Neighbor',
                          'Logistic Regression',
                          'K-Means Clustering',
                          'Linear Regression']}
algoDF=pd.DataFrame(algos); algoDF
Out[529]:
     machine learning   search    sorting
0   RandomForest    DFS    Quicksort
1   K Nearest Neighbor    BFS    Mergesort
2   Logistic Regression   Binary   Search    Heapsort
3   K-Means Clustering       Linear    Bubble Sort
4   Linear Regression    ShortestPath (Djikstra)    Insertion Sort


In [530]: pd.DataFrame(algos,index=['algo_1','algo_2','algo_3','algo_4',
'algo_5'])
Out[530]:
          machine learning  search     sorting
algo_1   RandomForest        DFS     Quicksort
```

```
algo_2    K Nearest Neighbor  BFS       Mergesort
algo_3    Logistic Regression  Binary Search Heapsort
algo_4    K-Means Clustering  Linear       Bubble Sort
algo_5    Linear Regression  ShortestPath (Djikstra) Insertion Sort
```

Using a structured array

In this case, we use a structured array, which is an array of records or
structs. For more information on structured arrays, refer to the following:
http://docs.scipy.org/doc/numpy/user/basics.rec.html.

```
In [533]: memberData = np.zeros((4,),
              dtype=[('Name','a15'),
                     ('Age','i4'),
                     ('Weight','f4')])
        memberData[:] = [('Sanjeev',37,162.4),
                         ('Yingluck',45,137.8),
                         ('Emeka',28,153.2),
                         ('Amy',67,101.3)]
          memberDF=pd.DataFrame(memberData);memberDF
Out[533]:        Name        Age     Weight
        0    Sanjeev    37   162.4
        1    Yingluck   45   137.8
        2    Emeka        28   153.2
        3    Amy        67   101.3
In [534]: pd.DataFrame(memberData, index=['a','b','c','d'])
Out[534]:    Name        Age     Weight
        a    Sanjeev    37   162.4
        b    Yingluck   45   137.8
        c    Emeka        28   153.2
        d    Amy        67   101.3
```

Using a Series structure

Here, we show how to construct a DataFrame structure from a Series structure:

```
In [ 540]: currSeries.name='currency'
          pd.DataFrame(currSeries)
Out[540]:        currency
```

```
China    yuan

Germany  euro

Japan    yen

Mexico   peso

Nigeria  naira

UK       pound

US       dollar
```

There are also alternative constructors for DataFrame; they can be summarized as follows:

- `DataFrame.from_dict`: It takes a dictionary of dictionaries or sequences and returns DataFrame.

- `DataFrame.from_records`: It takes a list of tuples or structured ndarray.

- `DataFrame.from_items`: It takes a sequence of (key, value) pairs. The keys are the column or index names, and the values are the column or row values. If you wish the keys to be row index names, you must specify orient='index' as a parameter and specify the column names.

- `pandas.io.parsers.read_csv`: This is a helper function that reads a CSV file into a pandas DataFrame structure.

- `pandas.io.parsers.read_table`: This is a helper function that reads a delimited file into a pandas DataFrame structure.

- `pandas.io.parsers.read_fwf`: This is a helper function that reads a table of fixed-width lines into a pandas DataFrame structure.

Operations

Here, I will briefly describe the various DataFrame operations.

Selection

A specific column can be obtained as a Series structure:

```
In [543]: memberDF['Name']

Out[543]: 0    Sanjeev

          1    Yingluck

          2    Emeka

          3    Amy

   Name: Name, dtype: object
```

Assignment

A new column can be added via assignment, as follows:

```
In [545]:    memberDF['Height']=60;memberDF
Out[545]:          Name      Age   Weight   Height
  0                Sanjeev   37    162.4    60
  1                Yingluck  45    137.8    60
  2                Emeka     28    153.2    60
  3                Amy       67    101.3    60
```

Deletion

A column can be deleted, as you would in the case of `dict`:

```
In [546]: del memberDF['Height']; memberDF
Out[546]:          Name      Age   Weight
  0                Sanjeev   37    162.4
  1                Yingluck  45    137.8
  2                Emeka     28    153.2
  3                Amy       67    101.3
```

It can also be popped, as with a dictionary:

```
In [547]: memberDF['BloodType']='O'
          bloodType=memberDF.pop('BloodType'); bloodType
Out[547]: 0    O
          1    O
          2    O
          3    O
     Name: BloodType, dtype: object
```

Basically, a DataFrame structure can be treated as if it were a dictionary of Series objects. Columns get inserted at the end; to insert a column at a specific location, you can use the insert function:

```
In [552]: memberDF.insert(2,'isSenior',memberDF['Age']>60);
          memberDF
Out[552]:          Name      Age   isSenior   Weight
  0                Sanjeev   37    False       162.4
  1                Yingluck  45    False       137.8
  2                Emeka     28    False       153.2
  3                Amy       67    True        101.3
```

Alignment

DataFrame objects align in a manner similar to Series objects, except that they align on both column and index labels. The resulting object is the union of the column and row labels:

```
In [559]: ore1DF=pd.DataFrame(np.array([[20,35,25,20],
                                        [11,28,32,29]]),
                              columns=['iron','magnesium',
                                       'copper','silver'])

      ore2DF=pd.DataFrame(np.array([[14,34,26,26],
                                    [33,19,25,23]]),
                          columns=['iron','magnesium',
                                   'gold','silver'])

    ore1DF+ore2DF
Out[559]:      copper  gold  iron  magnesium  silver
  0            NaN     NaN   34    69         46
  1            NaN     NaN   44    47         52
```

In the case where there are no row labels or column labels in common, the value is filled with `NaN`, for example, copper and gold. If you combine a DataFrame object and a Series object, the default behavior is to broadcast the Series object across the rows:

```
In [562]: ore1DF + pd.Series([25,25,25,25],
                             index=['iron','magnesium',
                             'copper','silver'])
Out[562]:    iron  magnesium   copper   silver
  0          45    60          50       45
  1          36    53          57       54
```

Other mathematical operations

Mathematical operators can be applied element wise on DataFrame structures:

```
In [565]: np.sqrt(ore1DF)
Out[565]:        iron      magnesium   copper      silver
  0              4.472136  5.916080    5.000000    4.472136
  1              3.316625  5.291503    5.656854    5.385165
```

Panel

Panel is a 3D array. It is not as widely used as Series or DataFrame. It is not as easily displayed on screen or visualized as the other two because of its 3D nature. The Panel data structure is the final piece of the data structure jigsaw puzzle in pandas. It is less widely used, and is used for 3D data. The three axis names are as follows:

- **items**: This is axis 0. Each each item corresponds to a DataFrame structure.
- **major_axis**: This is axis 1. Each item corresponds to the rows of the DataFrame structure.
- **minor_axis**: This is axis 2. Each item corresponds to the columns of each DataFrame structure.

As for Series and DataFrame, there are different ways to create Panel objects. They are explained in the upcoming sections.

Using 3D NumPy array with axis labels

Here, we show how to construct a Panel object from a 3D NumPy array.

```
In 586[]: stockData=np.array([[[63.03,61.48,75],
                               [62.05,62.75,46],
                                [62.74,62.19,53]],
                                 [[411.90, 404.38, 2.9],
                                  [405.45, 405.91, 2.6],
                                  [403.15, 404.42, 2.4]]])

          stockData
Out[586]: array([[[  63.03,   61.48,   75.  ],
                  [  62.05,   62.75,   46.  ],
                  [  62.74,   62.19,   53.  ]],
                [[ 411.9 ,  404.38,    2.9 ],
                 [ 405.45,  405.91,    2.6 ],
                 [ 403.15,  404.42,    2.4 ]]])
In [587]: stockHistoricalPrices = pd.Panel(stockData,
                                        items=['FB', 'NFLX'],
                                major_axis=pd.date_range('2/3/2014',
periods=3),
minor_axis=['open price', 'closing price', 'volume'])

          stockHistoricalPrices
```

```
Out[587]: <class 'pandas.core.panel.Panel'>

          Dimensions: 2 (items) x 3 (major_axis) x 3 (minor_axis)

          Items axis: FB to NFLX

          Major_axis axis: 2014-02-03 00:00:00 to 2014-02-05 00:00:00

          Minor_axis axis: open price to volume
```

Using a Python dictionary of DataFrame objects

We construct a Panel structure by using a Python dictionary of DataFrame structures.

```
In [591]: USData=pd.DataFrame(np.array([[249.62  , 8900],
                                        [ 282.16,12680],
                                        [309.35,14940]]),
                  columns=['Population(M)','GDP($B)'],
                  index=[1990,2000,2010])
          USData
Out[591]:         Population(M)    GDP($B)
          1990    249.62          8900
          2000    282.16          12680
          2010    309.35          14940
In [590]: ChinaData=pd.DataFrame(np.array([[1133.68, 390.28],
                                           [ 1266.83,1198.48],
                                           [1339.72, 6988.47]]),

                    columns=['Population(M)','GDP($B)'],
                    index=[1990,2000,2010])
          ChinaData
Out[590]:           Population(M)    GDP($B)
            1990    1133.68          390.28
            2000    1266.83          1198.48
            2010    1339.72          6988.47
In [592]:US_ChinaData={'US' : USData,
                       'China': ChinaData}
          pd.Panel(US_ChinaData)
Out[592]: <class 'pandas.core.panel.Panel'>
```

```
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: China to US
Major_axis axis: 1990 to 2010
```

Using the DataFrame.to_panel method

This method converts a DataFrame structure having a MultiIndex to a Panel structure:

```
In [617]: mIdx = pd.MultiIndex(levels=[['US', 'China'],
                                       [1990,2000, 2010]],
                  labels=[[1,1,1,0,0,0],[0,1,2,0,1,2]])
mIdx
Out[617]: MultiIndex
          [(u'China', 1990), (u'China', 2000), (u'China', 2010),
           (u'US', 1990), (u'US', 2000), (u'US', 2010)]


ChinaUSDF = pd.DataFrame({'Population(M)' : [1133.68, 1266.83,
                                             1339.72, 249.62,
                                             282.16,309.35],
                          'GDB($B)': [390.28, 1198.48, 6988.47,
                                      8900,12680, 14940]}, index=mIdx)
ChinaUSDF
In [618]: ChinaUSDF = pd.DataFrame({'Population(M)' : [1133.68,
                                                       1266.83,
                                                       1339.72,
                                                       249.62,
                                                       282.16,
                                                       309.35],
                                    'GDB($B)': [390.28, 1198.48,
                                                6988.47, 8900,
                                                12680,14940]},
                  index=mIdx)
          ChinaUSDF


Out[618]:                         GDB($B)        Population(M)
```

```
            China       1990    390.28      1133.68
                        2000    1198.48     1266.83
                        2010    6988.47     1339.72
            US          1990    8900.00      249.62
                        2000    12680.00     282.16
                        2010    14940.00     309.35
In [622]: ChinaUSDF.to_panel()
Out[622]: <class 'pandas.core.panel.Panel'>
          Dimensions: 2 (items) x 2 (major_axis) x 3 (minor_axis)
          Items axis: GDB($B) to Population(M)
          Major_axis axis: US to China
          Minor_axis axis: 1990 to 2010
```

The sources of US/China Economic data are the following sites:

- `http://www.multpl.com/us-gdp-inflation-adjusted/table`
- `http://www.multpl.com/united-states-population/table`
- `http://en.wikipedia.org/wiki/Demographics_of_China`
- `http://www.theguardian.com/news/datablog/2012/mar/23/china-gdp-since-1980`

Other operations

Insertion, deletion, and item-wise operations behave the same as in the case of DataFrame. Panel structures can be re-arranged via transpose. The feature set of the operations of Panel is relatively underdeveloped and not as rich as for Series and DataFrame.

Summary

To summarize this chapter, `numpy.ndarray` is the bedrock data structure on which the pandas data structures are based. The pandas data structures at their heart consist of NumPy `ndarray` of data and an array or arrays of labels.

There are three main data structures in pandas: Series, DataFrame, and Panel. The pandas data structures are much easier to use and more user-friendly than Numpy `ndarrays`, since they provide row indexes and column indexes in the case of DataFrame and Panel. The DataFrame object is the most popular and widely used object in pandas. In the next chapter, we will cover the topic of indexing in pandas.

4
Operations in pandas, Part I – Indexing and Selecting

In this chapter, we will focus on the indexing and selection of data from pandas objects. This is important since effective use of pandas requires a good knowledge of the indexing and selection of data. The topics that we will address in this chapter include the following:

- Basic indexing
- Label, integer, and mixed indexing
- MultiIndexing
- Boolean indexing
- Operations on indexes

Basic indexing

We have already discussed basic indexing on Series and DataFrames in the previous chapter, but here we will include some examples for the sake of completeness. Here, we list a time series of crude oil spot prices for the 4 quarters of 2013, taken from IMF data: `http://www.imf.org/external/np/res/commod/pdf/monthly/011014.pdf`.

```
In [642]:SpotCrudePrices_2013_Data={
                'U.K. Brent' : {'2013-Q1':112.9, '2013-Q2':103.0, '2013-Q3':110.1, '2013-Q4':109.4},
                'Dubai':{'2013-Q1':108.1, '2013-Q2':100.8, '2013-Q3':106.1,'2013-Q4':106.7},
                'West Texas Intermediate':{'2013-Q1':94.4, '2013-Q2':94.2, '2013-Q3':105.8,'2013-Q4':97.4}}
```

```
            SpotCrudePrices_2013=pd.DataFrame.from_
dict(SpotCrudePrices_2013_Data)
        SpotCrudePrices_2013
Out[642]:        Dubai    U.K. Brent  West Texas Intermediate
      2013-Q1  108.1    112.9              94.4
      2013-Q2  100.8    103.0              94.2
      2013-Q3  106.1    110.1             105.8
      2013-Q4  106.7    109.4              97.4
```

We can select the prices for the available time periods of Dubai crude oil by using the `[]` operator:

```
In [644]: dubaiPrices=SpotCrudePrices_2013['Dubai']; dubaiPrices
Out[644]: 2013-Q1    108.1
          2013-Q2    100.8
          2013-Q3    106.1
          2013-Q4    106.7
          Name: Dubai, dtype: float64
```

We can pass a list of columns to the `[]` operator in order to select the columns in a particular order:

```
In [647]: SpotCrudePrices_2013[['West Texas Intermediate','U.K. Brent']]
Out[647]:           West Texas Intermediate        U.K. Brent
          2013-Q1                    94.4         112.9
          2013-Q2                    94.2         103.0
          2013-Q3                   105.8         110.1
          2013-Q4                    97.4         109.4
```

If we specify a column that is not listed in the DataFrame, we will get a `KeyError` exception:

```
In [649]: SpotCrudePrices_2013['Brent Blend']
        -------------------------------------------------------
        KeyError                                 Traceback (most
recent call last)
        <ipython-input-649-cd2d76b24875> in <module>()
        ...
        KeyError: u'no item named Brent Blend'
```

We can avoid this error by using the `get` operator and specifying a default value in the case when the column is not present, as follows:

```
In [650]: SpotCrudePrices_2013.get('Brent Blend','N/A')
Out[650]: 'N/A'
```

> Note that rows cannot be selected with the bracket operator `[]` in a DataFrame.

Hence, we get an error in the following case:

```
In [755]:SpotCrudePrices_2013['2013-Q1']

        -------------------------------------------------
        KeyError        Traceback (most recent call last)
        ...
        KeyError: u'no item named 2013-Q1'
```

This was a design decision made by the creators in order to avoid ambiguity. In the case of a Series, where there is no ambiguity, selecting rows by using the `[]` operator works:

```
In [756]: dubaiPrices['2013-Q1']
Out[756]: 108.1
```

We shall see how we can perform row selection by using one of the newer indexing operators later in this chapter.

Accessing attributes using dot operator

One can retrieve values from a Series, DataFrame, or Panel directly as an attribute as follows:

```
In [650]: SpotCrudePrices_2013.Dubai
Out[650]: 2013-Q1    108.1
          2013-Q2    100.8
          2013-Q3    106.1
          2013-Q4    106.7
          Name: Dubai, dtype: float64
```

However, this only works if the index element is a valid Python identifier as follows:

```
In [653]: SpotCrudePrices_2013."West Texas Intermediate"
          File "<ipython-input-653-2a782563c15a>", line 1
```

```
        SpotCrudePrices_2013."West Texas Intermediate"
                              ^

        SyntaxError: invalid syntax
```

Otherwise, we get `SyntaxError` as in the preceding case because of the space in the column name. A valid Python identifier must follow the following lexical convention:

```
identifier::= (letter|"_") (letter | digit | "_")*
```

Thus, a valid Python identifier cannot contain a space. See the Python Lexical Analysis documents for more details at `http://docs.python.org/2.7/reference/lexical_analysis.html#identifiers`.

We can resolve this by renaming the column index names so that they are all valid identifiers:

```
In [654]: SpotCrudePrices_2013
Out[654]:              Dubai    U.K. Brent    West Texas Intermediate
        2013-Q1       108.1    112.9         94.4

        2013-Q2       100.8    103.0         94.2

        2013-Q3       106.1    110.1         105.8

        2013-Q4       106.7    109.4         97.4


In [655]:SpotCrudePrices_2013.columns=['Dubai','UK_Brent',
                                     'West_Texas_Intermediate']
SpotCrudePrices_2013
Out[655]:        Dubai    UK_Brent      West_Texas_Intermediate
        2013-Q1  108.1    112.9         94.4

        2013-Q2  100.8    103.0         94.2

        2013-Q3  106.1    110.1         105.8

        2013-Q4  106.7    109.4         97.4
```

We can then select the prices for West Texas Intermediate as desired:

```
In [656]:SpotCrudePrices_2013.West_Texas_Intermediate
Out[656]:2013-Q1     94.4
        2013-Q2     94.2

        2013-Q3     105.8

        2013-Q4     97.4

        Name: West_Texas_Intermediate, dtype: float64
```

We can also select prices by specifying a column index number to select column 1 (U.K. Brent) as follows:

```
In [18]: SpotCrudePrices_2013[[1]]
Out[18]:         U.K. Brent
        2013-Q1  112.9
        2013-Q2  103.0
        2013-Q3  110.1
        2013-Q4  109.4
```

Range slicing

As we saw in the section on NumPy ndarrays in *Chapter 3, The pandas Data structures*, we can slice a range by using the `[]` operator. The syntax of the slicing operator exactly matches that of NumPy:

```
ar[startIndex: endIndex: stepValue]
```

where the default values if not specified are as follows:

- 0 for startIndex
- arraysize -1 for endIndex
- 1 for stepValue

For a DataFrame, `[]` slices across rows as follows:

Obtain the first 2 rows:

```
In [675]: SpotCrudePrices_2013[:2]
Out[675]:        Dubai  UK_Brent West_Texas_Intermediate
        2013-Q1  108.1    112.9   94.4
        2013-Q2  100.8    103.0   94.2
```

Obtain all rows starting from index 2:

```
In [662]: SpotCrudePrices_2013[2:]
Out[662]:        Dubai  UK_Brent West_Texas_Intermediate
        2013-Q3  106.1    110.1   105.8
        2013-Q4  106.7    109.4   97.4
```

Obtain rows at intervals of two, starting from row 0:

```
In [664]: SpotCrudePrices_2013[::2]
Out[664]:         Dubai   UK_Brent West_Texas_Intermediate
        2013-Q1  108.1   112.9    94.4
        2013-Q3  106.1   110.1    105.8
```

Reverse the order of rows in DataFrame:

```
In [677]: SpotCrudePrices_2013[::-1]
Out[677]:         Dubai   UK_Brent West_Texas_Intermediate
        2013-Q4  106.7   109.4    97.4
        2013-Q3  106.1   110.1    105.8
        2013-Q2  100.8   103.0    94.2
        2013-Q1  108.1   112.9    94.4
```

For a Series, the behavior is just as intuitive:

```
In [666]: dubaiPrices=SpotCrudePrices_2013['Dubai']
```

Obtain the last 3 rows or all rows but the first:

```
In [681]: dubaiPrices[1:]
Out[681]: 2013-Q2    100.8
          2013-Q3    106.1
          2013-Q4    106.7
          Name: Dubai, dtype: float64
```

Obtain all rows but the last:

```
In [682]: dubaiPrices[:-1]
Out[682]: 2013-Q1    108.1
          2013-Q2    100.8
          2013-Q3    106.1
          Name: Dubai, dtype: float64
```

Reverse the rows:

```
In [683]: dubaiPrices[::-1]
Out[683]: 2013-Q4    106.7
          2013-Q3    106.1
          2013-Q2    100.8
          2013-Q1    108.1
          Name: Dubai, dtype: float64
```

Label, integer, and mixed indexing

In addition to the standard indexing operator `[]` and attribute operator, there are operators provided in pandas to make the job of indexing easier and more convenient. By label indexing, we generally mean indexing by a header name, which tends to be a string value in most cases. These operators are as follows:

- The `.loc` operator: It allows label-oriented indexing
- The `.iloc` operator: It allows integer-based indexing
- The `.ix` operator: It allows mixed label and integer-based indexing

We will now turn our attention to these operators.

Label-oriented indexing

The `.loc` operator supports pure label-based indexing. It accepts the following as valid inputs:

- A single label such as ['March'], [88] or ['Dubai']. Note that in the case where the label is an integer, it doesn't refer to the integer position of the index, but to the integer itself as a label.
- List or array of labels, for example, ['Dubai','UK Brent'].
- A slice object with labels, for example, 'May':'Aug'.
- A Boolean array.

For our illustrative dataset, we use the average snowy weather temperature data for New York city from the following:

- http://www.currentresults.com/Weather/New-York/Places/new-york-city-snowfall-totals-snow-accumulation-averages.php
- http://www.currentresults.com/Weather/New-York/Places/new-york-city-temperatures-by-month-average.php

Create DataFrame

```
In [723]: NYC_SnowAvgsData={'Months' :
['January','February','March',
'April', 'November', 'December'],
'Avg SnowDays' : [4.0,2.7,1.7,0.2,0.2,2.3],
'Avg Precip. (cm)' : [17.8,22.4,9.1,1.5,0.8,12.2],
'Avg Low Temp. (F)' : [27,29,35,45,42,32] }
```

```
In [724]: NYC_SnowAvgsData
Out[724]:{'Avg Low Temp. (F)': [27, 29, 35, 45, 42, 32],
          'Avg Precip. (cm)': [17.8, 22.4, 9.1, 1.5, 0.8, 12.2],
          'Avg SnowDays': [4.0, 2.7, 1.7, 0.2, 0.2, 2.3],
          'Months': ['January', 'February', 'March', 'April',
                     'November', 'December']}


In [726]:NYC_SnowAvgs=pd.DataFrame(NYC_SnowAvgsData,
                     index=NYC_SnowAvgsData['Months'],
                     columns=['Avg SnowDays','Avg Precip. (cm)',
                              'Avg Low Temp. (F)'])

          NYC_SnowAvgs


Out[726]:        Avg SnowDays   Avg Precip. (cm) Avg Low Temp. (F)
         January  4.0            17.8             27
         February 2.7            22.4             29
         March    1.7            9.1              35
         April    0.2            1.5              45
         November 0.2            0.8              42
         December 2.3            12.2             32
```

Using a single label:

```
In [728]: NYC_SnowAvgs.loc['January']
Out[728]: Avg SnowDays        4.0
          Avg Precip. (cm)    17.8
          Avg Low Temp. (F)   27.0
          Name: January, dtype: float64
```

Using a list of labels:

```
In [730]: NYC_SnowAvgs.loc[['January','April']]
Out[730]:        Avg SnowDays   Avg Precip. (cm) Avg Low Temp. (F)
         January  4.0            17.8             27
         April    0.2            1.5              45
```

Using a label range:

```
In [731]: NYC_SnowAvgs.loc['January':'March']
Out[731]:          Avg SnowDays    Avg Precip. (cm) Avg Low Temp. (F)
        January  4.0             17.8             27
        February 2.7             22.4             29
        March    1.7             9.1              35
```

Note that while using the `.loc`, `.iloc`, and `.ix` operators on a DataFrame, the row index must always be specified first. This is the opposite of the `[]` operator, where only columns can be selected directly. Hence, we get an error if we do the following:

```
In [771]: NYC_SnowAvgs.loc['Avg SnowDays']


KeyError: 'Avg SnowDays'
```

The correct way to do this is to specifically select all rows by using the colon (`:`) operator as follows:

```
In [772]: NYC_SnowAvgs.loc[:,'Avg SnowDays']
Out[772]: January     4.0
          February    2.7
          March       1.7
          April       0.2
          November    0.2
          December    2.3
          Name: Avg SnowDays, dtype: float64
```

Here, we see how to select a specific coordinate value, namely the average number of snow days in March:

```
In [732]: NYC_SnowAvgs.loc['March','Avg SnowDays']
Out[732]: 1.7
```

This alternative style is also supported:

```
In [733]: NYC_SnowAvgs.loc['March']['Avg SnowDays']
Out[733]: 1.7
```

The following is the equivalent of the preceding case using the square bracket operator `[]`:

```
In [750]: NYC_SnowAvgs['Avg SnowDays']['March']
Out[750]: 1.7
```

Note again, however, that specifying the row index value first as is done with the
`.loc` operator will result in `Keyerror`. This is a consequence of the fact discussed
previously, that the `[]` operator cannot be used to select rows directly. The columns
must be selected first to obtain a Series, which can then be selected by rows.
Thus, you will get `KeyError: u'no item named March'` if you use either of
the following:

```
In [757]: NYC_SnowAvgs['March']['Avg SnowDays']
```

Or

```
In [758]: NYC_SnowAvgs['March']
```

We can use the `.loc` operator to select the rows instead:

```
In [759]: NYC_SnowAvgs.loc['March']
Out[759]: Avg SnowDays          1.7
          Avg Precip. (cm)      9.1
          Avg Low Temp. (F)    35.0
          Name: March, dtype: float64
```

Selection using a Boolean array

Now, we will show how to select which months have less than one snow day on
average by using a Boolean array:

```
In [763]: NYC_SnowAvgs.loc[NYC_SnowAvgs['Avg SnowDays']<1,:]
Out[763]:          Avg SnowDays  Avg Precip. (cm) Avg Low Temp. (F)
          April    0.2           1.5              45
          November 0.2           0.8              42
```

Or, in the case of the spot crude prices mentioned earlier, select the column
corresponding to the brand of crude that was priced above 110 a barrel for
row 2013-Q1:

```
In [768]: SpotCrudePrices_2013.loc[:,SpotCrudePrices_2013.
loc['2013-Q1']>110]
Out[768]:          UK_Brent
          2013-Q1 112.9
          2013-Q2 103.0
          2013-Q3 110.1
          2013-Q4 109.4
```

Note that the preceding arguments involve the Boolean operators < and > that actually evaluate the Boolean arrays, for example:

```
In [769]: SpotCrudePrices_2013.loc['2013-Q1']>110
Out[769]: Dubai                    False
          UK_Brent                 True
          West_Texas_Intermediate  False
          Name: 2013-Q1, dtype: bool
```

Integer-oriented indexing

The `.iloc` operator supports integer-based positional indexing. It accepts the following as inputs:

- A single integer, for example, 7
- A list or array of integers, for example, [2,3]
- A slice object with integers, for example, 1:4

Let us create the following:

```
In [777]: import scipy.constants as phys
          import math
In [782]: sci_values=pd.DataFrame([[math.pi, math.sin(math.pi),
                                     math.cos(math.pi)],
                                    [math.e,math.log(math.e),
                                     phys.golden],
                                    [phys.c,phys.g,phys.e],
                                    [phys.m_e,phys.m_p,phys.m_n]],
                                    index=list(range(0,20,5)))
```

```
Out[782]:          0              1              2
          0    3.141593e+00   1.224647e-16   -1.000000e+00
          5    2.718282e+00   1.000000e+00   1.618034e+00
          10   2.997925e+08   9.806650e+00   1.602177e-19
          15   9.109383e-31   1.672622e-27   1.674927e-27
```

We can select the non-physical constants in the first two rows by using integer slicing:

```
In [789]: sci_values.iloc[:2]
Out[789]:                0            1           2
       0        3.141593  1.224647e-16 -1.000000
       5        2.718282  1.000000e+00  1.618034
```

Alternatively, we can use the speed of light and the acceleration of gravity in the third row:

```
In [795]: sci_values.iloc[2,0:2]
Out[795]: 0    2.997925e+08
          1    9.806650e+00
          dtype: float64
```

Note that the arguments to `.iloc` are strictly positional and have nothing to do with the index values. Hence, consider a case where we mistakenly think that we can obtain the third row by using the following:

```
In [796]: sci_values.iloc[10]
          --------------------------------------------------------
          IndexError                                 Traceback (most
recent call last)
          ...
          IndexError: index 10 is out of bounds for axis 0 with size 4
```

Here, we get `IndexError` in the preceding result; so, now, we should use the label-indexing operator `.loc` instead, as follows:

```
In [797]: sci_values.loc[10]
Out[797]: 0    2.997925e+08
          1    9.806650e+00
          2    1.602177e-19
          Name: 10, dtype: float64
```

To slice out a specific row, we can use the following:

```
In [802]: sci_values.iloc[2:3,:]
Out[802]:     0          1         2
       10   299792458  9.80665 1.602177e-19
```

To obtain a cross-section using an integer position, use the following:

```
In [803]: sci_values.iloc[3]
Out[803]: 0    9.109383e-31
          1    1.672622e-27
          2    1.674927e-27
          Name: 15, dtype: float64
```

If we attempt to slice past the end of the array, we obtain IndexError as follows:

```
In [805]: sci_values.iloc[6,:]

          -------------------------------------------------------

          IndexError                              Traceback (most
recent call last)

          IndexError: index 6 is out of bounds for axis 0 with size 4
```

The .iat and .at operators

The `.iat` and `.at` operators can be used for a quick selection of scalar values. This is best illustrated as follows:

```
In [806]: sci_values.iloc[3,0]
Out[806]: 9.1093829099999999e-31
In [807]: sci_values.iat[3,0]
Out[807]: 9.1093829099999999e-31


In [808]: %timeit sci_values.iloc[3,0]
          10000 loops, best of 3: 122 µs per loop
In [809]: %timeit sci_values.iat[3,0]
          10000 loops, best of 3: 28.4 µs per loop
```

Thus, we can see that `.iat` is much faster than the `.iloc`/`.ix` operators. The same applies to `.at` versus `.loc`.

Mixed indexing with the .ix operator

The `.ix` operator behaves like a mixture of the `.loc` and `.iloc` operators, with the `.loc` behavior taking precedence. It takes the following as possible inputs:

- A single label or integer
- A list of integers or labels

- An integer slice or label slice
- A Boolean array

Let us re-create the following DataFrame by saving the stock index closing price data to a file (`stock_index_closing.csv`) and reading it in:

```
TradingDate,Nasdaq,S&P 500,Russell 2000
2014/01/30,4123.13,1794.19,1139.36
2014/01/31,4103.88,1782.59,1130.88
2014/02/03,3996.96,1741.89,1094.58
2014/02/04,4031.52,1755.2,1102.84
2014/02/05,4011.55,1751.64,1093.59
2014/02/06,4057.12,1773.43,1103.93
```

The source for this data is `http://www.economagic.com/sp.htm#Daily`. Here's how we read the CSV data into a DataFrame:

```
In [939]: stockIndexDataDF=pd.read_csv('./stock_index_data.csv')
In [940]: stockIndexDataDF
Out[940]:       TradingDate  Nasdaq   S&P 500  Russell 2000
        0       2014/01/30   4123.13  1794.19  1139.36
        1       2014/01/31   4103.88  1782.59  1130.88
        2       2014/02/03   3996.96  1741.89  1094.58
        3       2014/02/04   4031.52  1755.20  1102.84
        4       2014/02/05   4011.55  1751.64  1093.59
        5       2014/02/06   4057.12  1773.43  1103.93
```

What we see from the preceding example is that the DataFrame created has an integer-based row index. We promptly set the index to be the trading date to index it based on the trading date so that we can use the `.ix` operator:

```
In [941]: stockIndexDF=stockIndexDataDF.set_index('TradingDate')
In [942]:stockIndexDF
Out[942]:            Nasdaq   S&P 500  Russell 2000
        TradingDate
        2014/01/30   4123.13  1794.19  1139.36
        2014/01/31   4103.88  1782.59  1130.88
        2014/02/03   3996.96  1741.89  1094.58
        2014/02/04   4031.52  1755.20  1102.84
        2014/02/05   4011.55  1751.64  1093.59
        2014/02/06   4057.12  1773.43  1103.93
```

We now show examples of using the `.ix` operator:

Using a single label:

```
In [927]: stockIndexDF.ix['2014/01/30']
Out[927]: Nasdaq          4123.13
          S&P 500         1794.19
          Russell 2000    1139.36
          Name: 2014/01/30, dtype: float64
```

Using a list of labels:

```
In [928]: stockIndexDF.ix[['2014/01/30']]
Out[928]:           Nasdaq    S&P 500   Russell 2000
          2014/01/30  4123.13   1794.19   1139.36
```

```
In [930]: stockIndexDF.ix[['2014/01/30','2014/01/31']]
Out[930]:           Nasdaq   S&P 500   Russell 2000
          2014/01/30  4123.13    1794.19   1139.36
          2014/01/31  4103.88    1782.59   1130.88
```

Note the difference in the output between using a single label versus using a list containing just a single label. The former results in a series and the latter, a DataFrame:

```
In [943]: type(stockIndexDF.ix['2014/01/30'])
Out[943]: pandas.core.series.Series
```

```
In [944]: type(stockIndexDF.ix[['2014/01/30']])
Out[944]: pandas.core.frame.DataFrame
```

For the former, the indexer is a scalar; for the latter, the indexer is a list. A list indexer is used to select multiple columns. A multi-column slice of a DataFrame can only result in another DataFrame since it is 2D; hence, what is returned in the latter case is a DataFrame.

Using a label-based slice:

```
In [932]: tradingDates=stockIndexDataDF.TradingDate
In [934]: stockIndexDF.ix[tradingDates[:3]]
Out[934]:              Nasdaq    S&P 500   Russell 2000
          2014/01/30    4123.13   1794.19   1139.36
```

```
2014/01/31        4103.88  1782.59  1130.88
2014/02/03        3996.96  1741.89  1094.58
```

Using a single integer:

```
In [936]: stockIndexDF.ix[0]
Out[936]: Nasdaq          4123.13
          S&P 500         1794.19
          Russell 2000    1139.36
          Name: 2014/01/30, dtype: float64
```

Using a list of integers:

```
In [938]: stockIndexDF.ix[[0,2]]
Out[938]:             Nasdaq   S&P 500   Russell 2000
         TradingDate
         2014/01/30    4123.13  1794.19   1139.36
         2014/02/03    3996.96  1741.89   1094.58
```

Using an integer slice:

```
In [947]: stockIndexDF.ix[1:3]
Out[947]:             Nasdaq   S&P 500   Russell 2000
         TradingDate
         2014/01/31    4103.88  1782.59   1130.88
         2014/02/03    3996.96  1741.89   1094.58
```

Using a Boolean array:

```
In [949]: stockIndexDF.ix[stockIndexDF['Russell 2000']>1100]
Out[949]:             Nasdaq   S&P 500   Russell 2000
         TradingDate
         2014/01/30    4123.13 1794.19   1139.36
         2014/01/31    4103.88 1782.59   1130.88
         2014/02/04    4031.52 1755.20   1102.84
         2014/02/06    4057.12 1773.43   1103.93
```

As in the case of .loc, the row index must be specified first for the .ix operator.

MultiIndexing

We now turn to the topic of MultiIndexing. Multi-level or hierarchical indexing is useful because it enables the pandas user to select and massage data in multiple dimensions by using data structures such as Series and DataFrame. In order to start, let us save the following data to a file: `stock_index_prices.csv` and read it in:

```
TradingDate,PriceType,Nasdaq,S&P 500,Russell 2000
2014/02/21,open,4282.17,1841.07,1166.25
2014/02/21,close,4263.41,1836.25,1164.63
2014/02/21,high,4284.85,1846.13,1168.43
2014/02/24,open,4273.32,1836.78,1166.74
2014/02/24,close,4292.97,1847.61,1174.55
2014/02/24,high,4311.13,1858.71,1180.29
2014/02/25,open,4298.48,1847.66,1176
2014/02/25,close,4287.59,1845.12,1173.95
2014/02/25,high,4307.51,1852.91,1179.43
2014/02/26,open,4300.45,1845.79,1176.11
2014/02/26,close,4292.06,1845.16,1181.72
2014/02/26,high,4316.82,1852.65,1188.06
2014/02/27,open,4291.47,1844.9,1179.28
2014/02/27,close,4318.93,1854.29,1187.94
2014/02/27,high,4322.46,1854.53,1187.94
2014/02/28,open,4323.52,1855.12,1189.19
2014/02/28,close,4308.12,1859.45,1183.03
2014/02/28,high,4342.59,1867.92,1193.5
```

```
In [950]:sharesIndexDataDF=pd.read_csv('./stock_index_prices.csv')
In [951]: sharesIndexDataDF
Out[951]:
   TradingDate  PriceType  Nasdaq    S&P 500  Russell 2000
0   2014/02/21    open     4282.17   1841.07  1166.25
1   2014/02/21    close    4263.41   1836.25  1164.63
2   2014/02/21    high     4284.85   1846.13  1168.43
3   2014/02/24    open     4273.32   1836.78  1166.74
4   2014/02/24    close    4292.97   1847.61  1174.55
5   2014/02/24    high     4311.13   1858.71  1180.29
```

```
6     2014/02/25    open      4298.48   1847.66   1176.00
7     2014/02/25    close     4287.59   1845.12   1173.95
8     2014/02/25    high      4307.51   1852.91   1179.43
9     2014/02/26    open      4300.45   1845.79   1176.11
10    2014/02/26    close     4292.06   1845.16   1181.72
11    2014/02/26    high      4316.82   1852.65   1188.06
12    2014/02/27    open      4291.47   1844.90   1179.28
13    2014/02/27    close     4318.93   1854.29   1187.94
14    2014/02/27    high      4322.46   1854.53   1187.94
15    2014/02/28    open      4323.52   1855.12   1189.19
16    2014/02/28    close     4308.12   1859.45   1183.03
17    2014/02/28    high      4342.59   1867.92   1193.50
```

Here, we create a MultiIndex from the trading date and `priceType` columns:

```
In [958]: sharesIndexDF=sharesIndexDataDF.set_index(['TradingDate','Price
Type'])
In [959]: mIndex=sharesIndexDF.index; mIndex
Out[959]: MultiIndex
    [(u'2014/02/21', u'open'), (u'2014/02/21', u'close'), (u'2014/02/21',
u'high'), (u'2014/02/24', u'open'), (u'2014/02/24', u'close'),
(u'2014/02/24', u'high'), (u'2014/02/25', u'open'), (u'2014/02/25',
u'close'), (u'2014/02/25', u'high'), (u'2014/02/26', u'open'),
(u'2014/02/26', u'close'), (u'2014/02/26', u'high'), (u'2014/02/27',
u'open'), (u'2014/02/27', u'close'), (u'2014/02/27', u'high'),
(u'2014/02/28', u'open'), (u'2014/02/28', u'close'), (u'2014/02/28',
u'high')]


In [960]: sharesIndexDF
Out[960]:                   Nasdaq   S&P 500    Russell 2000
TradingDate PriceType
2014/02/21  open     4282.17   1841.07   1166.25
            close    4263.41   1836.25   1164.63
            high     4284.85   1846.13   1168.43
2014/02/24  open     4273.32   1836.78   1166.74
            close    4292.97   1847.61   1174.55
            high     4311.13   1858.71   1180.29
2014/02/25  open     4298.48   1847.66   1176.00
            close    4287.59   1845.12   1173.95
```

```
         high    4307.51  1852.91  1179.43
2014/02/26  open    4300.45  1845.79  1176.11
         close   4292.06  1845.16  1181.72
         high    4316.82  1852.65  1188.06
2014/02/27  open    4291.47  1844.90  1179.28
         close   4318.93  1854.29  1187.94
         high    4322.46  1854.53  1187.94
2014/02/28  open    4323.52  1855.12  1189.19
         close   4308.12  1859.45  1183.03
         high    4342.59  1867.92  1193.50
```

Upon inspection, we see that the MultiIndex consists of a list of tuples. Applying the `get_level_values` function with the appropriate argument produces a list of the labels for each level of the index:

```
In [962]: mIndex.get_level_values(0)
Out[962]: Index([u'2014/02/21', u'2014/02/21', u'2014/02/21',
u'2014/02/24', u'2014/02/24', u'2014/02/24', u'2014/02/25',
u'2014/02/25', u'2014/02/25', u'2014/02/26', u'2014/02/26',
u'2014/02/26', u'2014/02/27', u'2014/02/27', u'2014/02/27',
u'2014/02/28', u'2014/02/28', u'2014/02/28'], dtype=object)
```

```
In [963]: mIndex.get_level_values(1)
Out[963]: Index([u'open', u'close', u'high', u'open', u'close', u'high',
u'open', u'close', u'high', u'open', u'close', u'high', u'open',
u'close', u'high', u'open', u'close', u'high'], dtype=object)
```

However, `IndexError` will be thrown if the value passed to `get_level_values()` is invalid or out of range:

```
In [88]: mIndex.get_level_values(2)
        --------------------------------------------------------
IndexError                      Traceback (most recent call last)
...
```

You can achieve hierarchical indexing with a MultiIndexed DataFrame:

```
In [971]: sharesIndexDF.ix['2014/02/21']
Out[971]:       Nasdaq   S&P 500  Russell 2000
  PriceType
  open       4282.17  1841.07  1166.25
  close      4263.41  1836.25  1164.63
```

```
  high      4284.85  1846.13  1168.43


In [976]: sharesIndexDF.ix['2014/02/21','open']
Out[976]: Nasdaq        4282.17
    S&P 500        1841.07
    Russell 2000   1166.25
    Name: (2014/02/21, open), dtype: float64
```

We can slice using a MultiIndex:

```
In [980]: sharesIndexDF.ix['2014/02/21':'2014/02/24']
Out[980]:       Nasdaq   S&P 500   Russell 2000
  TradingDate  PriceType
  2014/02/21   open  4282.17   1841.07    1166.25
         close 4263.41   1836.25    1164.63
         high  4284.85   1846.13    1168.43
  2014/02/24   open  4273.32   1836.78    1166.74
         close 4292.97   1847.61    1174.55
         high  4311.13   1858.71    1180.29
```

We can try slicing at a lower level:

```
In [272]:
sharesIndexDF.ix[('2014/02/21','open'):('2014/02/24','open')]
-----------------------------------------------------------------
KeyError                              Traceback (most recent call
last)
<ipython-input-272-65bb3364d980> in <module>()
----> 1 sharesIndexDF.ix[('2014/02/21','open'):('2014/02/24','open')]
...
KeyError: 'Key length (2) was greater than MultiIndex lexsort depth (1)'
```

However, this results in KeyError with a rather strange error message. The key lesson to be learned here is that the current incarnation of MultiIndex requires the labels to be sorted for the lower-level slicing routines to work correctly.

In order to do this, you can utilize the sortlevel() method, which sorts the labels of an axis within a MultiIndex. To be on the safe side, sort first before slicing with a MultiIndex. Thus, we can do the following:

```
In [984]: sharesIndexDF.sortlevel(0).ix[('2014/02/21','open'):('2014/02/2
4','open')]
Out[984]:          Nasdaq   S&P 500  Russell 2000

  TradingDate  PriceType

  2014/02/21   open      4282.17   1841.07    1166.25

  2014/02/24   close     4292.97   1847.61    1174.55

               high      4311.13   1858.71    1180.29

               open      4273.32   1836.78    1166.74
```

We can also pass a list of tuples:

```
In [985]: sharesIndexDF.ix[[('2014/02/21','close'),('2014/02/24','op
en')]]
Out[985]:      Nasdaq  S&P 500   Russell 2000

  TradingDate  PriceType

  2014/02/21   close  4263.41   1836.25   1164.63

  2014/02/24   open   4273.32   1836.78   1166.74

  2 rows × 3 columns
```

> Note that by specifying a list of tuples, instead of a range as in
> the previous example, we display only the values of the open
> PriceType rather than all three for the TradingDate 2014/02/24.

Swapping and reordering levels

The swaplevel function enables levels within the MultiIndex to be swapped:

```
In [281]: swappedDF=sharesIndexDF[:7].swaplevel(0, 1, axis=0)
          swappedDF
Out[281]:           Nasdaq    S&P 500   Russell 2000

  PriceType   TradingDate

  open        2014/02/21    4282.17   1841.07    1166.25
  close       2014/02/21    4263.41   1836.25    1164.63
  high        2014/02/21    4284.85   1846.13    1168.43
  open        2014/02/24    4273.32   1836.78    1166.74
  close       2014/02/24    4292.97   1847.61    1174.55
  high        2014/02/24    4311.13   1858.71    1180.29
  open        2014/02/25    4298.48   1847.66    1176.00
  7 rows × 3 columns
```

The `reorder_levels` function is more general, allowing you to specify the order of the levels:

```
In [285]: reorderedDF=sharesIndexDF[:7].reorder_levels(['PriceType',
                                                        'TradingDate'],
                                                       axis=0)
          reorderedDF
Out[285]:          Nasdaq     S&P 500   Russell 2000
  PriceType   TradingDate
  open       2014/02/21    4282.17   1841.07   1166.25
  close      2014/02/21    4263.41   1836.25   1164.63
  high       2014/02/21    4284.85   1846.13   1168.43
  open       2014/02/24    4273.32   1836.78   1166.74
  close      2014/02/24    4292.97   1847.61   1174.55
  high       2014/02/24    4311.13   1858.71   1180.29
  open       2014/02/25    4298.48   1847.66   1176.00
  7 rows × 3 columns
```

Cross sections

The `xs` method provides a shortcut means of selecting data based on a particular index level value:

```
In [287]: sharesIndexDF.xs('open',level='PriceType')
Out[287]:
            Nasdaq     S&P 500   Russell 2000
  TradingDate
  2014/02/21    4282.17   1841.07   1166.25
  2014/02/24    4273.32   1836.78   1166.74
  2014/02/25    4298.48   1847.66   1176.00
  2014/02/26    4300.45   1845.79   1176.11
  2014/02/27    4291.47   1844.90   1179.28
  2014/02/28    4323.52   1855.12   1189.19
  6 rows × 3 columns
```

The more long-winded alternative to the preceding command would be to use `swaplevel` to switch between the `TradingDate` and `PriceType` levels and then, perform the selection as follows:

```
In [305]: sharesIndexDF.swaplevel(0, 1, axis=0).ix['open']
Out[305]:      Nasdaq    S&P 500    Russell 2000
  TradingDate
  2014/02/21   4282.17   1841.07    1166.25
  2014/02/24   4273.32   1836.78    1166.74
  2014/02/25   4298.48   1847.66    1176.00
  2014/02/26   4300.45   1845.79    1176.11
  2014/02/27   4291.47   1844.90    1179.28
  2014/02/28   4323.52   1855.12    1189.19
  6 rows × 3 columns
```

Using `.xs` achieves the same effect as obtaining a cross-section in the previous section on integer-oriented indexing.

Boolean indexing

We use Boolean indexing to filter or select parts of the data. The operators are as follows:

Operators	Symbol
OR	|
AND	&
NOT	~

These operators must be grouped using parentheses when used together. Using the earlier DataFrame from the previous section, here, we display the trading dates for which the NASDAQ closed above 4300:

```
In [311]: sharesIndexDataDF.ix[(sharesIndexDataDF['PriceType']=='close')
& \

                 (sharesIndexDataDF['Nasdaq']>4300) ]
Out[311]:        PriceType   Nasdaq    S&P 500    Russell 2000
  TradingDate
  2014/02/27     close    4318.93    1854.29     1187.94
  2014/02/28     close    4308.12    1859.45     1183.03
  2 rows × 4 columns
```

You can also create Boolean conditions in which you use arrays to filter out parts of the data:

```
In [316]: highSelection=sharesIndexDataDF['PriceType']=='high'
          NasdaqHigh=sharesIndexDataDF['Nasdaq']<4300
          sharesIndexDataDF.ix[highSelection & NasdaqHigh]
Out[316]: TradingDate  PriceType Nasdaq  S&P 500  Russell 2000
          2014/02/21    high     4284.85  1846.13  1168.43
```

Thus, the preceding code snippet displays the only date in the dataset for which the Nasdaq Composite index stayed below the 4300 level for the entire trading session.

The is in and any all methods

These methods enable the user to achieve more with Boolean indexing than the standard operators used in the preceding sections. The `isin` method takes a list of values and returns a Boolean array with `True` at the positions within the Series or DataFrame that match the values in the list. This enables the user to check for the presence of one or more elements within a Series. Here is an illustration using Series:

```
In [317]:stockSeries=pd.Series(['NFLX','AMZN','GOOG','FB','TWTR'])
          stockSeries.isin(['AMZN','FB'])
Out[317]:0    False
         1     True
         2    False
         3     True
         4    False
         dtype: bool
```

Here, we use the Boolean array to select a sub-Series containing the values that we're interested in:

```
In [318]: stockSeries[stockSeries.isin(['AMZN','FB'])]
Out[318]: 1    AMZN
          3      FB
          dtype: object
```

For our DataFrame example, we switch to a more interesting dataset for those of us who are of a biological anthropology bent, that of classifying Australian mammals (a pet interest of mine):

```
In [324]: australianMammals=
             {'kangaroo': {'Subclass':'marsupial',
                            'Species Origin':'native'},
               'flying fox' : {'Subclass':'placental',
                                'Species Origin':'native'},
               'black rat': {'Subclass':'placental',
                              'Species Origin':'invasive'},
               'platypus' : {'Subclass':'monotreme',
                              'Species Origin':'native'},
               'wallaby' :  {'Subclass':'marsupial',
                              'Species Origin':'native'},
         'palm squirrel' : {'Subclass':'placental',
                             'Origin':'invasive'},
         'anteater':       {'Subclass':'monotreme', 'Origin':'native'},
         'koala':          {'Subclass':'marsupial', 'Origin':'native'}
}
```

> Some more information on mammals: Marsupials are pouched mammals, monotremes are egg-laying, and placentals give birth to live young. The source of this information is the following: `http://en.wikipedia.org/wiki/List_of_mammals_of_Australia`.

The source of the preceding image is Bennett's wallaby at `http://bit.ly/NG4R7N`.

```
In [328]: ozzieMammalsDF=pd.DataFrame(australianMammals)
In [346]: aussieMammalsDF=ozzieMammalsDF.T; aussieMammalsDF
Out[346]:          Subclass   Origin
  anteater        monotreme   native
  black rat       placental   invasive
  flying fox      placental   native
  kangaroo        marsupial   native
  koala            marsupial    native
  palm squirrel placental     invasive
  platypus        monotreme   native
  wallaby    marsupial    native
  8 rows × 2 columns
```

Let us try to select mammals that are native to Australia:

```
In [348]: aussieMammalsDF.isin({'Subclass':['marsupial'],'Origin':['nati
ve']})
Out[348]:      Subclass Origin
  anteater    False    True
  black rat   False    False
  flying fox   False    True
  kangaroo    True    True
  koala       True    True
  palm squirrel False False
  platypus    False    True
  wallaby    True    True
  8 rows × 2 columns
```

The set of values passed to `isin` can be an array or a dictionary. That works somewhat, but we can achieve better results by creating a mask as a combination of the `isin` and `all()` methods:

```
In [349]: nativeMarsupials={'Mammal Subclass':['marsupial'],
                            'Species Origin':['native']}
        nativeMarsupialMask=aussieMammalsDF.isin(nativeMarsupials).
all(True)
    aussieMammalsDF[nativeMarsupialMask]
Out[349]:       Subclass   Origin
```

```
kangaroo   marsupial   native
koala      marsupial   native
wallaby    marsupial   native
3 rows × 2 columns
```

Thus, we see that kangaroo, koala, and wallaby are the native marsupials in our dataset. The `any()` method returns whether any element is `True` in a Boolean DataFrame. The `all()` method filters return whether all elements are `True` in a Boolean DataFrame.

The source for this is `http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.any.html`.

Using the where() method

The `where()` method is used to ensure that the result of Boolean filtering is the same shape as the original data. First, we set the random number generator seed to 100 so that the user can generate the same values as shown next:

```
In [379]: np.random.seed(100)
       normvals=pd.Series([np.random.normal() for i in np.arange(10)])
    normvals
Out[379]: 0   -1.749765
          1    0.342680
          2    1.153036
          3   -0.252436
          4    0.981321
          5    0.514219
          6    0.221180
          7   -1.070043
          8   -0.189496
          9    0.255001
    dtype: float64


In [381]: normvals[normvals>0]
Out[381]: 1    0.342680
          2    1.153036
          4    0.981321
          5    0.514219
```

```
        6      0.221180

        9      0.255001

   dtype: float64


In [382]: normvals.where(normvals>0)

Out[382]: 0            NaN

          1      0.342680

          2      1.153036

          3            NaN

          4      0.981321

          5      0.514219

          6      0.221180

          7            NaN

          8            NaN

          9      0.255001

   dtype: float64
```

This method appears to be useful only in the case of a Series, as we get this behavior for free in the case of a DataFrame:

```
In [393]: np.random.seed(100)

       normDF=pd.DataFrame([[round(np.random.normal(),3) for i in
np.arange(5)] for j in range(3)],

          columns=['0','30','60','90','120'])

    normDF

Out[393]:  0    30    60    90    120

          0   -1.750    0.343    1.153   -0.252    0.981

          1    0.514    0.221   -1.070   -0.189    0.255

          2   -0.458    0.435   -0.584    0.817    0.673

          3 rows × 5 columns

In [394]: normDF[normDF>0]

Out[394]:  0    30    60    90    120

          0    NaN    0.343    1.153    NaN    0.981

          1    0.514    0.221    NaN        NaN    0.255

          2    NaN    0.435    NaN    0.817    0.673

          3 rows × 5 columns

In [395]: normDF.where(normDF>0)
```

```
Out[395]:  0   30   60   90   120

           0   NaN    0.343   1.153   NaN   0.981

           1   0.514  0.221   NaN     NaN   0.255

           2   NaN    0.435   NaN     0.817 0.673

           3   rows × 5 columns
```

The inverse operation of the `where` method is mask:

```
In [396]: normDF.mask(normDF>0)
Out[396]:  0   30   60   90   120

           0   -1.750  NaN   NaN    -0.252   NaN

           1   NaN     NaN   -1.070 -0.189   NaN

           2   -0.458  NaN   -0.584  NaN     NaN

           3   rows × 5 columns
```

Operations on indexes

To complete this chapter, we will discuss operations on indexes. We sometimes need to operate on indexes when we wish to re-align our data or select it in different ways. There are various operations:

The `set_index` - allows for the creation of an index on an existing DataFrame and returns an indexed DataFrame.

As we have seen before:

```
In [939]: stockIndexDataDF=pd.read_csv('./stock_index_data.csv')
In [940]: stockIndexDataDF
Out[940]:    TradingDate  Nasdaq   S&P 500  Russell 2000
  0          2014/01/30   4123.13  1794.19  1139.36
  1          2014/01/31   4103.88  1782.59  1130.88
  2          2014/02/03   3996.96  1741.89  1094.58
  3          2014/02/04   4031.52  1755.20  1102.84
  4          2014/02/05   4011.55  1751.64  1093.59
  5          2014/02/06   4057.12  1773.43  1103.93
```

Now, we can set the index as follows:

```
In [941]: stockIndexDF=stockIndexDataDF.set_index('TradingDate')
In [942]: stockIndexDF
Out[942]:    Nasdaq   S&P 500  Russell 2000
  TradingDate
```

```
   2014/01/30  4123.13    1794.19  1139.36
   2014/01/31 4103.88     1782.59  1130.88
   2014/02/03  3996.96    1741.89  1094.58
   2014/02/04  4031.52    1755.20  1102.84
   2014/02/05  4011.55    1751.64  1093.59
   2014/02/06  4057.12    1773.43  1103.93
```

The `reset_index` reverses `set_index`:

```
In [409]: stockIndexDF.reset_index()
Out[409]:
   TradingDate    Nasdaq    S&P 500   Russell 2000
0    2014/01/30   4123.13   1794.19      1139.36
1    2014/01/31   4103.88   1782.59      1130.88
2    2014/02/03   3996.96   1741.89      1094.58
3    2014/02/04   4031.52   1755.20      1102.84
4    2014/02/05   4011.55   1751.64      1093.59
5    2014/02/06   4057.12   1773.43      1103.93
6 rows × 4 columns
```

Summary

To summarize, there are various ways of selecting data from pandas:

- We can use basic indexing, which is closest to our understanding of accessing data in an array.
- We can use label- or integer-based indexing with the associated operators.
- We can use a MultiIndex, which is the pandas version of a composite key comprising multiple fields.
- We can use a Boolean/logical index.

For further references about indexing in pandas, please take a look at the official documentation at `http://pandas.pydata.org/pandas-docs/stable/indexing.html`.

In the next chapter, we will examine the topic of grouping, reshaping, and merging data using pandas.

5
Operations in pandas, Part II – Grouping, Merging, and Reshaping of Data

In this chapter, we tackle the question of rearranging data in our data structures. We examine the various functions that enable us to rearrange data by utilizing them on real-world datasets. Such functions include `groupby`, `concat`, `aggregate`, `append`, and so on. The topics that we'll discuss are as follow:

- Aggregation/grouping of data
- Merging and concatenating data
- Reshaping data

Grouping of data

We often detail granular data that we wish to aggregate or combine based on a grouping variable. We will illustrate some ways of doing this in the following sections.

The groupby operation

The `groupby` operation can be thought of as part of a process that involves the following three steps:

- Splitting the dataset
- Analyzing the data
- Aggregating or combining the data

The `groupby` clause is an operation on DataFrames. A Series is a 1D object, so performing a `groupby` operation on it is not very useful. However, it can be used to obtain distinct rows of the Series. The result of a `groupby` operation is not a DataFrame but `dict` of DataFrame objects. Let us start with a dataset involving the world's most popular sport—soccer.

This dataset, obtained from Wikipedia, contains data for the finals of the European club championship since its inception in 1955. For reference, you can go to `http://en.wikipedia.org/wiki/UEFA_Champions_League`.

Convert the `.csv` file into a DataFrame by using the following command:

```
In [27]: uefaDF=pd.read_csv('./euro_winners.csv')

In [28]: uefaDF.head()

Out[28]:
```

Season	Nation	Winners	Score	Runners-up	Runner-Up Nation	Venue	Attendance
1955–56	Spain	Real Madrid	4–3	Stade de Reims	France	Parc des Princes,Paris	38239
1956–57	Spain	Real Madrid	2–0	Fiorentina	Italy	Santiago Bernabéu Stadium, Madrid	124000
1957–58	Spain	Real Madrid	3–2	Milan	Italy	Heysel Stadium,Brussels	67000
1958–59	Spain	Real Madrid	2–0	Stade de Reims	France	Neckarstadion,Stuttgart	72000
1959–60	Spain	Real Madrid	7–3	Eintracht Frankfurt	Germany	Hampden Park,Glasgow	127621

Thus, the output shows the season, the nations to which the winning and runner-up clubs belong, the score, the venue, and the attendance figures. Suppose we wanted to rank the nations by the number of European club championships they had won. We can do this by using `groupby`. First, we apply `groupby` to the DataFrame and see what is the type of the result:

```
In [84]: nationsGrp =uefaDF.groupby('Nation');

       type(nationsGrp)

Out[84]: pandas.core.groupby.DataFrameGroupBy
```

Thus, we see that `nationsGrp` is of the `pandas.core.groupby.DataFrameGroupBy` type. The column on which we use `groupby` is referred to as the key. We can see what the groups look like by using the groups attribute on the resulting `DataFrameGroupBy` object:

```
In [97]: nationsGrp.groups

Out[97]: {'England': [12, 21, 22, 23, 24, 25, 26, 28, 43, 49, 52,
                      56],
```

```
        'France': [37],
       'Germany': [18, 19, 20, 27, 41, 45, 57],
        'Italy': [7, 8, 9, 13, 29, 33, 34, 38, 40, 47, 51, 54],
        'Netherlands': [14, 15, 16, 17, 32, 39],
        'Portugal': [5, 6, 31, 48],
        'Romania': [30],
        'Scotland': [11],
        'Spain': [0, 1, 2, 3, 4, 10, 36, 42, 44, 46, 50, 53, 55],
        'Yugoslavia': [35]}
```

This is basically a dictionary that just shows the unique groups and the axis labels corresponding to each group—in this case the row number. The number of groups is obtained by using the `len()` function:

```
In [109]: len(nationsGrp.groups)
Out[109]: 10
```

We can now display the number of wins of each nation in descending order by applying the `size()` function to the group and subsequently the `sort()` function, which sorts according to place:

```
In [99]: nationWins=nationsGrp.size()
In [100] nationWins.sort(ascending=False)
         nationWins
Out[100]: Nation
     Spain          13
     Italy          12
     England        12
     Germany         7
     Netherlands     6
     Portugal        4
     Yugoslavia      1
     Scotland        1
     Romania         1
     France          1
     dtype: int64
```

The size() function returns a Series with the group names as the index and the size of each group. The size() function is also an aggregation function. We will examine aggregation functions later in the chapter.

To do a further breakup of wins by country and club, we apply a multicolumn groupby function before applying size() and sort():

```
In [106]: winnersGrp =uefaDF.groupby(['Nation','Winners'])

          clubWins=winnersGrp.size()

          clubWins.sort(ascending=False)

          clubWins
Out[106]: Nation        Winners
          Spain         Real Madrid         9
          Italy         Milan               7
          Germany       Bayern Munich       5
          England       Liverpool           5
          Spain         Barcelona           4
          Netherlands   Ajax                4
          England       Manchester United   3
          Italy         Internazionale      3
                        Juventus            2
          Portugal      Porto               2
                        Benfica             2
          England       Nottingham Forest   2
                        Chelsea             1
          France        Marseille           1
          Yugoslavia    Red Star Belgrade   1
          Germany       Borussia Dortmund   1
                        Hamburg             1
          Netherlands   Feyenoord           1
                        PSV Eindhoven       1
          Romania       Steaua Bucuresti    1
          Scotland      Celtic              1
          England       Aston Villa         1
          dtype: int64
```

A multicolumn `groupby` specifies more than one column to be used as the key by specifying the key columns as a list. Thus, we can see that the most successful club in this competition has been Real Madrid of Spain. We now examine a richer dataset that will enable us to illustrate many more features of `groupby`. This dataset is also soccer related and provides statistics for the top four European soccer leagues in the 2012-2013 season:

- English Premier League or EPL
- Spanish Primera Division or La Liga
- Italian First Division or Serie A
- German Premier League or Bundesliga

The source of this information is at `http://soccerstats.com`.

Let us now read the goal stats data into a DataFrame as usual. In this case, we create a row index on the DataFrame using the month:

```
In [68]: goalStatsDF=pd.read_csv('./goal_stats_euro_leagues_2012-13.csv')
         goalStatsDF=goalStatsDF.set_index('Month')
```

We look at the snapshot of the head and tail ends of our dataset:

```
In [115]: goalStatsDF.head(3)
Out[115]:            Stat          EPL  La Liga  Serie A  Bundesliga
    Month
    08/01/2012   MatchesPlayed  20      20       10        10
    09/01/2012   MatchesPlayed  38      39       50        44
    10/01/2012   MatchesPlayed  31      31       39        27


In [116]: goalStatsDF.tail(3)
Out[116]:            Stat          EPL  La Liga  Serie A  Bundesliga
      Month
      04/01/2013   GoalsScored  105    127      102       104
      05/01/2013   GoalsScored  96     109      102       92
      06/01/2013   GoalsScored  NaN    80       NaN       NaN
```

There are two measures in this data frame—`MatchesPlayed` and `GoalsScored`—and the data is ordered first by `Stat` and then by `Month`. Note that the last row in the `tail()` output has the `NaN` values for all the columns except `La Liga` but we'll discuss this in more detail later. We can use `groupby` to display the stats, but this will be done by grouped year instead. Here is how this is done:

```
In [117]: goalStatsGroupedByYear = goalStatsDF.groupby(
lambda Month: Month.split('/')[2])
```

We can then iterate over the resulting `groupby` object and display the groups. In the following command, we see the two sets of statistics grouped by year. Note the use of the lambda function to obtain the year group from the first day of the month. For more information about lambda functions, go to `http://bit.ly/1apJNwS`:

```
In [118]: for name, group in goalStatsGroupedByYear:
          print name
          print group
   2012
```

Month	Stat	EPL	La Liga	Serie A	Bundesliga
08/01/2012	MatchesPlayed	20	20	10	10
09/01/2012	MatchesPlayed	38	39	50	44
10/01/2012	MatchesPlayed	31	31	39	27
11/01/2012	MatchesPlayed	50	41	42	46
12/01/2012	MatchesPlayed	59	39	39	26
08/01/2012	GoalsScored	57	60	21	23
09/01/2012	GoalsScored	111	112	133	135
10/01/2012	GoalsScored	95	88	97	77
11/01/2012	GoalsScored	121	116	120	137
12/01/2012	GoalsScored	183	109	125	72

```
   2013
```

Month	Stat	EPL	La Liga	Serie A	Bundesliga
01/01/2013	MatchesPlayed	42	40	40	18
02/01/2013	MatchesPlayed	30	40	40	36
03/01/2013	MatchesPlayed	35	38	39	36
04/01/2013	MatchesPlayed	42	42	41	36
05/01/2013	MatchesPlayed	33	40	40	27

```
06/02/2013   MatchesPlayed  NaN      10     NaN       NaN
01/01/2013     GoalsScored  117     121     104        51
02/01/2013     GoalsScored   87     110     100       101
03/01/2013     GoalsScored   91     101      99       106
04/01/2013     GoalsScored  105     127     102       104
05/01/2013     GoalsScored   96     109     102        92
06/01/2013     GoalsScored  NaN      80     NaN       NaN
```

If we wished to group by individual month instead, we would need to apply `groupby` with a level argument, as follows:

```
In [77]: goalStatsGroupedByMonth = goalStatsDF.groupby(level=0)


In [81]: for name, group in goalStatsGroupedByMonth:
           print name
            print group
           print "\n"


01/01/2013
                  Stat  EPL  La Liga  Serie A  Bundesliga
Month
01/01/2013  MatchesPlayed   42       40       40          18
01/01/2013    GoalsScored  117      121      104          51



02/01/2013
                  Stat  EPL  La Liga  Serie A  Bundesliga
Month
02/01/2013  MatchesPlayed   30       40       40          36
02/01/2013    GoalsScored   87      110      100         101



03/01/2013
                  Stat  EPL  La Liga  Serie A  Bundesliga
Month
03/01/2013  MatchesPlayed   35       38       39          36
03/01/2013    GoalsScored   91      101       99         106
```

```
04/01/2013
                  Stat  EPL  La Liga  Serie A  Bundesliga
Month
04/01/2013  MatchesPlayed   42       42       41          36
04/01/2013     GoalsScored  105      127      102         104



05/01/2013
                  Stat  EPL  La Liga  Serie A  Bundesliga
Month
05/01/2013  MatchesPlayed   33       40       40          27
05/01/2013     GoalsScored   96      109      102          92



06/01/2013
                Stat  EPL  La Liga  Serie A  Bundesliga
Month
06/01/2013  GoalsScored  NaN       80      NaN         NaN


06/02/2013
                   Stat  EPL  La Liga  Serie A  Bundesliga
Month
06/02/2013  MatchesPlayed  NaN       10      NaN         NaN



08/01/2012
                   Stat  EPL  La Liga  Serie A  Bundesliga
Month
08/01/2012  MatchesPlayed   20       20       10          10
08/01/2012     GoalsScored   57       60       21          23


09/01/2012
                   Stat  EPL  La Liga  Serie A  Bundesliga
Month
09/01/2012  MatchesPlayed   38       39       50          44
09/01/2012     GoalsScored  111      112      133         135
```

```
10/01/2012
                    Stat   EPL   La Liga   Serie A   Bundesliga
Month
10/01/2012   MatchesPlayed   31       31        39          27
10/01/2012     GoalsScored   95       88        97          77


11/01/2012
                    Stat   EPL   La Liga   Serie A   Bundesliga
Month
11/01/2012   MatchesPlayed   50       41        42          46
11/01/2012     GoalsScored  121      116       120         137


12/01/2012
                    Stat   EPL   La Liga   Serie A   Bundesliga
Month
12/01/2012   MatchesPlayed   59       39        39          26
12/01/2012     GoalsScored  183      109       125          72
```

Note that since in the preceding commands we're grouping on an index, we need to specify the level argument as opposed to just using a column name. When we group by multiple keys, the resulting group name is a tuple, as shown in the upcoming commands. First, we reset the index to obtain the original DataFrame and define a MultiIndex in order to be able to group by multiple keys. If this is not done, it will result in a `ValueError`:

```
In [246]: goalStatsDF=goalStatsDF.reset_index()
          goalStatsDF=goalStatsDF.set_index(['Month','Stat'])


In [247]: monthStatGroup=goalStatsDF.groupby(level=['Month','Stat'])


In [248]: for name, group in monthStatGroup:
          print name
          print group


('01/01/2013', 'GoalsScored')
                        EPL   La Liga   Serie A   Bundesliga
Month       Stat
01/01/2013 GoalsScored  117      121       104          51
```

```
('01/01/2013', 'MatchesPlayed')
                         EPL  La Liga  Serie A  Bundesliga
Month       Stat
01/01/2013 MatchesPlayed  42       40      40         18
('02/01/2013', 'GoalsScored')
                         EPL  La Liga  Serie A  Bundesliga
Month       Stat
02/01/2013 GoalsScored   87      110     100        101
```

Using groupby with a MultiIndex

If our DataFrame has a MultiIndex, we can use `groupby` to group by different levels of the hierarchy and compute some interesting statistics. Here is the goal stats data using a MultiIndex consisting of `Month` and then `Stat`:

```
In [134]:goalStatsDF2=pd.read_csv('./goal_stats_euro_leagues_2012-13.
csv')

       goalStatsDF2=goalStatsDF2.set_index(['Month','Stat'])
In [141]: print goalStatsDF2.head(3)

       print goalStatsDF2.tail(3)

                          EPL  La Liga  Serie A  Bundesliga
Month       Stat
08/01/2012 MatchesPlayed   20       20       10          10
09/01/2012 MatchesPlayed   38       39       50          44
10/01/2012 MatchesPlayed   31       31       39          27
                          EPL  La Liga  Serie A  Bundesliga
Month       Stat
04/01/2013 GoalsScored    105      127      102         104
05/01/2013 GoalsScored     96      109      102          92
06/01/2013 GoalsScored    NaN       80      NaN         NaN
```

Suppose we wish to compute the total number of goals scored and the total matches played for the entire season for each league, we could do this as follows:

```
In [137]: grouped2=goalStatsDF2.groupby(level='Stat')
In [139]: grouped2.sum()
Out[139]:        EPL  La Liga  Serie A  Bundesliga  Stat
        GoalsScored   1063  1133     1003  898
        MatchesPlayed 380    380      380  306
```

Incidentally, the same result as the preceding one can be obtained by using sum directly and passing the level as a parameter:

```
In [142]: goalStatsDF2.sum(level='Stat')
Out[142]:                EPL   La Liga   Serie A   Bundesliga    Stat
          GoalsScored    1063   1133      1003      898
          MatchesPlayed   380    380       380      306
```

Now, let us obtain a key statistic to determine how *exciting* the season was in each of the leagues—the goals per game ratio:

```
In [174]: totalsDF=grouped2.sum()

In [175]: totalsDF.ix['GoalsScored']/totalsDF.ix['MatchesPlayed']
Out[175]: EPL           2.797368
          La Liga       2.981579
          Serie A       2.639474
          Bundesliga    2.934641
          dtype: float64
```

This is returned as a Series, as shown in the preceding command. We can now display the goals per game ratio along with the goals scored and matches played to give a summary of how exciting the league was, as follows:

1. Obtain goals per game data as a DataFrame. Note that we have to transpose it since `gpg` is returned as a Series:

   ```
   In [234]: gpg=totalsDF.ix['GoalsScored']/totalsDF.
   ix['MatchesPlayed']
          goalsPerGameDF=pd.DataFrame(gpg).T

   In [235]: goalsPerGameDF
   Out[235]:     EPL    La Liga    Serie A    Bundesliga
          0   2.797368   2.981579   2.639474   2.934641
   ```

2. Reindex the `goalsPerGameDF` DataFrame so that the `0` index is replaced by `GoalsPerGame`:

   ```
   In [207]: goalsPerGameDF=goalsPerGameDF.rename(index={0:'GoalsPerG
   ame'})

   In [208]: goalsPerGameDF
   ```

```
Out[208]:           EPL      La Liga    Serie A    Bundesliga
         GoalsPerGame  2.797368  2.981579  2.639474  2.934641
```

3. Append the `goalsPerGameDF` DataFrame to the original one:

```
In [211]: pd.options.display.float_format='{:.2f}'.format
          totalsDF.append(goalsPerGameDF)
Out[211]:        EPL     La Liga     Serie A    Bundesliga
  GoalsScored    1063.00  1133.00   1003.00     898.00
  MatchesPlayed  380.00    380.00    380.00     306.00
  GoalsPerGame     2.80      2.98      2.64       2.93
```

The following is a graph that shows the goals per match of the European leagues, that we discussed, from 1955-2012. The source for this can be found at `http://mattstil.es/images/europe-football.png`.
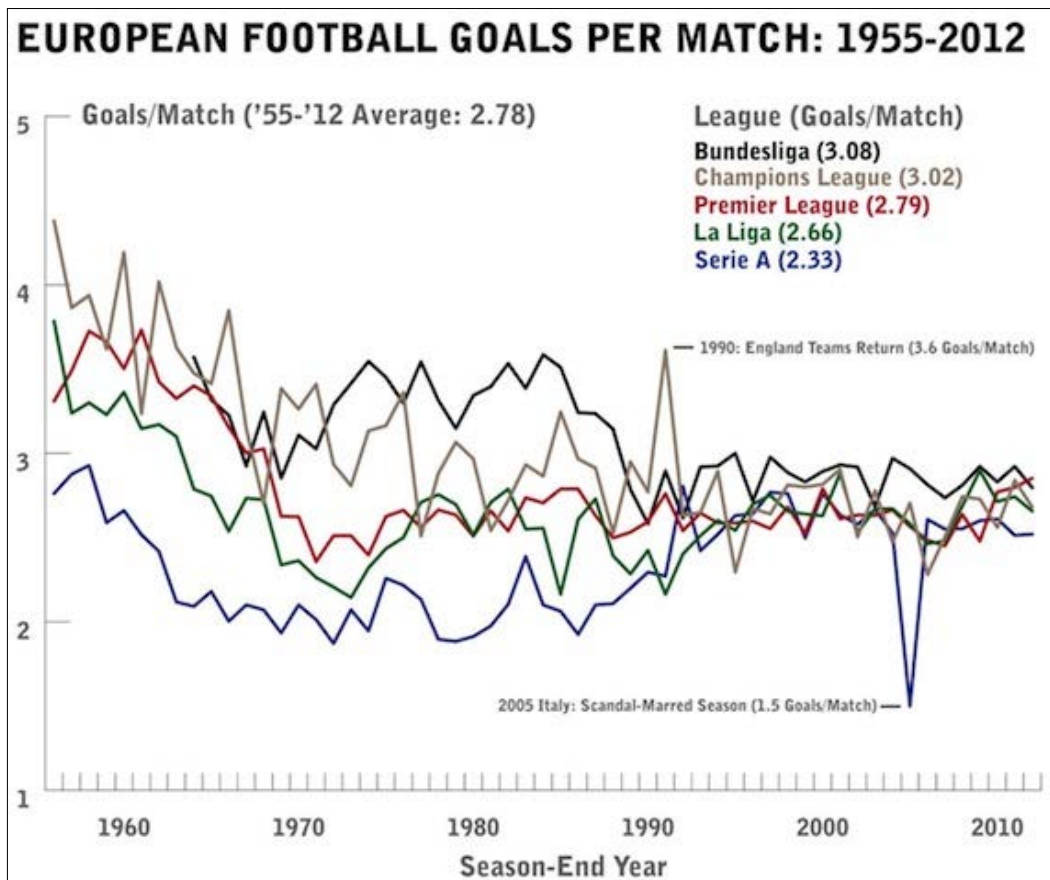
Using the aggregate method

Another way to generate summary statistics is by using the aggregate method explicitly:

```
In [254]: pd.options.display.float_format=None
In [256]: grouped2.aggregate(np.sum)
Out[256]:        EPL  La Liga  Serie A  Bundesliga    Stat
         GoalsScored    1063   1133   1003    898
         MatchesPlayed  380    380    380    306
```

This generates a grouped DataFrame object that is shown in the preceding command. We also reset the float format to `None`, so the integer-valued data would not be shown as floats due to the formatting from the previous section.

Applying multiple functions

For a grouped DataFrame object, we can specify a list of functions to be applied to each column:

```
In [274]: grouped2.agg([np.sum, np.mean,np.size])
Out[274]:       EPL            La Liga         Serie A          Bundesliga
       sum mean size   sum mean size   sum mean size sum mean size Stat
       GoalsScored   1063 106.3 11  1133 103.0 11  1003 100.3 11  898 89.8   11
       MatchesPlayed 380 38.0 11   380 34.6   11   380 38.0 11   306 30.6   11
```

Note the preceding output that shows NA values are excluded from aggregate calculations. The `agg` is an abbreviation form for aggregate. Thus, the calculations for the mean for EPL, Serie A, and Bundesliga are based on a size of 10 months and not 11. This is because no matches were played in the last month of June in these three leagues as opposed to La Liga, which had matches in June.

In the case of a grouped Series, we return to the `nationsGrp` example and compute some statistics on the attendance figures for the country of the tournament winners:

```
In [297]: nationsGrp['Attendance'].agg({'Total':np.sum, 'Average':np.
mean, 'Deviation':np.std})
Out[297]:        Deviation    Average      Total
         Nation
         England    17091.31     66534.25    798411
         France     NaN          64400       64400
```

```
        Germany    13783.83    67583.29    473083

        Italy      17443.52    65761.25    789135

        Netherlands 16048.58   67489.0     404934

        Portugal   15632.86    49635.5     198542

        Romania    NaN         70000       70000

        Scotland   NaN         45000       45000

        Spain      27457.53    73477.15    955203

        Yugoslavia NaN         56000       56000
```

For a grouped Series, we can pass a list or `dict` of functions. In the preceding case, a `dict` was specified and the key values were used for the names of the columns in the resulting DataFrame. Note that in the case of groups of a single sample size, the standard deviation is undefined and `NaN` is the result—for example, Romania.

The transform() method

The `groupby-transform` function is used to perform transformation operations on a `groupby` object. For example, we could replace `NaN` values in the `groupby` object using the `fillna` method. The resulting object after using transform has the same size as the original `groupby` object. Let us consider a DataFrame showing the goals scored for each month in the four soccer leagues:

```
In[344]: goalStatsDF3=pd.read_csv('./goal_stats_euro_leagues_2012-13.
csv')

goalStatsDF3=goalStatsDF3.set_index(['Month'])

goalsScoredDF=goalStatsDF3.ix[goalStatsDF3['Stat']=='GoalsScored']


goalsScoredDF.iloc[:,1:]
Out[344]:        EPL   La Liga   Serie A   Bundesliga
Month
08/01/2012   57   60    21           23
09/01/2012   111   112    133          135
10/01/2012   95   88    97           77
11/01/2012   121   116    120          137
12/01/2012   183   109    125          72
01/01/2013   117   121    104          51
02/01/2013   87   110    100          101
03/01/2013   91   101    99           106
04/01/2013   105   127    102          104
```

```
05/01/2013    96    109    102         92
06/01/2013    NaN    80    NaN        NaN
```

We can see that for June 2013, the only league for which matches were played was `La Liga`, resulting in the `NaN` values for the other three leagues. Let us group the data by year:

```
In [336]: goalsScoredPerYearGrp=goalsScoredDF.groupby(lambda Month:
Month.split('/')[2])

          goalsScoredPerYearGrp.mean()
Out[336]:           EPL    La Liga    Serie A   Bundesliga
          2012     113.4    97          99.2      88.8
          2013      99.2   108         101.4      90.8
```

The preceding function makes use of a lambda function to obtain the year by splitting the `Month` variable on the / character and taking the third element of the resulting list.

If we do a count of the number of months per year during which matches were held in the various leagues, we have:

```
In [331]: goalsScoredPerYearGrp.count()
Out[331]:        EPL  La Liga  Serie A  Bundesliga
          2012    5     5        5          5
          2013    5     6        5          5
```

It is often undesirable to display data with missing values and one common method to resolve this situation would be to replace the missing values with the group mean. This can be achieved using the transform-groupby function. First, we must define the transformation using a lambda function and then apply this transformation using the transform method:

```
In [338]: fill_fcn = lambda x: x.fillna(x.mean())

          trans = goalsScoredPerYearGrp.transform(fill_fcn)

          tGroupedStats = trans.groupby(lambda Month:   Month.split('/')
[2])

          tGroupedStats.mean()
Out[338]:           EPL    La Liga    Serie A   Bundesliga
          2012     113.4    97          99.2      88.8
          2013      99.2   108         101.4      90.8
```

One thing to note from the preceding results is that replacing the `NaN` values with the group mean in the original group, keeps the group means unchanged in the transformed data.

However, when we do a count on the transformed group, we see that the number of matches has changed from five to six for the EPL, Serie A, and Bundesliga:

```
In [339]: tGroupedStats.count()
Out[339]:        EPL   La Liga   Serie A  Bundesliga
         2012    5     5         5        5
         2013    6     6         6        6
```

Filtering

The filter method enables us to apply filtering on a `groupby` object that results in a subset of the initial object. Here, we illustrate how to display the months of the season in which more than 100 goals were scored in each of the four leagues:

```
In [391]:  goalsScoredDF.groupby(level='Month').filter(lambda x:
                              np.all([x[col] > 100
           for col in goalsScoredDF.columns]))
Out[391]:           EPL   La Liga   Serie A  Bundesliga
         Month
         09/01/2012  111   112        133       135
         11/01/2012  121   116        120       137
         04/01/2013  105   127        102       104
```

Note the use of the `np.all` operator to ensure that the constraint is enforced for all the columns.

Merging and joining

There are various functions that can be used to merge and join pandas' data structures, which include the following functions:

- `concat`
- `append`

The concat function

The `concat` function is used to join multiple pandas' data structures along a specified axis and possibly perform union or intersection operations along other axes. The following command explains the `concat` function:

```
concat(objs, axis=0, , join='outer', join_axes=None, ignore_index=False,
       keys=None, levels=None, names=None, verify_integrity=False)
```

The synopsis of the elements of `concat` function are as follows:

- The `objs` function: A list or dictionary of Series, DataFrame, or Panel objects to be concatenated.
- The `axis` function: The axis along which the concatenation should be performed. `0` is the default value.
- The `join` function: The type of join to perform when handling indexes on other axes. The `'outer'` function is the default.
- The `join_axes` function: This is used to specify exact indexes for the remaining indexes instead of doing outer/inner join.
- The `keys` function: This specifies a list of keys to be used to construct a MultiIndex.

For an explanation of the remaining options, please refer to the documentation at `http://pandas.pydata.org/pandas-docs/stable/merging.html`.

Here is an illustration of the workings of `concat` using our stock price examples from earlier chapters:

```
In [53]: stockDataDF=pd.read_csv('./tech_stockprices.csv').set_index(
['Symbol']);stockDataDF
Out[53]:
```

Symbol	Closing price	EPS	Shares Outstanding(M)	P/E	Market Cap(B)	Beta
AAPL	501.53	40.32	892.45	12.44	447.59	0.84
AMZN	346.15	0.59	459.00	589.80	158.88	0.52
FB	61.48	0.59	2450.00	104.93	150.92	NaN
GOOG	1133.43	36.05	335.83	31.44	380.64	0.87
TWTR	65.25	-0.30	555.20	NaN	36.23	NaN
YHOO	34.90	1.27	1010.00	27.48	35.36	0.66

We now take various slices of the data:

```
In [83]: A=stockDataDF.ix[:4, ['Closing price', 'EPS']]; A
Out[83]:   Closing price  EPS

           Symbol

           AAPL      501.53      40.32

           AMZN      346.15       0.59

           FB         61.48       0.59

           GOOG     1133.43      36.05


In [84]: B=stockDataDF.ix[2:-2, ['P/E']];B
Out[84]:          P/E

         Symbol

         FB      104.93

         GOOG    31.44


In [85]: C=stockDataDF.ix[1:5, ['Market Cap(B)']];C
Out[85]:         Market Cap(B)

         Symbol

         AMZN    158.88

         FB      150.92

         GOOG    380.64

         TWTR    36.23
```

Here, we perform a concatenation by specifying an outer join, which concatenates and performs a union on all the three data frames, and includes entries that do not have values for all the columns by inserting NaN for such columns:

```
In [86]: pd.concat([A,B,C],axis=1) # outer join
Out[86]:  Closing price  EPS     P/E    Market Cap(B)

         AAPL    501.53     40.32  NaN    NaN

         AMZN    346.15      0.59  NaN    158.88

         FB       61.48      0.59  104.93 150.92

         GOOG   1133.43     36.05  31.44  380.64

         TWTR    NaN         NaN   NaN     36.23
```

We can also specify an inner join that does the concatenation, but only includes rows that contain values for all the columns in the final data frame by throwing out rows with missing columns, that is, it takes the intersection:

```
In [87]: pd.concat([A,B,C],axis=1, join='inner') # Inner join
Out[87]:       Closing price  EPS   P/E   Market Cap(B)
         Symbol
         FB      61.48     0.59 104.93  150.92
         GOOG    1133.43   36.05   31.44   380.64
```

The third case enables us to use the specific index from the original DataFrame to join on:

```
In [102]: pd.concat([A,B,C], axis=1, join_axes=[stockDataDF.index])
Out[102]:       Closing price  EPS     P/E    Market Cap(B)
         Symbol
         AAPL    501.53     40.32   NaN    NaN
         AMZN    346.15     0.59    NaN    158.88
         FB      61.48      0.59   104.93  150.92
         GOOG    1133.43    36.05   31.44  380.64
         TWTR    NaN        NaN     NaN    36.23
         YHOO    NaN        NaN     NaN    NaN
```

In this last case, we see that the row for YHOO was included even though it wasn't contained in any of the slices that were concatenated. In this case, however, the values for all the columns are NaN. Here is another illustration of concat, but this time, it is on random statistical distributions. Note that in the absence of an axis argument, the default axis of concatenation is 0:

```
In[135]: np.random.seed(100)
         normDF=pd.DataFrame(np.random.randn(3,4));normDF
Out[135]:    0      1       2       3
         0  -1.749765  0.342680  1.153036  -0.252436
         1   0.981321  0.514219  0.221180  -1.070043
         2  -0.189496  0.255001 -0.458027   0.435163

In [136]: binomDF=pd.DataFrame(np.random.binomial(100,0.5,(3,4)));binomDF
Out[136]:    0  1  2  3
         0  57  50  57     50
         1  48  56  49     43
         2  40  47  49     55
```

```
In [137]: poissonDF=pd.DataFrame(np.random.poisson(100,(3,4)));poissonDF
Out[137]:   0   1   2    3

         0  93  96  96   89

         1  76  96  104  103

         2  96  93  107  84
```

```
In [138]: rand_distribs=[normDF,binomDF,poissonDF]
In [140]: rand_distribsDF=pd.concat(rand_distribs,keys=['Normal',
'Binomial', 'Poisson']);rand_distribsDF
Out[140]:          0          1         2          3
  Normal   0  -1.749765   0.342680  1.153036  -0.252436

          1   0.981321   0.514219  0.221180  -1.070043

          2  -0.189496   0.255001 -0.458027   0.435163
  Binomial 0   57.00      50.00     57.00      50.00

          1   48.00      56.00     49.00      43.00

          2   40.00      47.00     49.00      55.00
  Poisson  0   93.00      96.00     96.00      89.00

          1   76.00      96.00    104.00     103.00

          2   96.00      93.00    107.00      84.00
```

Using append

The `append` function is a simpler version of `concat` that concatenates along `axis=0`. Here is an illustration of its use where we slice out the first two rows and first three columns of the `stockData` DataFrame:

```
In [145]: stockDataA=stockDataDF.ix[:2,:3]
          stockDataA
Out[145]:  Closing price  EPS    Shares Outstanding(M)
  Symbol
  AAPL     501.53   40.32   892.45
  AMZN     346.15   0.59    459.00
```

And the remaining rows:

```
In [147]: stockDataB=stockDataDF[2:]
        stockDataB
Out[147]:
    Closing price EPS Shares Outstanding(M)  P/E  Market Cap(B) Beta
```

```
Symbol

FB    61.48        0.59   2450.00           104.93 150.92   NaN

GOOG   1133.43    36.05   335.83            31.44  380.64   0.87

TWTR    65.25     -0.30   555.20              NaN   36.23   NaN

YHOO    34.90  1.27  1010.00        27.48   35.36   0.66
```

Now, we use `append` to combine the two data frames from the preceding commands:

```
In [161]:stockDataA.append(stockDataB)
Out[161]:
        Beta Closing price EPS MarketCap(B) P/E   Shares Outstanding(M)
        Symbol
        AMZN  NaN    346.15     0.59  NaN    NaN    459.00
        GOOG  NaN    1133.43   36.05  NaN    NaN    335.83
        FB    NaN    61.48      0.59  150.92 104.93 2450.00
        YHOO  27.48  34.90      1.27  35.36  0.66   1010.00
        TWTR  NaN    65.25     -0.30  36.23  NaN    555.20
        AAPL  12.44  501.53    40.32  0.84   447.59 892.45
```

In order to maintain the order of columns similar to the original DataFrame, we can apply the `reindex_axis` function:

```
In [151]: stockDataA.append(stockDataB).reindex_axis(stockDataDF.columns,
axis=1)
Out[151]:
        Closing price EPS Shares Outstanding(M)  P/E Market Cap(B) Beta
        Symbol
        AAPL   501.53  40.32  892.45           NaN  NaN      NaN
        AMZN   346.15  0.59   459.00           NaN  NaN      NaN
        FB     61.48   0.59   2450.00          104.93 150.92     NaN
        GOOG   1133.43 36.05  335.83           31.44  380.64     0.87
        TWTR   65.25  -0.30   555.20           NaN  36.23      NaN
        YHOO   34.90   1.27   1010.00          27.48  35.36      0.66
```

Note that for the first two rows, the value of the last two columns is `NaN`, since the first DataFrame only contained the first three columns. The `append` function does not work in places, but it returns a new DataFrame with the second DataFrame appended to the first.

Appending a single row to a DataFrame

We can append a single row to a DataFrame by passing a series or dictionary to the `append` method:

```
In [152]:
algos={'search':['DFS','BFS','Binary Search','Linear'],
       'sorting': ['Quicksort','Mergesort','Heapsort','Bubble Sort'],
      'machine learning':['RandomForest','K Nearest Neighbor','Logistic
Regression','K-Means Clustering']}
algoDF=pd.DataFrame(algos);algoDF
Out[152]: machine learning     search        sorting
        0    RandomForest          DFS       Quicksort
        1    K Nearest Neighbor   BFS        Mergesort
        2    Logistic Regression  Binary Search Heapsort
        3    K-Means Clustering    Linear        Bubble Sort


In [154]:
moreAlgos={'search': 'ShortestPath'  , 'sorting': 'Insertion Sort',
           'machine learning': 'Linear Regression'}
   algoDF.append(moreAlgos,ignore_index=True)

Out[154]: machine learning     search        sorting
        0    RandomForest          DFS       Quicksort
        1    K Nearest Neighbor    BFS         Mergesort
        2    Logistic Regression Binary Search Heapsort
        3    K-Means Clustering  Linear        Bubble Sort
        4    Linear Regression    ShortestPath  Insertion Sort
```

In order for this to work, you must pass the `ignore_index=True` argument so that the `index [0,1,2,3]` in `algoDF` is ignored.

SQL-like merging/joining of DataFrame objects

The `merge` function is used to obtain joins of two DataFrame objects similar to those used in SQL database queries. The DataFrame objects are analogous to SQL tables. The following command explains this:

```
merge(left, right, how='inner', on=None, left_on=None,
      right_on=None, left_index=False, right_index=False,
      sort=True, suffixes=('_x', '_y'), copy=True)
```

Following is the synopsis of `merge` function:

- The `left` argument: This is the first DataFrame object
- The `right` argument: This is the second DataFrame object
- The `how` argument: This is the type of join and can be inner, outer, left, or right. The default is inner.
- The `on` argument: This shows the names of columns to join on as join keys.
- The `left_on` and `right_on` arguments : This shows the left and right `DataFrame` column names to join on.
- The `left_index` and `right_index` arguments: This has a Boolean value. If this is `True`, use the left or right `DataFrame` index/row labels to join on.
- The `sort` argument: This has a Boolean value. The default `True` setting results in a lexicographical sorting. Setting the default value to `False` may improve performance.
- The `suffixes` argument: The tuple of string suffixes to be applied to overlapping columns. The defaults are `'_x'` and `'_y'`.
- The `copy` argument: The default `True` value causes data to be copied from the passed `DataFrame` objects.

The source of the preceding information can be found at `http://pandas.pydata.org/pandas-docs/stable/merging.html`.

Let us start to examine the use of merge by reading the U.S. stock index data into a `DataFrame`:

```
In [254]: USIndexDataDF=pd.read_csv('./us_index_data.csv')
          USIndexDataDF
Out[254]:     TradingDate  Nasdaq   S&P 500  Russell 2000  DJIA
          0   2014/01/30   4123.13  1794.19      1139.36  15848.61
          1   2014/01/31   4103.88  1782.59      1130.88  15698.85
          2   2014/02/03   3996.96  1741.89      1094.58  15372.80
          3   2014/02/04   4031.52  1755.20      1102.84  15445.24
          4   2014/02/05   4011.55  1751.64      1093.59  15440.23
          5   2014/02/06   4057.12  1773.43      1103.93  15628.53
```

The source of this information can be found at `http://finance.yahoo.com`.

We can obtain `slice1` of the data for rows 0 and 1 and the `Nasdaq` and `S&P 500` columns by using the following command:

```
In [255]: slice1=USIndexDataDF.ix[:1,:3]
          slice1
Out[255]:    TradingDate  Nasdaq        S&P 500
         0       2014/01/30  4123.13   1794.19
         1       2014/01/31  4103.88   1782.59
```

We can obtain `slice2` of the data for rows 0 and 1 and the `Russell 2000` and `DJIA` columns by using the following command:

```
In [256]: slice2=USIndexDataDF.ix[:1,[0,3,4]]
       slice2
Out[256]:    TradingDate  Russell 2000    DJIA
         0       2014/01/30  1139.36      15848.61
         1       2014/01/31  1130.88      15698.85
```

We can obtain `slice3` of the data for rows 1 and 2 and the `Nasdaq` and `S&P 500` columns by using the following command:

```
In [248]: slice3=USIndexDataDF.ix[[1,2],:3]
           slice3
Out[248]:    TradingDate      Nasdaq     S&P 500
         1   2014/01/31      4103.88   1782.59
         2   2014/02/03      3996.96   1741.89
```

We can now merge `slice1` and `slice2` as follows:

```
In [257]: pd.merge(slice1,slice2)
Out[257]:    TradingDate  Nasdaq   S&P 500   Russell 2000   DJIA
         0   2014/01/30   4123.13  1794.19   1139.36        15848.61
         1   2014/01/31    4103.88  1782.59   1130.88        15698.85
```

As you can see, this results in a combination of the columns in `slice1` and `slice2`. Since the `on` argument was not specified, the intersection of the columns in `slice1` and `slice2` was used which is `TradingDate` as the join column, and the rest of the columns from `slice1` and `slice2` were used to produce the output.

Note that in this case, passing a value for `how` has no effect on the result since the values of the `TradingDate` join key match for `slice1` and `slice2`.

We now merge `slice3` and `slice2` specifying `inner` as the value of the `how` argument:

```
In [258]: pd.merge(slice3,slice2,how='inner')
Out[258]:    TradingDate  Nasdaq       S&P 500   Russell 2000    DJIA
        0   2014/01/31   4103.88     1782.59     1130.88       15698.85
```

The `slice3` argument has values 2014/01/31 and 2014/02/03 unique values for `TradingDate`, and `slice2` has values 2014/01/30 and 2014/01/31 unique values for `TradingDate`.

The `merge` function uses the intersection of these values, which is 2014/01/31. This results in the single row result. Here, we specify `outer` as the value of the `how` argument:

```
In [269]: pd.merge(slice3,slice2,how='outer')
Out[269]:    TradingDate  Nasdaq       S&P 500   Russell 2000  DJIA
        0   2014/01/31   4103.88     1782.59     1130.88      15698.85
        1   2014/02/03   3996.96     1741.89     NaN            NaN
        2   2014/01/30      NaN      NaN         1139.36       15848.61
```

Specifying `outer` uses all the keys (union) from both DataFrames, which gives the three rows specified in the preceding output. Since not all the columns are present in the two DataFrames, the columns from the other DataFrame are `NaN` for each row in a DataFrame that is not part of the intersection.

Now, we specify `how='left'` as shown in the following command:

```
In [271]: pd.merge(slice3,slice2,how='left')
Out[271]:   TradingDate  Nasdaq   S&P 500   Russell 2000     DJIA
        0   2014/01/31   4103.88  1782.59   1130.88         15698.85
        1   2014/02/03   3996.96  1741.89   NaN             NaN
```

Here, we see that the keys from the left DataFrame `slice3` are used for the output. For columns that are not available in `slice3`, that is `Russell 2000` and `DJIA`, `NaN` are used for the row with `TradingDate` as 2014/02/03. This is equivalent to a SQL left outer join.

We specify `how='right'` in the following command:

```
In [270]: pd.merge(slice3,slice2,how='right')
Out[270]:   TradingDate  Nasdaq   S&P 500   Russell 2000  DJIA
        0   2014/01/31   4103.88  1782.59   1130.88      15698.85
        1   2014/01/30   NaN      NaN       1139.36      15848.61
```

This is the corollary to the `how='left'` keys from the right DataFrame `slice2` that are used. Therefore, rows with `TradingDate` as 2014/01/31 and 2014/01/30 are in the result. For columns that are not in `slice2`—`Nasdaq` and `S&P 500`—NaN are used.

This is equivalent to a SQL right outer join. For a simple explanation of how SQL joins work, please refer to `http://bit.ly/1yqR9vw`.

The join function

The `DataFrame.join` function is used to combine two DataFrames that have different columns with nothing in common. Essentially, this does a longitudinal join of two DataFrames. Here is an example:

```
In [274]: slice_NASD_SP=USIndexDataDF.ix[:3,:3]
          slice_NASD_SP
Out[274]:    TradingDate  Nasdaq  S&P 500
         0   2014/01/30   4123.13  1794.19
         1   2014/01/31   4103.88  1782.59
         2   2014/02/03   3996.96  1741.89
         3   2014/02/04   4031.52  1755.20


In [275]: slice_Russ_DJIA=USIndexDataDF.ix[:3,3:]
          slice_Russ_DJIA
Out[275]:    Russell 2000    DJIA
         0   1139.36        15848.61
         1   1130.88        15698.85
         2   1094.58        15372.80
         3   1102.84        15445.24
```

Here, we call the `join` operator, as follows:

```
In [276]: slice_NASD_SP.join(slice_Russ_DJIA)
Out[276]: TradingDate  Nasdaq  S&P 500  Russell 2000    DJIA
         0  2014/01/30  4123.13  1794.19    1139.36      15848.61
         1  2014/01/31  4103.88  1782.59    1130.88      15698.85
         2  2014/02/03  3996.96  1741.89    1094.58      15372.80
         3  2014/02/04  4031.52  1755.20    1102.84      15445.24
```

In this case, we see that the result is a combination of the columns from the two Dataframes. Let us see what happens when we try to use join with two DataFrames that have a column in common:

```
In [272]: slice1.join(slice2)
---------------------------------------------------------
Exception                     Traceback (most recent call last)
...


Exception: columns overlap: Index([u'TradingDate'], dtype=object)
```

This results in an exception due to overlapping columns. You can find more information on using `merge`, `concat`, and `join` operations in the official documentation page at `http://pandas.pydata.org/pandas-docs/stable/merging.html`.

Pivots and reshaping data

This section deals with how you can reshape data. Sometimes, data is stored in what is known as the *stacked* format. Here is an example of a stacked data using the `PlantGrowth` dataset:

```
In [344]: plantGrowthRawDF=pd.read_csv('./PlantGrowth.csv')
          plantGrowthRawDF
Out[344]:      observation   weight   group
          0    1             4.17     ctrl
          1    2             5.58     ctrl
          2    3             5.18     ctrl
          ...
          10   1             4.81     trt1
          11   2             4.17     trt1
          12   3             4.41     trt1
          ...
          20   1             6.31     trt2
          21   2             5.12     trt2
          22   3             5.54     trt2
```

This data consists of results from an experiment to compare dried weight yields of plants that were obtained under a **control** (**ctrl**) and two different treatment conditions (**trt1**, **trt2**). Suppose we wanted to do some analysis of this data by their group value. One way to do this would be to use a logical filter on the data frame:

```
In [346]: plantGrowthRawDF[plantGrowthRawDF['group']=='ctrl']
Out[346]:   observation   weight  group

       0    1       4.17   ctrl

       1    2       5.58   ctrl

       2    3       5.18   ctrl

       3    4       6.11   ctrl

       ...
```

This can be tedious, so we would instead like to pivot/unstack this data and display it in a form that is more conducive to analysis. We can do this using the `DataFrame.pivot` function as follows:

```
In [345]: plantGrowthRawDF.pivot(index='observation',columns='group',values='weight')
Out[345]: weight

       group   ctrl    trt1    trt2

       observation

       1       4.17    4.81    6.31

       2       5.58    4.17    5.12

       3       5.18    4.41    5.54

       4       6.11    3.59    5.50

       5       4.50    5.87    5.37

       6       4.61    3.83    5.29

       7       5.17    6.03    4.92

       8       4.53    4.89    6.15

       9       5.33    4.32    5.80

       10      5.14    4.69    5.26
```

Here, a DataFrame is created with columns corresponding to different values of a group, or in statistical parlance, levels of the factor. The same result can be achieved via the pandas `pivot_table` function, as follows:

```
In [427]: pd.pivot_table(plantGrowthRawDF,values='weight',
                         rows='observation', cols=['group'])
```

```
Out[427]:   group   ctrl    trt1    trt2

          observation

            1       4.17    4.81    6.31

            2       5.58    4.17    5.12

            3       5.18    4.41    5.54

            4       6.11    3.59    5.50

            5       4.50    5.87    5.37

            6       4.61    3.83    5.29

            7       5.17    6.03    4.92

            8       4.53    4.89    6.15

            9       5.33    4.32    5.80

            10      5.14    4.69    5.26
```

The key difference between the `pivot` and the `pivot_table` functions is that `pivot_table` allows the user to specify an aggregate function over which the values can be aggregated. So, for example, if we wish to obtain the mean for each group over the 10 observations, we would do the following, which would result in a Series:

```
In [430]: pd.pivot_table(plantGrowthRawDF,values='weight',cols=['group'],
aggfunc=np.mean)
```

```
Out[430]: group
    ctrl     5.032
    trt1     4.661
    trt2     5.526
    Name: weight, dtype: float64
```

The full synopsis of `pivot_table` is available at `http://bit.ly/1QomJ5A`. You can find more information and examples on its usage at: `http://bit.ly/1BYGsNn` and `https://www.youtube.com/watch?v=mCLuwCql5t4`.

Stacking and unstacking

In addition to the pivot functions, the `stack` and `unstack` functions are also available on Series and DataFrames, that work on objects containing MultiIndexes.

The stack() function

First, we set the group and observation column values to be the components of the row index respectively, which results in a MultiIndex:

```
In [349]: plantGrowthRawDF.set_index(['group','observation'])
Out[349]:                     weight

        group   observation
        ctrl    1             4.17
                2             5.58
                3             5.18

                ...
        trt1    1             4.81
                2             4.17
                3             4.41

                ...
        trt2    1             6.31
                2             5.12
                3             5.54

                ...
```

Here, we see that the row index consists of a MultiIndex on the group and observation with the weight column as the data value. Now, let us see what happens if we apply `unstack` to the `group` level:

```
In [351]: plantGrowthStackedDF.unstack(level='group')
Out[351]:                 weight

        group     ctrl  trt1    trt2
     observation
            1       4.17  4.81    6.31
            2       5.58  4.17    5.12
            3       5.18  4.41    5.54
            4       6.11  3.59    5.50
            5       4.50  5.87    5.37
            6       4.61  3.83    5.29
            7       5.17  6.03    4.92
            8       4.53  4.89    6.15
            9      5.33    4.32   5.80
           10      5.14    4.69   5.26
```

The following call is equivalent to the preceding one:
`plantGrowthStackedDF.unstack(level=0).`

Here, we can see that the DataFrame is pivoted and the group has now changed from a row index (headers) to a column index (headers), resulting in a more compact looking DataFrame. To understand what's going on in more detail, we have a MultiIndex as a row index initially on group, observation:

```
In [356]: plantGrowthStackedDF.index
Out[356]: MultiIndex
          [(u'ctrl', 1), (u'ctrl', 2), (u'ctrl', 3), (u'ctrl', 4),
(u'ctrl', 5),
           (u'ctrl', 6), (u'ctrl', 7), (u'ctrl', 8), (u'ctrl', 9),
(u'ctrl', 10),
           (u'trt1', 1), (u'trt1', 2), (u'trt1', 3), (u'trt1', 4),
(u'trt1', 5),
           (u'trt1', 6), (u'trt1', 7), (u'trt1', 8), (u'trt1', 9),
(u'trt1', 10),
           (u'trt2', 1), (u'trt2', 2), (u'trt2', 3), (u'trt2', 4),
(u'trt2', 5),
           (u'trt2', 6), (u'trt2', 7), (u'trt2', 8), (u'trt2', 9),
(u'trt2', 10)]


In [355]: plantGrowthStackedDF.columns
Out[355]: Index([u'weight'], dtype=object)
```

The unstacking operation removes the group from the row index, changing it into a single-level index:

```
In [357]: plantGrowthStackedDF.unstack(level='group').index
Out[357]: Int64Index([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=int64)
```

The MultiIndex is now on the columns:

```
In [352]: plantGrowthStackedDF.unstack(level='group').columns
Out[352]: MultiIndex
          [(u'weight', u'ctrl'), (u'weight', u'trt1'), (u'weight',
u'trt2')]
```

Let us see what happens when we call the reverse operation, `stack`:

```
In [366]: plantGrowthStackedDF.unstack(level=0).stack('group')
Out[366]:                   weight
          observation  group
          1   ctrl    4.17
              trt1    4.81
              trt2    6.31
          2   ctrl    5.58
              trt1    4.17
              trt2    5.12
          3   ctrl    5.18
              trt1    4.41
              trt2    5.54
          4   ctrl    6.11
              trt1    3.59
              trt2    5.50
          ...
          10  ctrl    5.14
              trt1    4.69
              trt2    5.26
```

Here, we see that what we get isn't the original stacked DataFrame since the stacked level—that is, `'group'`–becomes the new lowest level in a MultiIndex on the columns. In the original stacked DataFrame, `group` was the highest level. Here are the sequence of calls to `stack` and `unstack` that are exactly reversible. The `unstack()` function by default unstacks the last level, which is `observation`, which is shown as follows:

```
In [370]: plantGrowthStackedDF.unstack()
Out[370]:                                 weight
          observation     1    2    3    4    5    6    7    8    9    10
          group
          ctrl         4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14
          trt1         4.81 4.17 4.41 3.59 5.87  3.83 6.03 4.89 4.32 4.69
          trt2         6.31 5.12 5.54 5.50 5.37  5.29 4.92 6.15 5.80 5.26
```

The `stack()` function by default sets the stacked level as the lowest level in the resulting MultiIndex on the rows:

```
In [369]: plantGrowthStackedDF.unstack().stack()
Out[369]:              weight
       group  observation
       ctrl     1    4.17
                2    5.58
                3    5.18
       ...
                10   5.14
       trt1     1    4.81
                2    4.17
                3    4.41
       ...
                10   4.69
       trt2     1    6.31
                2    5.12
                3    5.54
       ...
                10   5.26
```

Other methods to reshape DataFrames

There are various other methods that are related to reshaping DataFrames; we'll discuss them here.

Using the melt function

The `melt` function enables us to transform a DataFrame by designating some of its columns as ID columns. This ensures that they will always stay as columns after any pivoting transformations. The remaining non-ID columns can be treated as variable and can be pivoted and become part of a name-value two column scheme. ID columns uniquely identify a row in the DataFrame.

The names of those non-ID columns can be customized by supplying the `var_name` and `value_name` parameters. The use of `melt` is perhaps best illustrated by an example, as follows:

```
In [385]: from pandas.core.reshape import melt
```

```
In [401]: USIndexDataDF[:2]
Out[401]:    TradingDate   Nasdaq     S&P 500   Russell 2000   DJIA
       0    2014/01/30    4123.13    1794.19   1139.36        15848.61
       1    2014/01/31    4103.88    1782.59   1130.88        15698.85
```

```
In [402]: melt(USIndexDataDF[:2], id_vars=['TradingDate'], var_
name='Index Name', value_name='Index Value')
Out[402]:
```

	TradingDate	Index Name	Index value
0	2014/01/30	Nasdaq	4123.13
1	2014/01/31	Nasdaq	4103.88
2	2014/01/30	S&P 500	1794.19
3	2014/01/31	S&P 500	1782.59
4	2014/01/30	Russell 2000	1139.36
5	2014/01/31	Russell 2000	1130.88
6	2014/01/30	DJIA	15848.61
7	2014/01/31	DJIA	15698.85

The pandas.get_dummies() function

This function is used to convert a categorical variable into an indicator DataFrame, which is essentially a truth table of possible values of the categorical variable. An example of this is the following command:

```
In [408]: melted=melt(USIndexDataDF[:2], id_vars=['TradingDate'], var_
name='Index Name', value_name='Index Value')
       melted
Out[408]:    TradingDate    Index Name     Index Value
       0     2014/01/30     Nasdaq         4123.13
       1     2014/01/31     Nasdaq         4103.88
       2     2014/01/30     S&P 500        1794.19
       3     2014/01/31     S&P 500        1782.59
       4     2014/01/30     Russell 2000   1139.36
```

```
5      2014/01/31    Russell 2000    1130.88
6      2014/01/30    DJIA           15848.61
7      2014/01/31    DJIA           15698.85


In [413]: pd.get_dummies(melted['Index Name'])
Out[413]:    DJIA  Nasdaq  Russell 2000   S&P 500
        0    0     1        0             0
        1    0     1        0             0
        2    0     0        0             1
        3    0     0        0             1
        4    0     0        1             0
        5    0     0        1             0
        6    1     0        0             0
        7    1     0        0             0
```

The source of the preceding data can be found at `http://vincentarelbundock.github.io/Rdatasets/csv/datasets/PlantGrowth.csv`.

Summary

In this chapter, we saw that there are various ways to rearrange data in pandas. We can group data using the `pandas.groupby` operator and the associated methods on `groupby` objects. We can merge and join `Series` and `DataFrame` objects using the `concat`, `append`, `merge`, and `join` functions. Lastly, we can reshape and create `pivot` tables using the `stack/unstack` and `pivot/pivot_table` functions. This is very useful functionality to present data for visualization or prepare data for input into other programs or algorithms.

In the next chapter, we will examine some useful tasks in data analysis for which we can apply pandas, such as processing time series data and how to handle missing values in our data.

To have more information on these topics on pandas, please take a look at the official documentation at `http://pandas.pydata.org/pandas-docs/stable/`.

6
Missing Data, Time Series, and Plotting Using Matplotlib

In this chapter, we take a tour of some topics that are necessary to develop expertise in using pandas. Knowledge of these topics is very useful for the preparation of data as input for programs or code that process data for analysis, prediction, or visualization. The topics that we'll discuss are as follows:

- Handling missing data
- Handling time series and dates
- Plotting using `matplotlib`

By the end of this chapter the user should be proficient in these critical areas.

Handling missing data

Missing data refers to data points that show up as NULL or N/A in our datasets for some reason; for example, we may have a time series that spans all calendar days of the month that shows the closing price of a stock for each day, and the closing price for nonbusiness days would show up as missing. An example of corrupted data would be a financial dataset that shows the activity date of a transaction in the wrong format; for example, YYYY-MM-DD instead of YYYYMMDD due to an error on the part of the data provider.

In the case of pandas, missing values are generally represented by the **NaN** value.

Other than appearing natively in the source dataset, missing values can be added to a dataset by an operation such as reindexing, or changing frequencies in the case of a time series:

```
In [84]: import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
         %matplotlib inline
In [85]: date_stngs = ['2014-05-01','2014-05-02',
                        '2014-05-05','2014-05-06','2014-05-07']
         tradeDates = pd.to_datetime(pd.Series(date_stngs))
In [86]: closingPrices=[531.35,527.93,527.81,515.14,509.96]
In [87]: googClosingPrices=pd.DataFrame(data=closingPrices,
                                        columns=['closingPrice'],
                                        index=tradeDates)
         googClosingPrices
Out[87]:            closingPrice
         tradeDates
         2014-05-01     531.35
         2014-05-02     527.93
         2014-05-05     527.81
         2014-05-06     515.14
         2014-05-07     509.96
         5 rows 1 columns
```

The source of the preceding data can be found at `http://yhoo.it/1dmJqW6`.

The pandas also provides an API to read stock data from various data providers, such as Yahoo:

```
In [29]: import pandas.io.data as web
In [32]: import datetime
         googPrices = web.get_data_yahoo("GOOG",
                      start=datetime.datetime(2014, 5, 1),
                      end=datetime.datetime(2014, 5, 7))
In [38]: googFinalPrices=pd.DataFrame(googPrices['Close'],
                                      index=tradeDates)
In [39]: googFinalPrices
Out[39]:           Close
```

```
       2014-05-01   531.34998

       2014-05-02   527.92999

       2014-05-05   527.81000

       2014-05-06   515.14001

       2014-05-07   509.95999
```

For more details, refer to `http://pandas.pydata.org/pandas-docs/stable/remote_data.html`.

We now have a time series that depicts the closing price of Google's stock from May 1, 2014 to May 7, 2014 with gaps in the date range since the trading only occur on business days. If we want to change the date range so that it shows calendar days (that is, along with the weekend), we can change the frequency of the time series index from business days to calendar days as follows:

```
In [90]: googClosingPricesCDays=googClosingPrices.asfreq('D')

         googClosingPricesCDays

Out[90]:     closingPrice

  2014-05-01   531.35

  2014-05-02   527.93

  2014-05-03   NaN

  2014-05-04   NaN

  2014-05-05   527.81

  2014-05-06   515.14

  2014-05-07   509.96

  7 rows 1 columns
```

Note that we have now introduced `NaN` values for the `closingPrice` for the weekend dates of May 3, 2014 and May 4, 2014.

We can check which values are missing by using the `isnull` and `notnull` functions as follows:

```
In [17]: googClosingPricesCDays.isnull()

Out[17]: closingPrice

    2014-05-01    False

    2014-05-02    False

    2014-05-03    True

    2014-05-04    True

    2014-05-05    False
```

```
    2014-05-06    False
    2014-05-07    False
    7 rows 1 columns


In [18]: googClosingPricesCDays.notnull()
Out[18]: closingPrice
    2014-05-01    True
    2014-05-02    True
    2014-05-03    False
    2014-05-04    False
    2014-05-05    True
    2014-05-06    True
    2014-05-07    True
    7 rows 1 columns
```

A Boolean DataFrame is returned in each case. In `datetime` and pandas Timestamps, missing values are represented by the `NaT` value. This is the equivalent of `NaN` in pandas for time-based types.

```
In [27]: tDates=tradeDates.copy()
         tDates[1]=np.NaN
         tDates[4]=np.NaN


In [28]: tDates
Out[28]: 0    2014-05-01
         1           NaT
         2    2014-05-05
         3    2014-05-06
         4           NaT
         Name: tradeDates, dtype: datetime64[ns]


In [4]: FBVolume=[82.34,54.11,45.99,55.86,78.5]
        TWTRVolume=[15.74,12.71,10.39,134.62,68.84]


In [5]: socialTradingVolume=pd.concat([pd.Series(FBVolume),
                      pd.Series(TWTRVolume),
                                tradeDates], axis=1,
```

```
              keys=['FB','TWTR','TradeDate'])
        socialTradingVolume
Out[5]:       FB        TWTR      TradeDate
        0     82.34     15.74     2014-05-01
        1     54.11     12.71     2014-05-02
        2     45.99     10.39     2014-05-05
        3     55.86     134.62    2014-05-06
        4     78.50     68.84     2014-05-07
        5 rows × 3 columns


In [6]: socialTradingVolTS=socialTradingVolume.set_index('TradeDate')
socialTradingVolTS
Out[6]:
  TradeDate     FB        TWTR
  2014-05-01    82.34     15.74
  2014-05-02    54.11     12.71
  2014-05-05    45.99     10.39
  2014-05-06    55.86     134.62
  2014-05-07    78.50     68.84
  5 rows × 2 columns


In [7]: socialTradingVolTSCal=socialTradingVolTS.asfreq('D')
        socialTradingVolTSCal
Out[7]:
              FB        TWTR
  2014-05-01  82.34     15.74
  2014-05-02  54.11     12.71
  2014-05-03  NaN       NaN
  2014-05-04  NaN       NaN
  2014-05-05  45.99     10.39
  2014-05-06  55.86     134.62
  2014-05-07  78.50     68.84
  7 rows × 2 columns
```

We can perform arithmetic operations on data containing missing values. For example, we can calculate the total trading volume (in millions of shares) across the two stocks for Facebook and Twitter as follows:

```
In [8]: socialTradingVolTSCal['FB']+socialTradingVolTSCal['TWTR']
Out[8]: 2014-05-01      98.08
        2014-05-02      66.82
        2014-05-03        NaN
        2014-05-04        NaN
        2014-05-05      56.38
        2014-05-06     190.48
        2014-05-07     147.34
        Freq: D, dtype: float64
```

By default, any operation performed on an object that contains missing values will return a missing value at that position as shown in the following command:

```
In [12]: pd.Series([1.0,np.NaN,5.9,6])+pd.Series([3,5,2,5.6])
Out[12]: 0      4.0
         1      NaN
         2      7.9
         3     11.6
         dtype: float64
In [13]: pd.Series([1.0,25.0,5.5,6])/pd.Series([3,np.NaN,2,5.6])
Out[13]: 0    0.333333
         1         NaN
         2    2.750000
         3    1.071429
         dtype: float64
```

There is a difference, however, in the way NumPy treats aggregate calculations versus what pandas does.

In pandas, the default is to treat the missing value as `0` and do the aggregate calculation, whereas for NumPy, `NaN` is returned if any of the values are missing. Here is an illustration:

```
In [15]: np.mean([1.0,np.NaN,5.9,6])
Out[15]: nan
```

```
In [16]: np.sum([1.0,np.NaN,5.9,6])
Out[16]: nan
```

However, if this data is in a pandas Series, we will get the following output:

```
In [17]: pd.Series([1.0,np.NaN,5.9,6]).sum()
Out[17]: 12.9
In [18]: pd.Series([1.0,np.NaN,5.9,6]).mean()
Out[18]: 4.2999999999999998
```

It is important to be aware of this difference in behavior between pandas and NumPy. However, if we wish to get NumPy to behave the same way as pandas, we can use the `np.nanmean` and `np.nansum` functions, which are illustrated as follows:

```
In [41]: np.nanmean([1.0,np.NaN,5.9,6])
Out[41]: 4.2999999999999998


In [43]: np.nansum([1.0,np.NaN,5.9,6])

Out[43]: 12.9
```

For more information on NumPy np.nan* aggregation functions, refer to `http://docs.scipy.org/doc/numpy-dev/reference/routines.statistics.html`.

Handling missing values

There are various ways to handle missing values, which are as follows:

1. By using the `fillna()` function to fill in the NA values. This is an example:

   ```
   In [19]: socialTradingVolTSCal
   Out[19]:      FB     TWTR
            2014-05-01  82.34   15.74
            2014-05-02  54.11   12.71
            2014-05-03  NaN     NaN
            2014-05-04  NaN     NaN
            2014-05-05  45.99   10.39
            2014-05-06  55.86   134.62
            2014-05-07  78.50   68.84
            7 rows × 2 columns
   ```

```
In [20]: socialTradingVolTSCal.fillna(100)
Out[20]:                FB       TWTR
         2014-05-01   82.34     15.74
         2014-05-02   54.11     12.71
         2014-05-03   100.00    100.00
         2014-05-04   100.00    100.00
         2014-05-05   45.99     10.39
         2014-05-06   55.86     134.62
         2014-05-07   78.50     68.84
         7 rows × 2 columns
```

We can also fill forward or backward values using the `ffill` or `bfill` arguments:

```
In [23]: socialTradingVolTSCal.fillna(method='ffill')
Out[23]:                FB       TWTR
         2014-05-01   82.34     15.74
         2014-05-02   54.11     12.71
         2014-05-03   54.11     12.71
         2014-05-04   54.11     12.71
         2014-05-05   45.99     10.39
         2014-05-06   55.86     134.62
         2014-05-07   78.50     68.84
         7 rows × 2 columns


In [24]: socialTradingVolTSCal.fillna(method='bfill')
Out[24]:                FB       TWTR
         2014-05-01   82.34     15.74
         2014-05-02   54.11     12.71
         2014-05-03   45.99     10.39
         2014-05-04   45.99     10.39
         2014-05-05   45.99     10.39
         2014-05-06   55.86     134.62
         2014-05-07   78.50     68.84
         7 rows × 2 columns
```

The `pad` method is an alternative name for `ffill`. For more details, you can go to `http://bit.ly/1f4jvDq`.

2. By using the `dropna()` function to drop/delete rows and columns with missing values. The following is an example of this:

```
In [21]: socialTradingVolTSCal.dropna()

Out[21]:         FB       TWTR
  2014-05-01  82.34    15.74
  2014-05-02  54.11    12.71
  2014-05-05  45.99    10.39
  2014-05-06  55.86    134.62
  2014-05-07  78.50    68.84
  5 rows × 2 columns
```

3. We can also interpolate and fill in the missing values by using the `interpolate()` function, as explained in the following commands:

```
In [27]: pd.set_option('display.precision',4)
         socialTradingVolTSCal.interpolate()

Out[27]:         FB       TWTR
  2014-05-01  82.340   15.740
  2014-05-02  54.110   12.710
  2014-05-03  51.403   11.937
  2014-05-04  48.697   11.163
  2014-05-05  45.990   10.390
  2014-05-06  55.860   134.620
  2014-05-07  78.500   68.840
  7 rows × 2 columns
```

The `interpolate()` function also takes an argument—*method* that denotes the method. These methods include linear, quadratic, cubic spline, and so on. You can obtain more information from the official documentation at `http://pandas.pydata.org/pandas-docs/stable/missing_data.html#interpolation`.

Handling time series

In this section, we show you how to handle time series data. We will start by showing how to create time series data using the data read in from a `csv` file.

Reading in time series data

Here, we demonstrate the various ways to read in time series data:

```
In [7]: ibmData=pd.read_csv('ibm-common-stock-closing-prices-1959_1960.
csv')

  ibmData.head()
Out[7]:    TradeDate     closingPrice
       0    1959-06-29    445
       1    1959-06-30    448
       2    1959-07-01    450
       3    1959-07-02    447
       4    1959-07-06    451
       5 rows 2 columns
```

The source of this information can be found at `http://datamarket.com`.

We would like the `TradeDate` column to be a series of `datetime` values so that we can index it and create a time series. Let us first check the type of values in the `TradeDate` series:

```
In [16]: type(ibmData['TradeDate'])
Out[16]: pandas.core.series.Series
In [12]: type(ibmData['TradeDate'][0])
Out[12]: str
```

Next, we convert it to a `Timestamp` type:

```
In [17]: ibmData['TradeDate']=pd.to_datetime(ibmData['TradeDate'])
         type(ibmData['TradeDate'][0])
Out[17]: pandas.tslib.Timestamp
```

We can now use the `TradeDate` column as an index:

```
In [113]: #Convert DataFrame to TimeSeries
          #Resampling creates NaN rows for weekend dates, hence use
dropna
  ibmTS=ibmData.set_index('TradeDate').resample('D')['closingPrice'].
dropna()
    ibmTS
Out[113]: TradeDate
        1959-06-29    445
        1959-06-30    448
```

```
        1959-07-01     450

        1959-07-02     447

 1959-07-06    451

 ...

 Name: closingPrice, Length: 255
```

DateOffset and TimeDelta objects

A `DateOffset` object represents a change or offset in time. The key features of a `DateOffset` object are as follows:

- This can be added/subtracted to/from a `datetime` object to obtain a shifted date
- This can be multiplied by an integer (positive or negative) so that the increment can be applied multiple times
- This has the rollforward and rollback methods to move a date forward to the next offset date or backward to the previous offset date

We illustrate how we use a DateOffset object as follows:

```
In [371]: xmasDay=pd.datetime(2014,12,25)

          xmasDay

Out[371]: datetime.datetime(2014, 12, 25, 0, 0)


In [373]: boxingDay=xmasDay+pd.DateOffset(days=1)

          boxingDay

Out[373]: Timestamp('2014-12-26 00:00:00', tz=None)


In [390}: today=pd.datetime.now()

          today

Out[390]: datetime.datetime(2014, 5, 31, 13, 7, 36, 440060)
```

Note that `datetime.datetime` is different from `pd.Timestamp`. The former is a Python class and is inefficient, while the latter is based on the `numpy.datetime64` datatype. The `pd.DateOffset` object works with `pd.Timestamp` and adding it to a `datetime.datetime` function casts that object into a `pd.Timestamp` object.

The following illustrates the command for one week from today:

```
In [392]: today+pd.DateOffset(weeks=1)

Out[392]: Timestamp('2014-06-07 13:07:36.440060', tz=None)
```

The following illustrates the command for five years from today:

```
In [394]: today+2*pd.DateOffset(years=2, months=6)
Out[394]: Timestamp('2019-05-30 13:07:36.440060', tz=None)
```

Here is an example of using the `rollforward` functionality. `QuarterBegin` is a `DateOffset` object that is used to increment a given `datetime` object to the start of the next calendar quarter:

```
In [18]: lastDay=pd.datetime(2013,12,31)
In [24]: from pandas.tseries.offsets import QuarterBegin

        dtoffset=QuarterBegin()

        lastDay+dtoffset
Out[24]: Timestamp('2014-03-01 00:00:00', tz=None)


In [25]: dtoffset.rollforward(lastDay)
Out[25]: Timestamp('2014-03-01 00:00:00', tz=None)
```

Thus, we can see that the next quarter after December 31, 2013 starts on March 1, 2014. `Timedeltas` are similar to `DateOffsets` but work with `datetime.datetime` objects. The use of these has been explained by the following command:

```
In [40]: weekDelta=datetime.timedelta(weeks=1)

        weekDelta
Out[40]: datetime.timedelta(7)


In [39]: today=pd.datetime.now()

        today
Out[39]: datetime.datetime (2014, 6, 2, 3, 56, 0, 600309)


In [41]: today+weekDelta
Out[41]: datetime.datetime (2014, 6, 9, 3, 56,0, 600309)
```

Time series-related instance methods

In this section, we explore various methods for Time Series objects such as shifting, frequency conversion, and resampling.

Shifting/lagging

Sometimes, we may wish to shift the values in a Time Series backward or forward in time. One possible scenario is when a dataset contains the list of start dates for last year's new employees in a firm, and the company's human resource program wishes to shift these dates forward by one year so that the employees' benefits can be activated. We can do this by using the `shift()` function as follows:

```
In [117]: ibmTS.shift(3)
Out[117]: TradeDate
    1959-06-29    NaN
    1959-06-30    NaN
    1959-07-01    NaN
    1959-07-02    445
    1959-07-06    448
    1959-07-07    450
    1959-07-08    447
    ...
```

This shifts all the calendar days. However, if we wish to shift only business days, we must use the following command:

```
In [119]: ibmTS.shift(3, freq=pd.datetools.bday)
Out[119]: TradeDate
    1959-07-02    445
    1959-07-03    448
    1959-07-06    450
    1959-07-07    447
    1959-07-09    451
```

In the preceding snippet, we have specified the `freq` argument to shift; this tells the function to shift only the business days. The `shift` function has a `freq` argument whose value can be a `DateOffset` class, `timedelta`-like object, or an offset alias. Thus, using `ibmTS.shift(3, freq='B')` would also produce the same result.

Frequency conversion

We can use the `asfreq` function to change frequencies, as explained:

```
In [131]: # Frequency conversion using asfreq
          ibmTS.asfreq('BM')
Out[131]: 1959-06-30    448
```

```
1959-07-31    428
1959-08-31    425
1959-09-30    411
1959-10-30    411
1959-11-30    428
1959-12-31    439
1960-01-29    418
1960-02-29    419
1960-03-31    445
1960-04-29    453
1960-05-31    504
1960-06-30    522
Freq: BM, Name: closingPrice, dtype: float64
```

In this case, we just obtain the values corresponding to the last day of the month from the `ibmTS` time series. Here, `bm` stands for business month end frequency. For a list of all possible frequency aliases, go to `http://bit.ly/1cMI3iA`.

If we specify a frequency that is smaller than the granularity of the data, the gaps will be filled in with `NaN` values:

```
In [132]: ibmTS.asfreq('H')
Out[132]: 1959-06-29 00:00:00    445
    1959-06-29 01:00:00    NaN
    1959-06-29 02:00:00    NaN
    1959-06-29 03:00:00    NaN
    ...
    1960-06-29 23:00:00    NaN
    1960-06-30 00:00:00    522
    Freq: H, Name: closingPrice, Length: 8809
```

We can also apply the `asfreq` method to the `Period` and `PeriodIndex` objects similar to how we do for the `datetime` and `Timestamp` objects. `Period` and `PeriodIndex` are introduced later and are used to represent time intervals.

The `asfreq` method accepts a method argument that allows you to forward fill (`ffill`) or back fill the gaps, similar to `fillna`:

```
In [140]: ibmTS.asfreq('H', method='ffill')
Out[140]: 1959-06-29 00:00:00    445
    1959-06-29 01:00:00    445
```

```
1959-06-29 02:00:00    445

1959-06-29 03:00:00    445

...

1960-06-29 23:00:00    522

1960-06-30 00:00:00    522

Freq: H, Name: closingPrice, Length: 8809
```

Resampling of data

The `TimeSeries.resample` function enables us to summarize/aggregate more granular data based on a sampling interval and a sampling function.

Downsampling is a term that originates from digital signal processing and refers to the process of reducing the sampling rate of a signal. In the case of data, we use it to reduce the amount of data that we wish to process.

The opposite process is **upsampling**, which is used to increase the amount of data to be processed and requires interpolation to obtain the intermediate data points. For more information on downsampling and upsampling, refer to *Practical Applications of Upsampling and Downsampling* at `http://bit.ly/1JC95HD` and *Downsampling Time Series for Visual Representation* at `http://bit.ly/1zrExVP`.

Here, we examine some tick data for use in resampling. Before we examine the data, we need to prepare it. In doing so, we will learn some useful techniques for time series data, which are as follows:

- Epoch Timestamps
- Timezone handling

Here is an example that uses tick data for stock prices of Google for Tuesday, May 27, 2014:

```
In [150]: googTickData=pd.read_csv('./GOOG_tickdata_20140527.csv')
In [151]: googTickData.head()
Out[151]:     Timestamp    close    high     low    open    volume
          0    1401197402  555.008 556.41  554.35 556.38   81100
          1    1401197460  556.250 556.30  555.25 555.25   18500
          2    1401197526  556.730 556.75  556.05 556.39   9900
          3    1401197582  557.480 557.67  556.73 556.73   14700
          4    1401197642  558.155 558.66  557.48 557.59   15700
          5 rows 6 columns
```

The source for the preceding data can be found at `http://bit.ly/1MKBwlB`.

As you can see from the preceding section, we have a Timestamp column along with the columns for the close, high, low, and opening prices and the volume of trades of the Google stock.

So, why does the Timestamp column seem a bit strange? Well, tick data Timestamps are generally expressed in epoch time (for more information, refer to `http://en.wikipedia.org/wiki/Unix_epoch`) as a more compact means of storage. We'll need to convert this into a more human-readable time, and we can do this as follows:

```
In [201]: googTickData['tstamp']=pd.to_datetime(googTickData['Timestamp']
,unit='s',utc=True)


In [209]: googTickData.head()
Out[209]:

     Timestamp    close    high    low     open    volume tstamp
  0  14011974020  555.008  556.41  554.35  556.38  81100  2014-05-27 13:30:02
  1  1401197460   556.250  556.30  555.25  555.25  18500  2014-05-27 13:31:00
  2  1401197526   556.730  556.75  556.05  556.39  9900   2014-05-27 13:32:06
  3  1401197582   557.480  557.67  556.73  556.73  14700  2014-05-27 13:33:02
  4  1401197642   558.155  558.66  557.48  557.59  15700  2014-05-27 13:34:02
  5 rows 7 columns
```

We would now like to make the `tstamp` column, as the index and eliminate the epoch Timestamp column:

```
In [210]: googTickTS=googTickData.set_index('tstamp')
          googTickTS=googTickTS.drop('Timestamp',axis=1)
          googTickTS.head()
Out[210]:
          tstamp                 close    high    low     open    volume
          2014-05-27 13:30:02    555.008  556.41  554.35  556.38  811000
          2014-05-27 13:31:00    556.250  556.30  555.25  555.25  18500
          2014-05-27 13:32:06    556.730  556.75  556.05  556.39  9900
          2014-05-27 13:33:02    557.480  557.67  556.73  556.73  14700
          2014-05-27 13:34:02    558.155  558.66  557.48  557.59  15700
          5 rows 5 columns
```

Note that the tstamp index column has the times in UTC, and we can convert it to US/Eastern time using two operators—`tz_localize` and `tz_convert`:

```
In [211]: googTickTS.index=googTickTS.index.tz_localize('UTC').tz_
convert('US/Eastern')
```

```
In [212]: googTickTS.head()
Out[212]:
```

tstamp	close	high	low	open	volume
2014-05-27 09:30:02-04:00	555.008	556.41	554.35	556.38	81100
2014-05-27 09:31:00-04:00	556.250	556.30	555.25	555.25	18500
2014-05-27 09:32:06-04:00	556.730	556.75	556.05	556.39	9900
2014-05-27 09:33:02-04:00	557.480	557.67	556.73	556.73	14700
2014-05-27 09:34:02-04:00	558.155	558.66	557.48	557.59	15700

5 rows 5 columns

```
In [213]: googTickTS.tail()
Out[213]:
```

tstamp	close	high	low	open	volume
2014-05-27 15:56:00-04:00	565.4300	565.48	565.30	565.385	14300
2014-05-27 15:57:00-04:00	565.3050	565.46	565.20	565.400	14700
2014-05-27 15:58:00-04:00	565.1101	565.31	565.10	565.310	23200
2014-05-27 15:59:00-04:00	565.9400	566.00	565.08	565.230	55600
2014-05-27 16:00:00-04:00	565.9500	565.95	565.95	565.950	126000

5 rows 5 columns

```
In [214]: len(googTickTS)
Out[214]: 390
```

From the preceding output, we can see ticks for every minute of the trading day—from 9:30 a.m., when the stock market opens, until 4:00 p.m., when it closes. This results in 390 rows in the dataset since there are 390 minutes between 9:30 a.m. and 4:00 p.m.

Suppose we want to obtain a snapshot every 5 minutes instead of every minute? We can achieve this by using downsampling as follows:

```
In [216]: googTickTS.resample('5Min').head(6)
Out[216]:                close      high    low     open      volume    tstamp
2014-05-27 09:30:00-04:00 556.72460 557.15800 555.97200 556.46800 27980
2014-05-27 09:35:00-04:00 556.93648 557.64800 556.85100 557.34200  24620
2014-05-27 09:40:00-04:00 556.48600 556.79994 556.27700 556.60678   8620
2014-05-27 09:45:00-04:00 557.05300 557.27600 556.73800 556.96600   9720
2014-05-27 09:50:00-04:00  556.66200  556.93596  556.46400  556.80326
14560
2014-05-27 09:55:00-04:00  555.96580  556.35400  555.85800  556.23600
12400
6 rows 5 columns
```

The default function used for resampling is the mean. However, we can also specify other functions, such as the minimum, and we can do this via the `how` parameter to resample:

```
In [245]: googTickTS.resample('10Min', how=np.min).head(4)
Out[245]:           close    high       low   open   volume
tstamp
2014-05-27 09:30:00-04:00   555.008  556.3000  554.35  555.25   9900
2014-05-27 09:40:00-04:00   556.190  556.5600  556.13  556.35   3500
2014-05-27 09:50:00-04:00   554.770  555.5500  554.77  555.55   3400
2014-05-27 10:00:00-04:00   554.580  554.9847  554.45  554.58   1800
```

Various function names can be passed to the `how` parameter, such as `sum`, `ohlc`, `max`, `min`, `std`, `mean`, `median`, `first`, and `last`.

The `ohlc` function that returns *open-high-low-close* values on time series data that is; the first, maximum, minimum, and last values. To specify whether the left or right interval is closed, we can pass the `closed` parameter as follows:

```
In [254]: pd.set_option('display.precision',5)
          googTickTS.resample('5Min', closed='right').tail(3)
Out[254]:                      close      high  low      open        volume
tstamp
2014-05-27 15:45:00-04:00   564.3167  564.3733   564.1075  564.1700
12816.6667
```

```
2014-05-27 15:50:00-04:00    565.1128   565.1725    565.0090   565.0650
13325.0000

2014-05-27 15:55:00-04:00    565.5158   565.6033    565.3083   565.4158
40933.3333

3 rows 5 columns
```

Thus, in the preceding command, we can see that the last row shows the tick at 15:55 instead of 16:00.

For upsampling, we need to specify a fill method to determine how the gaps should be filled via the `fill_method` parameter:

```
In [263]: googTickTS[:3].resample('30s', fill_method='ffill')
Out[263]:      close     high     low    open  volume     tstamp
    2014-05-27 09:30:00-04:00   555.008   556.41   554.35   556.38    81100
    2014-05-27 09:30:30-04:00   555.008   556.41   554.35   556.38    81100
    2014-05-27 09:31:00-04:00   556.250   556.30   555.25   555.25    18500
    2014-05-27 09:31:30-04:00   556.250   556.30   555.25   555.25    18500
    2014-05-27 09:32:00-04:00   556.730   556.75   556.05   556.39    9900
    5 rows 5 columns



In [264]: googTickTS[:3].resample('30s', fill_method='bfill')
Out[264]:
                close     high     low   open   volume     tstamp
    2014-05-27 09:30:00-04:00   555.008    556.41   554.35   556.38    81100
    2014-05-27 09:30:30-04:00   556.250    556.30   555.25   555.25    18500
    2014-05-27 09:31:00-04:00   556.250    556.30   555.25   555.25    18500
    2014-05-27 09:31:30-04:00   556.730    556.75   556.05   556.39    9900
    2014-05-27 09:32:00-04:00   556.730    556.75   556.05   556.39    9900
    5 rows 5 columns
```

Unfortunately, the `fill_method` parameter currently supports only two methods—forward fill and back fill. An interpolation method would be valuable.

Aliases for Time Series frequencies

To specify offsets, a number of aliases are available; some of the most commonly used ones are as follows:

- **B, BM**: This stands for business day, business month. These are the working days of the month, that is, any day that is not a holiday or a weekend.

- **D, W, M, Q, A**: It stands for calendar day, week, month, quarter, year-end.

- **H, T, S, L, U**: It stands for hour, minute, second, millisecond, and microsecond.

These aliases can also be combined. In the following case, we resample every 7 minutes and 30 seconds:

```
In [267]: googTickTS.resample('7T30S').head(5)
Out[267]:

                          close     high     low     open     volume
tstamp
2014-05-27 09:30:00-04:00 556.8266 557.4362 556.3144 556.8800 28075.0
2014-05-27 09:37:30-04:00 556.5889 556.9342 556.4264 556.7206 11642.9
2014-05-27 09:45:00-04:00 556.9921 557.2185 556.7171 556.9871  9800.0
2014-05-27 09:52:30-04:00 556.1824 556.5375 556.0350 556.3896 14350.0
2014-05-27 10:00:00-04:00 555.2111 555.4368 554.8288 554.9675 12512.5

5 rows x 5 columns
```

Suffixes can be applied to the frequency aliases to specify when in a frequency period to start. These are known as anchoring offsets:

- W - SUN, MON, ... for example, W-TUE indicates a weekly frequency starting on a Tuesday.

- Q - JAN, FEB, ... DEC for example, Q-MAY indicates a quarterly frequency with the year-end in May.

- A - JAN, FEB, ... DEC for example, A-MAY indicates an annual frequency with the year-end in May.

These offsets can be used as arguments to the `date_range` and `bdate_range` functions as well as constructors for index types such as `PeriodIndex` and `DatetimeIndex`. A comprehensive discussion on this can be found in the pandas documentation at `http://pandas.pydata.org/pandas-docs/stable/timeseries.html#`.

Time series concepts and datatypes

When dealing with time series, there are two main concepts that you have to consider: points in time and ranges, or time spans. In pandas, the former is represented by the Timestamp datatype, which is equivalent to Python's `datetime.datetime` (`datetime`) datatype and is interchangeable with it. The latter (time span) is represented by the Period datatype, which is specific to pandas.

Each of these datatypes has index datatypes associated with them: `DatetimeIndex` for `Timestamp`/`Datetime` and `PeriodIndex` for `Period`. These index datatypes are basically subtypes of `numpy.ndarray` that contain the corresponding Timestamp and Period datatypes and can be used as indexes for Series and DataFrame objects.

Period and PeriodIndex

The `Period` datatype is used to represent a range or span of time. Here are a few examples:

```
# Period representing May 2014
In [287]: pd.Period('2014', freq='A-MAY')
Out[287]: Period('2014', 'A-MAY')


# Period representing specific day – June 11, 2014
In [292]: pd.Period('06/11/2014')
Out[292]: Period('2014-06-11', 'D')


# Period representing 11AM, Nov 11, 1918
In [298]: pd.Period('11/11/1918 11:00',freq='H')
Out[298]: Period('1918-11-11 11:00', 'H')
```

We can add integers to `Periods` which advances the period by the requisite number of unit of the frequency:

```
In [299]: pd.Period('06/30/2014')+4
Out[299]: Period('2014-07-04', 'D')


In [303]: pd.Period('11/11/1918 11:00',freq='H') - 48
Out[303]: Period('1918-11-09 11:00', 'H')
```

We can also calculate the difference between two `Periods` and return the number of units of frequency between them:

```
In [304]: pd.Period('2014-04', freq='M')-pd.Period('2013-02', freq='M')
Out[304]: 14
```

PeriodIndex

A `PeriodIndex` object, which is an `index` type for a `Period` object, can be created in two ways:

1. From a series of `Period` objects using the `period_range` function an analogue of `date_range`:

   ```
   In [305]: perRng=pd.period_range('02/01/2014','02/06/2014',freq=
   'D')

             perRng
   Out[305]: <class 'pandas.tseries.period.PeriodIndex'>
             freq: D
             [2014-02-01, ..., 2014-02-06]
             length: 6


   In [306]: type(perRng[:2])
   Out[306]: pandas.tseries.period.PeriodIndex


   In [307]: perRng[:2]
   Out[307]: <class 'pandas.tseries.period.PeriodIndex'>
              freq: D
             [2014-02-01, 2014-02-02]
   ```

   As we can confirm from the preceding command, when you pull the covers, a `PeriodIndex` function is really an `ndarray` of `Period` objects underneath.

2. It can also be done via a direct call to the `Period` constructor:

   ```
   In [312]: JulyPeriod=pd.PeriodIndex(['07/01/2014','07/31/2014'],
   freq='D')
        JulyPeriod
   Out[312]: <class 'pandas.tseries.period.PeriodIndex'>
        freq: D
        [2014-07-01, 2014-07-31]
   ```

The difference between the two approaches, as can be seen from the preceding output, is that `period_range` fills in the resulting ndarray, but the `Period` constructor does not and you have to specify all the values that should be in the index.

Conversions between Time Series datatypes

We can convert the `Period` and `PeriodIndex` datatypes to `Datetime`/`Timestamp` and `DatetimeIndex` datatypes via the `to_period` and `to_timestamp` functions, as follows:

```
In [339]: worldCupFinal=pd.to_datetime('07/13/2014',
                                         errors='raise')
          worldCupFinal
  Out[339]: Timestamp('2014-07-13 00:00:00')


In [340]: worldCupFinal.to_period('D')
  Out[340]: Period('2014-07-13', 'D')


In [342]: worldCupKickoff=pd.Period('06/12/2014','D')
          worldCupKickoff
Out[342]: Period('2014-06-12', 'D')
In [345]: worldCupKickoff.to_timestamp()
Out[345]: Timestamp('2014-06-12 00:00:00', tz=None)


In [346]: worldCupDays=pd.date_range('06/12/2014',periods=32,
                                      freq='D')
          worldCupDays
Out[346]: <class 'pandas.tseries.index.DatetimeIndex'>
    [2014-06-12, ..., 2014-07-13]
    Length: 32, Freq: D, Timezone: None


In [347]: worldCupDays.to_period()
Out[347]: <class 'pandas.tseries.period.PeriodIndex'>
    freq: D
    [2014-06-12, ..., 2014-07-13]
    length: 32
```

A summary of Time Series-related objects

The following table gives a summary of Time Series-related objects:

Object	Summary
`datetime.datetime`	This is a Standard Python `datetime` class
`Timestamp`	This is a pandas class derived from `datetime.datetime`
`DatetimeIndex`	This is a pandas class and is implemented as an immutable `numpy.ndarray` of the `Timestamp`/`datetime` objects
`Period`	This is a pandas class representing a time period
`PeriodIndex`	This is a pandas class and is implemented as an immutable `numpy.ndarray` of `Period` objects
`timedelta`	This is a Python class expressing the difference between two `datetime.datetime` instances. It is implemented as `datetime.timedelta`
`relativedelta`	Implemented as `dateutil.relativedelta`. dateutil is an extension to the standard Python datetime module. It provides extra functionality such as timedeltas that are expressed in units larger than 1 day.
`DateOffset`	This is a pandas class representing a regular frequency increment. It has similar functionalty to `dateutil.relativedelta`.

Plotting using matplotlib

This section provides a brief introduction to plotting in pandas using `matplotlib`. The `matplotlib api` is imported using the standard convention, as shown in the following command:

```
In [1]: import matplotlib.pyplot as plt
```

Series and DataFrame have a plot method, which is simply a wrapper around `plt.plot`. Here, we will examine how we can do a simple plot of a sine and cosine function. Suppose we wished to plot the following functions over the interval pi to pi:

- *f(x) = cos(x) + sin (x)*
- *g(x) = cos (x) - sin (x)*

This gives the following interval:

```
In [51]: import numpy as np
In [52]: X = np.linspace(-np.pi, np.pi, 256,endpoint=True)


In [54]: f,g = np.cos(X)+np.sin(X), np.sin(X)-np.cos(X)
In [61]: f_ser=pd.Series(f)
         g_ser=pd.Series(g)



In [31]: plotDF=pd.concat([f_ser,g_ser],axis=1)
         plotDF.index=X
         plotDF.columns=['sin(x)+cos(x)','sin(x)-cos(x)']
         plotDF.head()
Out[31]:   sin(x)+cos(x)   sin(x)-cos(x)
-3.141593   -1.000000    1.000000
-3.116953   -1.024334    0.975059
-3.092313   -1.048046    0.949526
-3.067673   -1.071122    0.923417
-3.043033   -1.093547    0.896747
5 rows × 2 columns
```

We can now plot the DataFrame using the `plot()` command and the `plt.show()` command to display it:

```
In [94]: plotDF.plot()
         plt.show()


We can apply a title to the plot as follows:
In [95]: plotDF.columns=['f(x)','g(x)']
         plotDF.plot(title='Plot of f(x)=sin(x)+cos(x), \n
g(x)=sinx(x)-cos(x)')
         plt.show()
```

The following is the output of the preceding command:
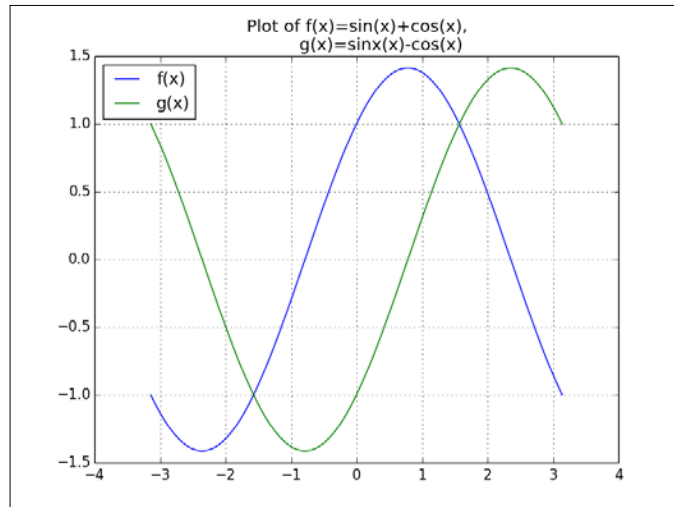
We can also plot the two series (functions) separately in different subplots using the following command:

```
In [96]: plotDF.plot(subplots=True, figsize=(6,6))
         plt.show()
```

The following is the output of the preceding command:
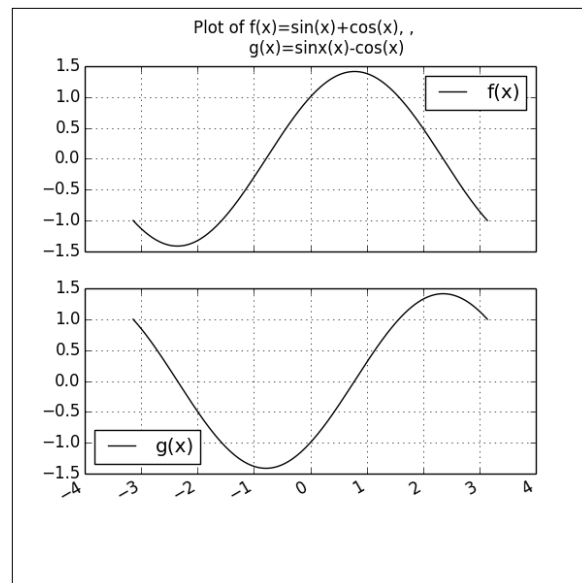
There is a lot more to using the plotting functionality of `matplotlib` within pandas. For more information, take a look at the documentation at `http://pandas.pydata.org/pandas-docs/dev/visualization.html`.

Summary

To summarize, we have discussed how to handle missing data values and manipulate dates and time series in pandas. We also took a brief detour to investigate the plotting functionality in pandas using `matplotlib`. Handling missing data plays a very important part in the preparation of clean data for analysis and prediction, and the ability to plot and visualize data is an indispensable part of every good data analyst's toolbox.

In the next chapter, we will do some elementary data analysis on a real-world dataset where we will analyze and answer basic questions about the data. For further references about these topics in pandas, please take a look at the official documentation at `http://pandas.pydata.org/pandas-docs/stable/index.html`.

7
A Tour of Statistics – The Classical Approach

In this chapter, we take a brief tour of classical statistics (also called the frequentist approach) and show how we can use pandas together with `stats` packages, such as `scipy.stats` and `statsmodels`, to conduct statistical analyses. This chapter and the following ones are not intended to be a primer on statistics, but they just serve as an illustration of using pandas along with the `stats` packages. In the next chapter, we will examine an alternative approach to the classical view — *Bayesian statistics*. The various topics that are discussed in this chapter are as follows:

- Descriptive statistics and inferential statistics
- Measures of central tendency and variability
- Statistical hypothesis testing
- Z-test
- T-test
- Analysis of variance
- Confidence intervals
- Correlation and linear regression

Descriptive statistics versus inferential statistics

In descriptive or summary statistics, we attempt to describe the features of a collection of data in a quantitative way. This is different from inferential or inductive statistics because its aim is to summarize a sample rather than use the data to infer or draw conclusions about the population from which the sample is drawn.

Measures of central tendency and variability

Some of the measures used in descriptive statistics include the measures of central tendency and measures of variability.

A measure of central tendency is a single value that attempts to describe a dataset by specifying a central position within the data. The three most common measures of central tendency are the **mean**, **median**, and **mode**.

A measure of variability is used to describe the variability in a dataset. Measures of variability include variance and standard deviation.

Measures of central tendency

Let's take a look at the measures of central tendency and an illustration in the following sections.

The mean

The mean or sample is the most popular measure of central tendency. It is equal to the sum of all values in the dataset divided by the number of values in the dataset. Thus, in a dataset of *n* values, the mean is calculated as follows:

$$\overline{x} = \frac{x_1 + x_2 + x_3 + \ldots + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

We use $\overline{x}$ if the data values are from a sample and μ if the data values are from a population.

The sample mean and population mean are different. The sample mean is what is known as an unbiased estimator of the true population mean. By repeated random sampling of the population to calculate the sample mean, we can obtain a mean of sample means. We can then invoke the law of large numbers and the **central limit theorem** (**CLT**) and denote the mean of sample means as an estimate of the true population mean.

The population mean is also referred to as the expected value of the population.

The mean, as a calculated value, is often not one of the values observed in the dataset. The main drawback of using the mean is that it is very susceptible to outlier values, or if the dataset is very skewed. For additional information, please refer to these links at `http://en.wikipedia.org/wiki/Sample_mean_and_sample_covariance`, `http://en.wikipedia.org/wiki/Law_of_large_numbers`, and `http://bit.ly/1bv7l4s`.

The median

The median is the data value that divides the set of sorted data values into two halves. It has exactly half of the population to its left and the other half to its right. In the case when the number of values in the dataset is even, the median is the average of the two middle values. It is less affected by outliers and skewed data.

The mode

The mode is the most frequently occurring value in the dataset. It is more commonly used for categorical data in order to know which category is most common. One downside to using the mode is that it is not unique. A distribution with two modes is described as bimodal, and one with many modes is denoted as multimodal. Here is an illustration of a bimodal distribution with modes at two and seven since they both occur four times in the dataset:

```
In [4]: import matplotlib.pyplot as plt
        %matplotlib inline
In [5]: plt.hist([7,0,1,2,3,7,1,2,3,4,2,7,6,5,2,1,6,8,9,7])
        plt.xlabel('x')
        plt.ylabel('Count')
```

```
plt.title('Bimodal distribution')
plt.show()
```
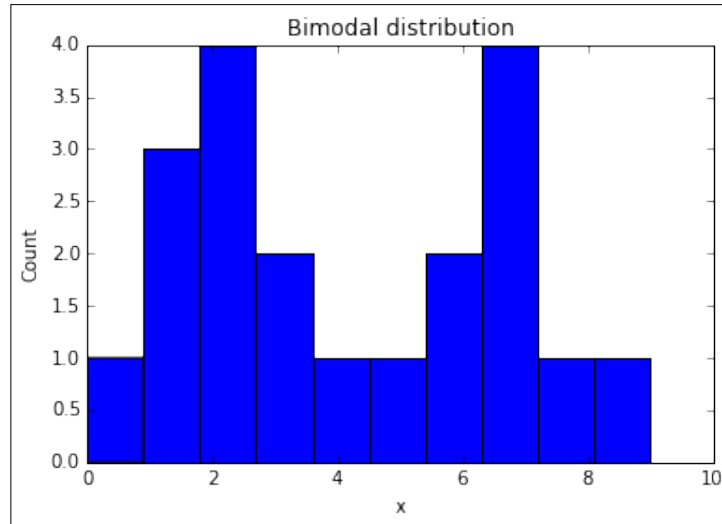

Computing measures of central tendency of a dataset in Python

To illustrate, let us consider the following dataset consisting of marks obtained by 15 pupils for a test scored out of 20:

```
In [18]: grades = [10, 10, 14, 18, 18, 5, 10, 8, 1, 12, 14, 12, 13, 1, 18]
```

The mean, median, and mode can be obtained as follows:

```
In [29]: %precision 3  # Set output precision to 3 decimal places
Out[29]:u'%.3f'


In [30]: import numpy as np
         np.mean(grades)
Out[30]: 10.933


In [35]: %precision
         np.median(grades)
```

```
Out[35]: 12.0


In [24]: from scipy import stats
         stats.mode(grades)
Out[24]: (array([ 10.]), array([ 3.]))
In [39]: import matplotlib.pyplot as plt
In [40]: plt.hist(grades)
         plt.title('Histogram of grades')
         plt.xlabel('Grade')
         plt.ylabel('Frequency')
         plt.show()
```
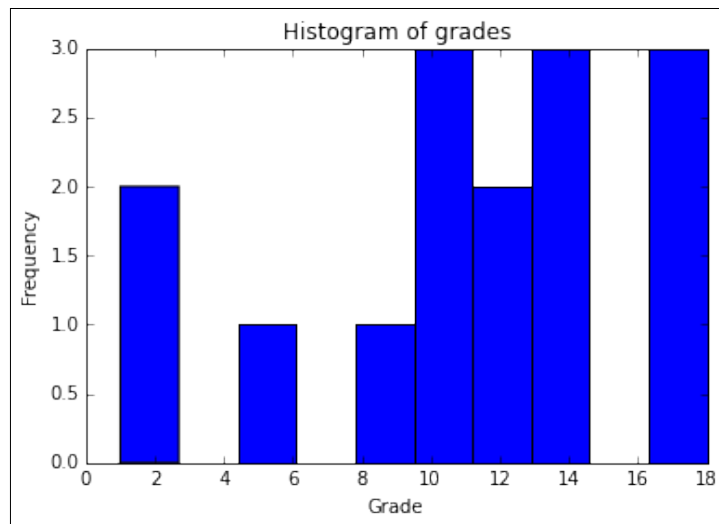

To illustrate how the skewness of data or an outlier value can drastically affect the usefulness of the mean as a measure of central tendency, consider the following dataset that shows the wages (in thousands of dollars) of the staff at a factory:

```
In [45]: %precision 2
         salaries = [17, 23, 14, 16, 19, 22, 15, 18, 18, 93, 95]


In [46]: np.mean(salaries)
Out[46]: 31.82
```

Based on the mean value, we may make the assumption that the data is centered around the mean value of 31.82. However, we would be wrong. To see this, let's display an empirical distribution of the data using a bar plot:

```
In [59]: fig = plt.figure()
         ax = fig.add_subplot(111)
         ind = np.arange(len(salaries))
         width = 0.2
         plt.hist(salaries, bins=xrange(min(salaries),
         max(salaries)).__len__())
         ax.set_xlabel('Salary')
         ax.set_ylabel('# of employees')
         ax.set_title('Bar chart of salaries')
         plt.show()
```
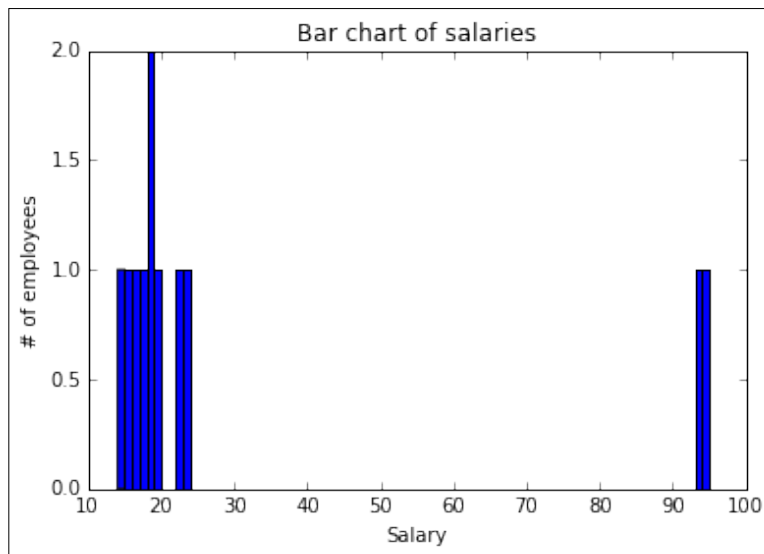

From the preceding bar plot, we can see that most of the salaries are far below 30K and no one is close to the mean of 32K. Now, if we take a look at the median, we see that it is better measure of central tendency in this case:

```
In [47]: np.median(salaries)
Out[47]: 18.00
```

We can also take a look at a histogram of the data:

```
In [56]: plt.hist(salaries, bins=len(salaries))
         plt.title('Histogram of salaries')
         plt.xlabel('Salary')
         plt.ylabel('Frequency')
         plt.show()
```
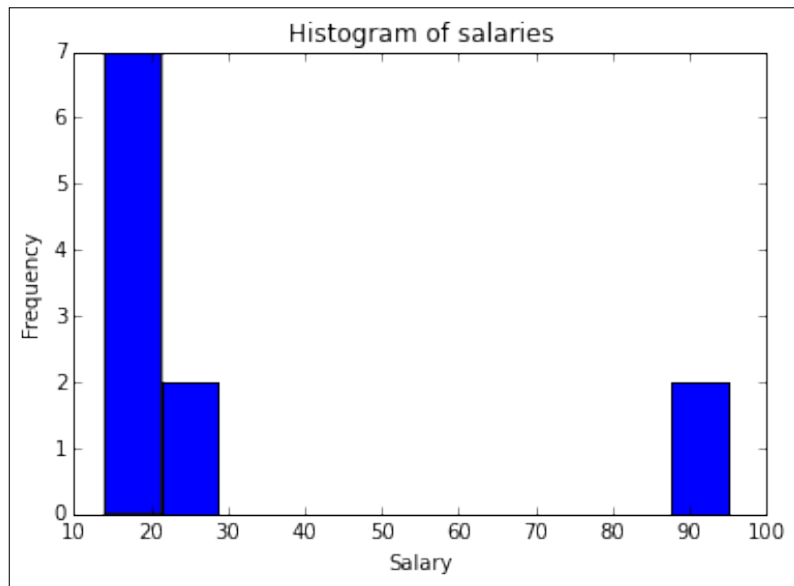

The histogram is actually a better representation of the data as bar plots are generally used to represent categorical data while histograms are preferred for quantitative data, which is the case for the salaries' data.

For more information on when to use histograms versus bar plots, refer to `http://onforb.es/1Dru2gv`.

If the distribution is symmetrical and unimodal (that is, has only one mode), the three measures—mean, median, and mode—will be equal. This is not the case if the distribution is skewed. In that case, the mean and median will differ from each other. With a negatively skewed distribution, the mean will be lower than the median and vice versa for a positively skewed distribution:

FIGURE 15.6 Examples of normal and skewed distributions
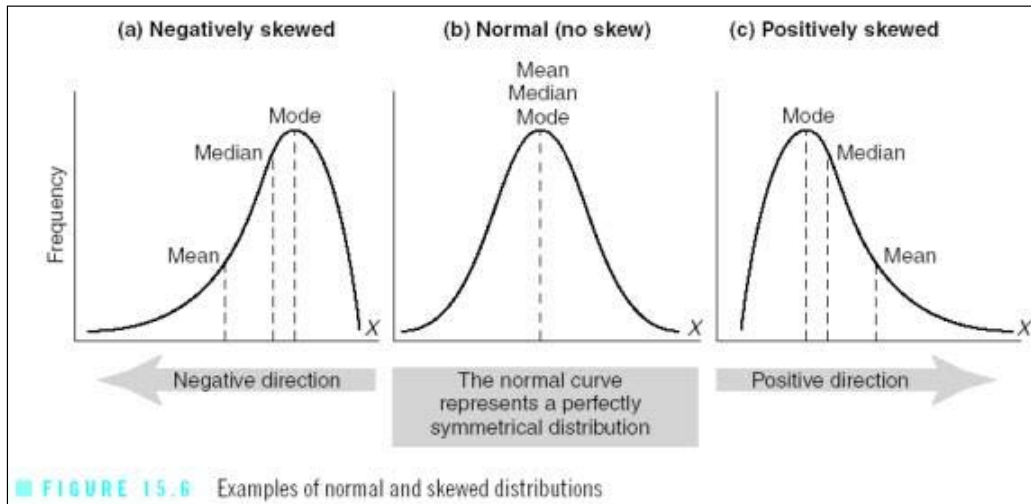
The preceding figure is sourced from `http://www.southalabama.edu/coe/bset/johnson/lectures/lec15_files/image014.jpg`.

Measures of variability, dispersion, or spread

Another characteristic of distribution that we measure in descriptive statistics is variability.

Variability specifies how much the data points are different from each other, or dispersed. Measures of variability are important because they provide an insight into the nature of the data that is not provided by the measures of central tendency.

As an example, suppose we conduct a study to examine how effective a pre-K education program is in lifting test scores of economically disadvantaged children. We can measure the effectiveness not only in terms of the average value of the test scores of the entire sample but also with the dispersion of the scores. Is it useful for some students and not so much for others? The variability of the data may help us identify some steps to be taken to improve the usefulness of the program.

Range

The simplest measure of dispersion is the range. The range is the difference between the lowest and highest scores in a dataset. This is the simplest measure of spread.

$$Range = highest\ value - lowest\ value$$

Quartile

A more significant measure of dispersion is the quartile and related interquartile ranges. It also stands for *quarterly percentile*, which means that it is the value on the measurement scale below which 25, 50, 75, and 100 percent of the scores in the sorted dataset fall. The quartiles are three points that split the dataset into four groups, with each one containing one-fourth of the data. To illustrate, suppose we have a dataset of 20 test scores where we rank them as follows:

```
In [27]: import random

         random.seed(100)

         testScores = [random.randint(0,100) for p in
                       xrange(0,20)]

         testScores
Out[27]: [14, 45, 77, 71, 73, 43, 80, 53, 8, 46, 4, 94, 95, 33, 31, 77,
20, 18, 19, 35]


In [28]: #data needs to be sorted for quartiles
         sortedScores = np.sort(testScores)
In [30]: rankedScores = {i+1: sortedScores[i] for i in
                       xrange(len(sortedScores))}


In [31]: rankedScores
Out[31]:
{1: 4,
 2: 8,
 3: 14,
 4: 18,
 5: 19,
 6: 20,
 7: 31,
```

```
8: 33,
 9: 35,
 10: 43,
 11: 45,
 12: 46,
 13: 53,
 14: 71,
 15: 73,
 16: 77,
 17: 77,
 18: 80,
 19: 94,
 20: 95}
```

The first quartile (Q1) lies between the fifth and sixth score, the second quartile (Q2) between the tenth and eleventh score, and the third quartile between the fifteenth and sixteenth score. Thus, we have (by using linear interpolation and calculating the midpoint):

```
Q1 = (19+20)/2 = 19.5
Q2 = (43 + 45)/2 = 44
Q3 = (73 + 77)/2 = 75
```

To see this in IPython, we can use the `scipy.stats` or `numpy.percentile` packages:

```
In [38]: from scipy.stats.mstats import mquantiles
         mquantiles(sortedScores)
Out[38]: array([ 19.45,  44.  ,  75.2 ])


In [40]: [np.percentile(sortedScores, perc) for perc in [25,50,75]]
Out[40]: [19.75, 44.0, 74.0]
```

The reason why the values don't match exactly with our previous calculations is due to the different interpolation methods. More information on the various types of methods to obtain quartile values can be found at `http://en.wikipedia.org/wiki/Quartile`. The interquartile range is the first quartile subtracted from the third quartile (**Q3 - Q1**), It represents the middle 50 in a dataset.

For more information, refer to `http://bit.ly/1cMMycN`.

For more details on the `scipy.stats` and `numpy.percentile` functions, see the documents at `http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mstats.mquantiles.html` and `http://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.percentile.html`.

Deviation and variance

A fundamental idea in the discussion of variability is the concept of deviation. Simply put, a deviation measure tells us how far away a given value is from the mean of the distribution, that is, $X - X'$.

To find the deviation of a set of values, we define the variance as the sum of squared deviations and normalize it by dividing it by the size of the dataset. This is referred to as the variance. We need to use the sum of squared deviations as taking the sum of deviations around the mean results in 0 since the negative and positive deviations cancel each other out. The sum of squared deviations is defined as follows:

$$SS = \sum_{i=1}^{N}(X - \overline{X})^2$$

It can be shown that the preceding expression is equivalent to:

$$SS = \sum_{I=1}^{N} X^2 - \frac{\left(\sum_{I=1}^{N} X\right)^2}{N}$$

Formally, the variance is defined as follows:

- For sample variance, use the following formula:

$$s^2 = \frac{SS}{N-1} = \frac{1}{N-1}\sum_{i=1}^{N}\left(X - \overline{X}\right)^2$$

- For population variance, use the following formula:

$$\sigma^2 = \frac{SS}{N} = \frac{1}{N}\sum_{i=1}^{N}\left(X - \mu\right)^2$$

The reason why the denominator is $N-1$ for the sample variance instead of N is that for sample variance, we wish to use an unbiased estimator. For more details on this, take a look at `http://en.wikipedia.org/wiki/Bias_of_an_estimator`.

The values of this measure are in squared units. This emphasizes the fact that what we have calculated as the variance is the squared deviation. Therefore, to obtain the deviation in the same units as the original points of the dataset, we must take the square root, and this gives us what we call the standard deviation. Thus, the standard deviation of a sample is given by using the following formula:

$$s = \sqrt{\frac{SS}{N-1}} = \sqrt{\frac{\sum(X-\bar{X})^2}{N-1}}$$

However, for a population, the standard deviation is given by the following formula:

$$\sigma = \sqrt{\frac{SS}{N}} = \sqrt{\frac{\sum(X-\mu)^2}{N}}$$

Hypothesis testing – the null and alternative hypotheses

In the preceding section, we had a brief discussion of what is referred to as descriptive statistics. In this section, we will discuss what is known as inferential statistics whereby we try to use characteristics of the sample dataset to draw conclusions about the wider population as a whole.

One of the most important methods in inferential statistics is hypothesis testing. In hypothesis testing, we try to determine whether a certain hypothesis or research question is true to a certain degree. One example of a hypothesis would be this: Eating spinach improves long-term memory.

In order to investigate this question by using hypothesis testing, we can select a group of people as subjects for our study and divide them into two groups or samples. The first group will be the experimental group, and it will eat spinach over a predefined period of time. The second group, which does not receive spinach, will be the control group. Over selected periods of times, the memory of individuals in the two groups will be measured and tallied.

Our goal at the end of our experiment would be to be able to make a statement such as "Eating spinach results in improvement in long-term memory, which is not due to chance". This is also known as significance.

In the preceding scenario, the collection of subjects in the study is referred to as the sample, and the general set of people about whom we would like to draw conclusions is the population.

The ultimate goal of our study would be to determine whether any effects that we observed in the sample can be generalized to the population as a whole. In order to carry out hypothesis testing, we will need to come up with what are known as the null and alternative hypotheses.

The null and alternative hypotheses

By referring to the preceding spinach example, the null hypothesis would be: Eating spinach has no effect on long-term memory performance.

The null hypothesis is just that—it nullifies what we're trying to *prove* by running our experiment. It does so by asserting that some statistical metric (to be explained later) is zero.

The alternative hypothesis is what we hope to support. It is the opposite of the null hypothesis and we assume it to be true until the data provides sufficient evidence that indicates otherwise. Thus, our alternative hypothesis in this case is: Eating spinach results in an improvement in long-term memory.

Symbolically, the null hypothesis is referred to as *H0* and the alternative hypothesis as *H1*. You may wish to restate the preceding null and alternative hypotheses as something more concrete and measurable for our study. For example, we could recast *H0* as follows:

The mean memory score for a sample of 1,000 subjects who ate 40 grams of spinach daily for a period of 90 days would not differ from the control group of 1,000 subjects who consumed no spinach within the same time period.

In conducting our experiment/study, we focus on trying to prove or disprove the null hypothesis. This is because we can calculate the probability that our results are due to chance. However, there is no easy way to calculate the probability of the alternative hypothesis since any improvement in long-term memory could be due to factors other than just eating spinach.

We test out the null hypothesis by assuming that it is true and calculate the probability getting of the results we do by chance alone. We set a threshold level—alpha *a*—for which we can reject the null hypothesis if the calculated probability is smaller or accept it if it is greater. Rejecting the null hypothesis is tantamount to accepting the alternative hypothesis and vice versa.

The alpha and p-values

In order to conduct an experiment to decide for or against our null hypothesis, we need to come up with an approach that will enable us to make the decision in a concrete and measurable way. To do this test of significance, we have to consider two numbers—the p-value of the test statistic and the threshold level of significance, which is also known as **alpha**.

The p-value is the probability if the result we observe by assuming that the null hypothesis is true or it occurred by occurred by chance alone.

The p-value can also be thought of as the probability of obtaining a test statistic as extreme as or more extreme than the actual obtained test statistic, given that the null hypothesis is true.

The alpha value is the threshold value against which we compare p-values. This gives us a cut-off point in order to accept or reject the null hypothesis. It is a measure of how extreme the results we observe must be in order to reject the null hypothesis of our experiment. The most commonly used values of alpha are 0.05 or 0.01.

In general, the rule is as follows:

If the p-value is less than or equal to alpha ($p < .05$), then we reject the null hypothesis and state that the result is statistically significant.

If the p-value is greater than alpha ($p > .05$), then we have failed to reject the null hypothesis, and we say that the result is not statistically significant.

The seemingly arbitrary values of alpha in usage are one of the shortcomings of the frequentist methodology, and there are many questions concerning this approach. The following article in the *Nature* journal highlights some of the problems: `http://www.nature.com/news/scientific-method-statistical-errors-1.14700`.

For more details on this topic, refer to:

- `http://statistics.about.com/od/Inferential-Statistics/a/What-Is-The-Difference-Between-Alpha-And-P-Values.htm`

- `http://bit.ly/1GzYX1P`

- `http://en.wikipedia.org/wiki/P-value`

Type I and Type II errors

There are two type of errors, as explained here:

- **Type I Error**: In this type of error, we reject *H0* when in fact *H0* is true. An example of this would be a jury convicting an innocent person for a crime that the person did not commit.

- **Type II Error**: In this type of error, we fail to reject *H0* when in fact *H1* is true. This is equivalent to a guilty person escaping conviction.

Statistical hypothesis tests

A statistical hypothesis test is a method to make a decision using data from a statistical study or experiment. In statistics, a result is termed statistically significant if it is unlikely to have occurred only by chance, based on a predetermined threshold probability or significance level. There are two classes of statistical tests: 1-tailed and 2-tailed tests.

In a 2-tailed test, we allot half of our alpha to test the statistical significance in one direction and the other half to test statistical significance in the other direction.

In a 1-tailed test, the test is performed in one direction only.

For more details on this topic, refer to `http://www.ats.ucla.edu/stat/mult_pkg/faq/general/tail_tests.htm`.

Background

To apply statistical inference, it is important to understand the concept of what is known as a sampling distribution. A sampling distribution is the set of all possible values of a statistic along with their probabilities, assuming we sample at random from a population where the null hypothesis holds true.

A more simplistic definition is this—a sampling distribution is the set of values the statistic can assume (distribution) if we were to repeatedly draw samples from the population along with their associated probabilities.

The value of a statistic is a random sample from the statistic's sampling distribution. The sampling distribution of the mean is calculated by obtaining many samples of various sizes and taking their mean. It has a mean, $\mu_{X'}$, equal to μ and a standard deviation, $\sigma_{X'}$ $\sigma_{\bar{x}}$, equal to $\frac{\sigma}{\sqrt{N}}$ $\sigma/\sqrt{N}$.

The CLT states that the sampling distribution is normally distributed if the original or raw-score population is normally distributed, or if the sample size is large enough. Conventionally, statisticians denote large-enough sample sizes as $N \geq 30$, that is, a sample size of 30 or more. This is still a topic of debate though.

For more details on this topic, refer to `http://stattrek.com/sampling/sampling-distribution.aspx` and `http://en.wikipedia.org/wiki/Central_limit_theorem`.

The standard deviation of the sampling distribution is often referred to as the standard error of the mean or just standard error.

The z-test

The z-test is appropriate under the following conditions:

- The study involves a single sample mean and the parameters—μ and σ—of the null hypothesis population are known
- The sampling distribution of the mean is normally distributed
- The size of the sample is $N \geq 30$

We use the z-test when the mean of the population is *known*. In the z-test, we ask the question whether the population mean, μ, is different from a hypothesized value. The null hypothesis in the case of the z-test is as follows:

$$H_0 : \mu = \mu_0$$

where, μ = population mean

μ_0 = hypothesized value

The alternative hypothesis, H_a, can be one of the following:

$$H_a : \mu < \mu_0$$

$$H_a : \mu > \mu_0$$

$$H_a : \mu \neq \mu_0$$

The first two are 1-tailed tests while the last one is a 2-tailed test. In concrete terms, to test H_0, we calculate the test statistic:

$$z = \frac{X - \mu_0}{\sigma_{\bar{X}}}$$

Here, $\sigma_{\bar{x}}$ is the true standard deviation of the sampling distribution of $\bar{X}$. If H_0 is true, the z-test statistics will have the standard normal distribution.

Here, we present a quick illustration of the z-test.

Suppose we have a fictional company Intelligenza, that claims that they have come up with a radical new method for improved memory retention and study. They claim that their technique can improve grades over traditional study techniques. Suppose the improvement in grades is 40 percent with a standard deviation of 10 percent by using traditional study techniques.

A random test was run on 100 students using the Intelligenza method, and this resulted in a mean improvement of 44 percent. Does Intelligenza's claim hold true?

The null hypothesis for this study states that there is no improvement in grades using Intelligenza's method over traditional study techniques. The alternative hypothesis is that there is an improvement by using Intelligenza's method over traditional study techniques.

The null hypothesis is given by the following:

$$H_0 : \mu = \mu_0$$

The alternative hypothesis is given by the following:

$$H_a : \mu > \mu_0$$

std error = 10/sqrt(100) = 1

z = (43.75-40)/(10/10) = 3.75 std errors

Recall that if the null hypothesis is true, the test statistic z will have a standard normal distribution that would look like this:

For reference, go to `http://mathisfun.com/data/images/normal-distrubution-large.gif`.
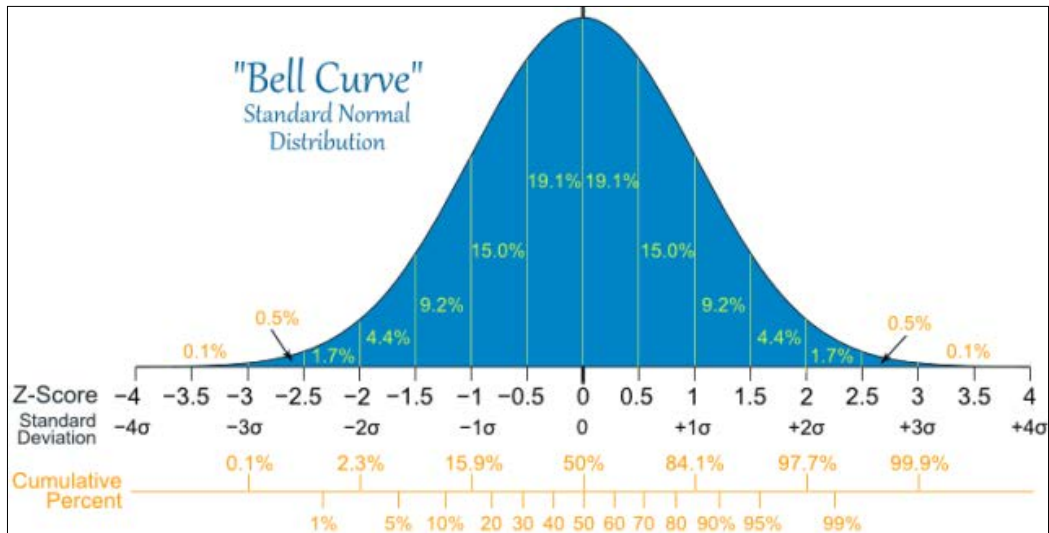
This value of *z* would be a random sample from the standard normal distribution, which is the distribution of *z* if the null hypothesis is true.

The observed value of z=43.75 corresponds to an extreme outlier p-value on the standard normal distribution curve, much less than 0.1 percent.

The p-value is the area under the curve, to the right of the value of 3.75 on the preceding normal distribution curve.

This suggests that it would be highly unlikely for us to obtain the observed value of the test statistic if we were sampling from a standard normal distribution.

We can look up the actual p-value using Python by using the `scipy.stats` package as follows:

```
In [104]: 1 - stats.norm.cdf(3.75)
Out[104]: 8.841728520081471e-05
```

Therefore, $P(z \geq 3.75) = 8.8e - 05$, that is, if the test statistic was normally distributed, then the probability to obtain the observed value is 8.8e-05, which is close to zero. So, it would be almost impossible to obtain the value that we observe if the null hypothesis was actually true.

In more formal terms, we would normally define a threshold or alpha value and reject the null hypothesis if the p-value ≤ α or fail to reject otherwise.

The typical values for α are 0.05 or 0.01. Following list explains the different values of alpha:

- **p-value <0.01**: There is VERY strong evidence against *H0*
- **0.01 < p-value < 0.05**: There is strong evidence against *H0*
- **0.05 < p-value < 0.1**: There is weak evidence against *H0*
- **p-value > 0.1**: There is little or no evidence against *H0*

Therefore, in this case, we would reject the null hypothesis and give credence to Intelligenza's claim and state that their claim is highly significant. The evidence against the null hypothesis in this case is significant. There are two methods that we use to determine whether to reject the null hypothesis:

- The p-value approach
- The rejection region approach

The approach that we used in the preceding example was the latter one.

The smaller the p-value, the less likely it is that the null hypothesis is true. In the rejection region approach, we have the following rule:

If $s_{\bar{X}} = s/\sqrt{N}$, reject the null hypothesis, else retain it.

The t-test

The z-test is most useful when the standard deviation of the population is known. However, in most real-world cases, this is an unknown quantity. For these cases, we turn to the t-test of significance.

For the t-test, given that the standard deviation of the population is unknown, we replace it by the standard deviation, *s*, of the sample. The standard error of the mean now becomes as follows:

$$s_{\bar{X}} = s/\sqrt{N}$$

The standard deviation of the sample s is calculated as follows:

$$s = \sqrt{\frac{\sum(X - X')^2}{N-1}}$$

The denominator is *N-1* and not *N*. This value is known as the number of degrees of freedom. I will now state without explanation that by the CLT the t-distribution approximates the normal, Guassian, or z-distribution as *N* and hence *N-1* increases, that is, with increasing **degrees of freedom** (**df**). When df = ∞, the t-distribution is identical to the normal or z-distribution. This is intuitive since as df increases, the sample size increases and s approaches σ , the true standard deviation of the population. There are an infinite number of t-distributions, each corresponding to a different value of df.

This can be seen in the following figure:
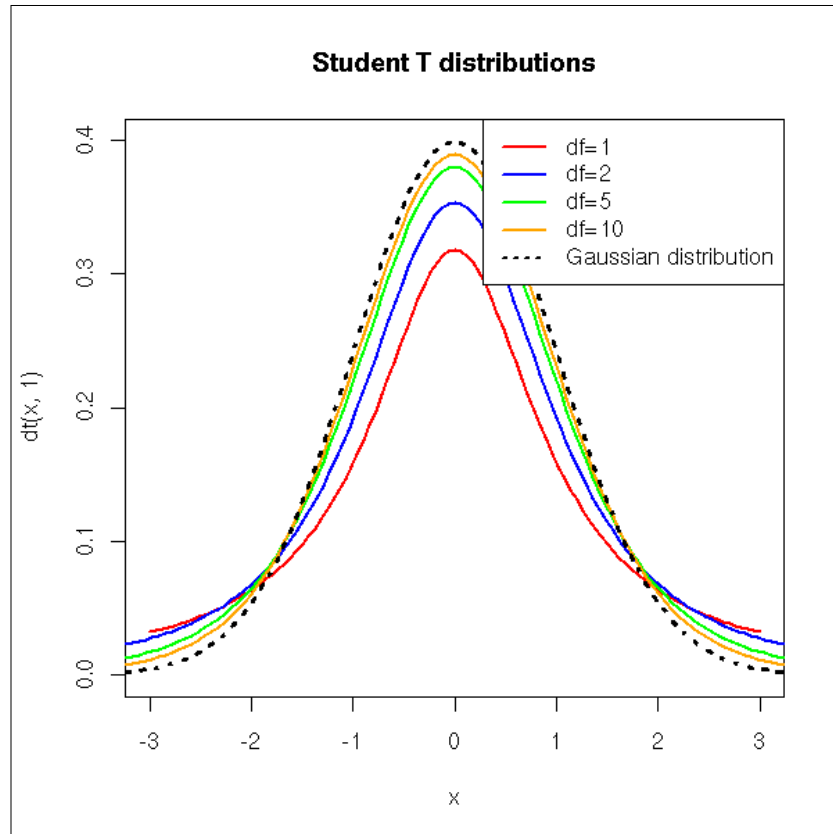
The reference of this image is from: `http://zoonek2.free.fr/UNIX/48_R/g593.png`.

A more detailed technical explanation on the relationship between t-distribution, z-distribution, and the degrees of freedom can be found at `http://en.wikipedia.org/wiki/Student's_t-distribution`.

Types of t-tests

There are various types of t-tests. Following are the most common ones; they typically formulate a null hypothesis that makes a claim about the mean of a distribution:

- **One sample independent t-test**: This is used to compare the mean of a sample with that of a known population mean or known value. Let's assume that we're health researchers in Australia who are concerned about the health of the aboriginal population and wish to ascertain whether babies born to low-income aboriginal mothers have lower birth weight than is normal.

  An example of a null hypothesis test for a one-sample t-test would be this: the mean birth weight for our sample of 150 deliveries of full-term, live baby deliveries from low-income aboriginal mothers is no different from the mean birth weight of babies in the general Australian population, that is, 3,367 grams.

  The reference of this information is: `http://bit.ly/1KY9T7f`.

- **Independent samples t-tests**: This is used to compare means from independent samples with each other. An example of an independent sample t-test would be a comparison of fuel economy of automatic transmission versus manual transmission vehicles. This is what our real-world example will focus on.

  The null hypothesis for the t-test would be this: there is no difference between the average fuel efficiency of cars with manual and automatic transmissions in terms of their average combined city/highway mileage.

- **Paired samples t-test**: In a paired/dependent samples t-test, we take each data point in one sample and pair it with a data point in the other sample in a meaningful way. One way to do this would be to measure against the same sample at different points in time. An example of this would be to examine the efficacy of a slimming diet by comparing the weight of a sample of participants before and after the diet.

  The null hypothesis in this case would be this: there is no difference between the mean weights of participants before and after going on the slimming diet, or, more succinctly, the mean difference between paired observations is zero.

  The reference for this information can be found at `http://en.wikiversity.org/wiki/T-test`.

A t-test example

In simplified terms, to do **Null Signifcance Hypothesis Testing** (**NHST**), we need to do the following:

1. Formulate our null hypothesis. The null hypothesis is our model of the system, assuming that the effect we wish to verify was actually due to chance.

2. Calculate our p-value.

3. Compare the calculated p-value with that of our alpha, or threshold value, and decide whether to reject or accept the null hypothesis. If the p-value is low enough (lower than alpha), we will draw the conclusion that the null hypothesis is likely to be untrue.

For our real-world illustration, we wish to investigate whether manual transmission vehicles are more fuel efficient than automatic transmission vehicles. In order to do this, we will make use of the Fuel Economy data published by the US government for 2014 at `http://www.fueleconomy.gov`.

```
In [53]: import pandas as pd

         import numpy as np

         feRawData = pd.read_csv('2014_FEGuide.csv')


In [54]: feRawData.columns[:20]

Out[54]: Index([u'Model Year', u'Mfr Name', u'Division', u'Carline',
u'Verify Mfr Cd', u'Index (Model Type Index)', u'Eng Displ', u'# Cyl',
u'Trans as listed in FE Guide (derived from col AA thru AF)', u'City FE
(Guide) - Conventional Fuel', u'Hwy FE (Guide) - Conventional Fuel',
u'Comb FE (Guide) - Conventional Fuel', u'City Unadj FE - Conventional
Fuel', u'Hwy Unadj FE - Conventional Fuel', u'Comb Unadj FE -
Conventional Fuel', u'City Unrd Adj FE - Conventional Fuel', u'Hwy Unrd
Adj FE - Conventional Fuel', u'Comb Unrd Adj FE - Conventional Fuel',
u'Guzzler? ', u'Air Aspir Method'], dtype='object')


In [51]: feRawData = feRawData.rename(columns={'Trans as listed in FE
Guide (derived from col AA thru AF)' :'TransmissionType',

                                               'Comb FE (Guide) -
Conventional Fuel' : 'CombinedFuelEcon'})


In [57]: transType=feRawData['TransmissionType']

         transType.head()
```

```
Out[57]: 0       Auto(AM7)
         1       Manual(M6)
         2       Auto(AM7)
         3       Manual(M6)
         4       Auto(AM-S7)
         Name: TransmissionType, dtype: object
```

Now, we wish to modify the preceding series so that the values just contain the `Auto` and `Manual` strings . We can do this as follows:

```
In [58]: transTypeSeries = transType.str.split('(').str.get(0)
         transTypeSeries.head()
Out[58]: 0       Auto
         1       Manual
         2       Auto
         3       Manual
         4       Auto
         Name: TransmissionType, dtype: object
```

We now create a final modified DataFrame from a Series that consists of the transmission type and the combined fuel economy figures:

```
In [61]: feData=pd.DataFrame([transTypeSeries,feRawData['CombinedFuelEcon
']]).T
         feData.head()
Out[61]:    TransmissionType    CombinedFuelEcon
         0  Auto                16
         1  Manual              15
         2  Auto                16
         3  Manual              15
         4  Auto                17
         5 rows × 2 columns
```

We can now separate the data for vehicles with automatic transmission from those with manual transmission as follows:

```
In [62]: feData_auto=feData[feData['TransmissionType']=='Auto']
         feData_manual=feData[feData['TransmissionType']=='Manual']
```

```
In [63]: feData_auto.head()
Out[63]:    TransmissionType      CombinedFuelEcon
        0   Auto                  16
        2   Auto                  16
        4   Auto                  17
        6   Auto                  16
        8   Auto                  17
        5 rows × 2 columns
```

This shows that there were 987 vehicles with automatic transmission versus 211 with manual transmission:

```
In [64]: len(feData_auto)
Out[64]: 987


In [65]: len(feData_manual)
Out[65]: 211


In [87]: np.mean(feData_auto['CombinedFuelEcon'])
Out[87]: 22.173252279635257


In [88]: np.mean(feData_manual['CombinedFuelEcon'])
Out[88]: 25.061611374407583


In [84]: import scipy.stats as stats
         stats.ttest_ind(feData_auto['CombinedFuelEcon'].tolist(),
                          feData_manual['CombinedFuelEcon'].tolist())
Out[84]: (array(-6.5520663209014325), 8.4124843426100211e-11)


In [86]: stats.ttest_ind(feData_auto['CombinedFuelEcon'].tolist(),
                          feData_manual['CombinedFuelEcon'].tolist(),
                          equal_var=False)
Out[86]: (array(-6.949372262516113), 1.9954143680382091e-11)
```

Confidence intervals

In this section, we will address the issue of confidence intervals. A confidence interval enables us to make a probabilistic estimate of the value of the mean of a population's given sample data.

This estimate, called an interval estimate, consists of a range of values (intervals) that act as good estimates of the unknown population parameter.

The confidence interval is bounded by confidence limits. A 95 percent confidence interval is defined as an interval in which the interval contains the population mean with 95 percent probability. So, how do we construct a confidence interval?

Suppose we have a 2-tailed t-test and we wish to construct a 95 percent confidence interval. In this case, we want the sample t-value, t_{samp} , corresponding to the mean to satisfy the following inequality:

$$-t_{0.025} \leq t_{samp} \leq t_{0.025}$$

Given that $t_{samp} = \dfrac{X_{samp} - \mu}{s_X}$, which we can substitute in the preceding inequality relation to obtain this:

$$X_{samp} - s_X t_{0.025} \leq \mu \leq X_{samp} + s_X t_{0.025}$$

The $\left[X_{samp} - s_X t_{0.025} \leq \mu \leq X_{samp} + s_X t_{0.025} \right]$ interval is our 95 percent confidence interval.

Generalizing, any confidence interval for any percentage y can be expressed as $\left[X_{samp} - s_X t_{cri} \leq \mu \leq X_{samp} + s_X t_{cri} \right]$, where t_{cri} is the t-tailed value of t, that is, $t_{y/2}$ corresponding to the desired confidence interval for y.

We will now take the opportunity to illustrate how we can calculate the confidence interval using a dataset from the popular statistical environment R. The `stats` models' module provides access to the datasets that are available in the core datasets package of R via the `get_rdataset` function.

An illustrative example

We will consider the dataset known as `faithful` that consists of data obtained by observing the eruptions of the Old Faithful geyser in the Yellowstone National Park in the U.S. The two variables in the dataset are eruptions, which are the length of time the geyser erupts and waiting which is the time interval until the next eruption. There were 272 observations.

```
In [46]: import statsmodels.api as sma
         faithful=sma.datasets.get_rdataset("faithful")
         faithful
Out[46]: <class 'statsmodels.datasets.utils.Dataset'>


In [48]: faithfulDf=faithful.data
         faithfulDf.head()
Out[48]:    eruptions   waiting
         0    3.600        79
         1    1.800        54
         2    3.333        74
         3    2.283        62
         4    4.533        85
5 rows × 2 columns


In [50]: len(faithfulDf)
Out[50]: 272
```

Let us calculate a 95 percent confidence interval for the mean waiting time of the geyser. To do this, we first obtain the sample mean and standard deviation of the data:

```
In [80]: mean,std=(np.mean(faithfulDf['waiting']),
                    np.std(faithfulDf['waiting']))
```

We now make use of the `scipy.stats` package to calculate the confidence interval:

```
In [81]: from scipy import stats
         N=len(faithfulDf['waiting'])

         ci=stats.norm.interval(0.95,loc=mean,scale=std/np.sqrt(N))
In [82]: ci
Out[82]: (69.28440107709261, 72.509716569966201)
```

Thus, we can state that with 95 percent confidence that the [69.28, 72.51] interval contains the actual mean waiting time of the geyser.

Reference for this information: `http://statsmodels.sourceforge.net/devel/datasets/index.html` and `http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.norm.html`.

Correlation and linear regression

One of the most common tasks in statistics in determining the relationship between two variables is whether there is dependence between them. Correlation is the general term we use in statistics for variables that express dependence with each other.

We can then use this relationship to try and predict the value of one set of variables from the other; this is termed as regression.

Correlation

The statistical dependence expressed in a correlation relationship does not imply a causal relationship between the two variables; the famous line on this is "Correlation does not imply Causation". Thus, correlation between two variables or datasets implies just a casual rather than a causal relationship or dependence. For example, there is a correlation between the amount of ice cream purchased on a given day and the weather.

For more information on correlation and dependency, refer to `http://en.wikipedia.org/wiki/Correlation_and_dependence`.

The correlation measure, known as correlation coefficient, is a number that captures the size and direction of the relationship between the two variables. It can vary from -1 to +1 in direction and 0 to 1 in magnitude. The direction of the relationship is expressed via the sign, with a + sign expressing positive correlation and a - sign negative correlation. The higher the magnitude, the greater the correlation with a one being termed as the perfect correlation.

The most popular and widely used correlation coefficient is the Pearson product-moment correlation coefficient, known as *r*. It measures the linear correlation or dependence between two x and y variables and takes values between *-1* and *+1*.

The sample correlation coefficient *r* is defined as follows:

$$r = \frac{\sum_{i=1}^{N}\left(X_i - \bar{X}\right)\left(Y_i - \bar{Y}\right)}{\sqrt{\sum_{i=1}^{N}\left(X_i - \bar{X}\right)^2 \sum_{i=1}^{N}\left(Y_i - \bar{Y}\right)^2}}$$

This can also be written as follows:

$$r = \frac{N\sum X_i Y_i - \sum X_i Y_i}{\sqrt{N\sum X_i^2 - \left(\sum X_i\right)^2}\sqrt{N\sum Y_i^2 - \left(\sum Y_i\right)^2}}$$

Here, we have omitted the summation limits.

Linear regression

As mentioned earlier, regression focuses on using the relationship between two variables for prediction. In order to make predictions using linear regression, the best-fitting straight line must be computed.

If all the points (values for the variables) lie on a straight line, then the relationship is deemed perfect. This rarely happens in practice and the points do not all fit neatly on a straight line. Then the relationship is imperfect. In some cases, a linear relationship only occurs among log-transformed variables. This is a log-log model. An example of such a relationship would be a power law distribution in physics where one variable varies as a power of another.

Thus, an expression such as $Y = a^x$ results in the $ln(Y) = x * ln(a)$ linear relationship.

For more information see: `http://en.wikipedia.org/wiki/Power_law`

To construct the *best-fit* line, the method of least squares is used. In this method, the best-fit line is the optimal line that is constructed between the points for which the sum of the squared distance from each point to the line is the minimum. This is deemed to be the best linear approximation of the relationship between the variables we are trying to model using the linear regression. The best-fit line in this case is called the Least Squares Regression Line.

More formally, the least squares regression line is the line that has the minimum possible value for the sum of squares of the vertical distance from the data points to the line. These vertical distances are also known as residuals.

Thus, by constructing the least-squares regression line, we're trying to minimize the following expression:

$$\sum_{i=1}^{N} \left(Y_i - Y \right)^2$$

An illustrative example

We will now illustrate all the preceding points with an example. Suppose we're doing a study in which we would like to illustrate the effect of temperature on how often crickets chirp. Data for this example is obtained from a book *The Song of Insects*, *George W Pierce* in 1948. George Pierce measured the frequency of chirps made by a ground cricket at various temperatures.

We wish to investigate the frequency of cricket chirps and the temperature as we suspect that there is a relationship. The data consists of 16 data points and we read it into a data frame:

```
In [38]: import pandas as pd
         import numpy as np
         chirpDf= pd.read_csv('cricket_chirp_temperature.csv')
In [39]: chirpDf
Out[39]:chirpFrequency   temperature
0        20.000000       88.599998
1        16.000000       71.599998
2        19.799999       93.300003
3        18.400000       84.300003
4        17.100000       80.599998
5        15.500000       75.199997
6        14.700000       69.699997
7        17.100000       82.000000
```

```
8        15.400000        69.400002
9        16.200001        83.300003
10       15.000000        79.599998
11       17.200001        82.599998
12       16.000000        80.599998
13       17.000000        83.500000
14       14.400000        76.300003
15 rows × 2 columns
```

As a start, let us do a scatter plot of the data along with a regression line or line of best fit:

```
In [29]: plt.scatter(chirpDf.temperature,chirpDf.chirpFrequency,
            marker='o',edgecolor='b',facecolor='none',alpha=0.5)

        plt.xlabel('Temperature')

        plt.ylabel('Chirp Frequency')

        slope, intercept = np.polyfit(chirpDf.temperature,chirpDf.
chirpFrequency,1)

        plt.plot(chirpDf.temperature,chirpDf.temperature*slope +
intercept,'r')

        plt.show()
```
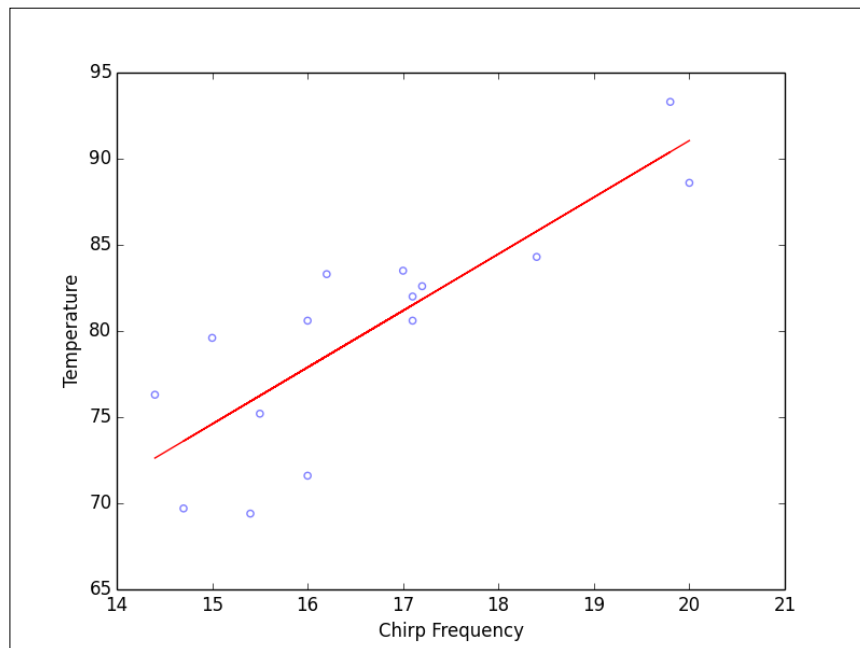
From the plot, we can see that there seems to be a linear relationship between temperature and the chirp frequency. We can now proceed to investigate further by using the `statsmodels.ols` (ordinary least squares) method to:

```
[37]: chirpDf= pd.read_csv('cricket_chirp_temperature.csv')

      chirpDf=np.round(chirpDf,2)

      result=sm.ols('temperature ~ chirpFrequency',chirpDf).fit()

      result.summary()
```

```
Out[37]: OLS Regression Results
   Dep. Variable: temperature      R-squared:        0.697
   Model: OLS      Adj. R-squared: 0.674
   Method:          Least Squares   F-statistic:     29.97
   Date:  Wed, 27 Aug 2014     Prob (F-statistic):     0.000107
   Time:  23:28:14          Log-Likelihood: -40.348
   No. Observations:        15      AIC:      84.70
   Df Residuals: 13        BIC:      86.11
   Df Model:        1
                    coef      std err t     P>|t| [95.0% Conf. Int.]
   Intercept      25.2323 10.060   2.508 0.026 3.499 46.966
   chirpFrequency 3.2911  0.601   5.475 0.000 1.992 4.590


   Omnibus:          1.003    Durbin-Watson:   1.818
   Prob(Omnibus):  0.606    Jarque-Bera (JB):       0.874
   Skew:    -0.391  Prob(JB):        0.646
   Kurtosis:        2.114   Cond. No.        171.
```

We will ignore most of the preceding results, except for the `R-squared`, `Intercept`, and `chirpFrequency` values.

From the preceding result, we can conclude that the slope of the regression line is 3.29, and the intercept on the temperature axis is 25.23. Thus, the regression line equation looks like this: *temperature = 25.23 + 3.29 * chirpFrequency.*

This means that as the chirp frequency increases by one, the temperature increases by about 3.29 degrees Fahrenheit. However, note that the intercept value is not really meaningful as it is outside the bounds of the data. We can also only make predictions for values within the bounds of the data. For example, we cannot predict what the chirpFrequency is at 32 degrees Fahrenheit as it is outside the bounds of the data; moreover, at 32 degrees Fahrenheit, the crickets would have frozen to death. The value of *R*, the correlation coefficient, is given as follows:

```
In [38]: R=np.sqrt(result.rsquared)
         R
Out[38]: 0.83514378678237422
```

Thus, our correlation coefficient is *R = 0.835*. This would indicate that about 84 percent of the chirp frequency can be explained by the changes in temperature.

Reference of this information: *The Song of Insects* `http://www.hup.harvard.edu/catalog.php?isbn=9780674420663`

The data is sourced from `http://bit.ly/1MrlJqR`.

For a more in-depth treatment of single and multi-variable regression, refer to the following websites:

- Regression (Part I): `http://bit.ly/1Eq5kSx`
- Regression (Part II): `http://bit.ly/1OmuFTV`

Summary

In this chapter, we took a brief tour of the classical or frequentist approach to statistics and showed you how to combine pandas along with the `stats` packages—`scipy.stats` and `statsmodels`—to calculate, interpret, and make inferences from statistical data.

In the next chapter, we will examine an alternative approach to statistics, which is the Bayesian approach. For deeper look at the statistics topics that we touched on, please take a look at *Understanding Statistics in the Behavioral Sciences*, which can be found at `http://www.amazon.com/Understanding-Statistics-Behavioral-Sciences-Robert/dp/0495596523`.

8

A Brief Tour of Bayesian Statistics

In this chapter, we will take a brief tour of an alternative approach to statistical inference called **Bayesian statistics**. It is not intended to be a full primer but just serve as an introduction to the Bayesian approach. We will also explore the associated Python-related libraries, how to use pandas, and `matplotlib` to help with the data analysis. The various topics that will be discussed are as follows:

- Introduction to Bayesian statistics
- Mathematical framework for Bayesian statistics
- Probability distributions
- Bayesian versus Frequentist statistics
- Introduction to PyMC and Monte Carlo simulation
- Illustration of Bayesian inference – Switchpoint detection

Introduction to Bayesian statistics

The field of Bayesian statistics is built on the work of Reverend Thomas Bayes, an 18th century statistician, philosopher, and Presbyterian minister. His famous Bayes' theorem, which forms the theoretical underpinnings for Bayesian statistics, was published posthumously in 1763 as a solution to the problem of *inverse probability*. For more details on this topic, refer to `http://en.wikipedia.org/wiki/Thomas_Bayes`.

Inverse probability problems were all the rage in the early 18th century and were often formulated as follows:

Suppose you play a game with a friend. There are 10 green balls and 7 red balls in bag 1 and 4 green and 7 red balls in bag 2. Your friend turns away from your view, tosses a coin and picks a ball from one of the bags at random, and shows it to you. The ball is red. What is the probability that the ball was drawn from bag 1?

These problems are termed inverse probability problems because we are trying to estimate the probability of an event that has already occurred (which bag the ball was drawn from) in light of the subsequent event (that the ball is red).
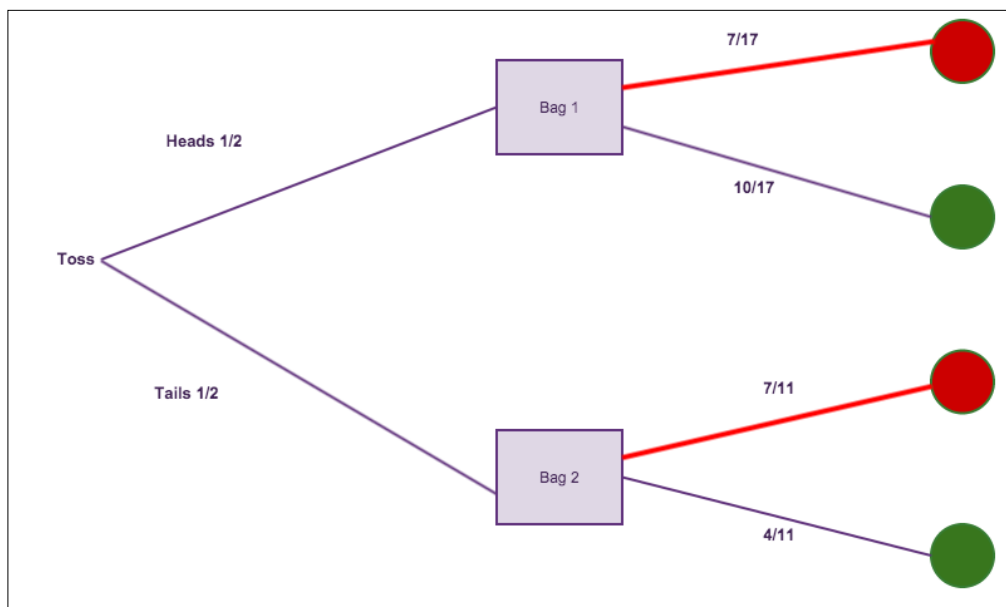
Bayesian_balls_illustration

Let us quickly illustrate how one would go about solving the inverse probability problem illustrated earlier. We wish to calculate the probability that the ball was drawn from bag 1, given that it is red. This can be denoted as $P(Bag\,1\,|\,Red\,Ball)$.

Let us start by calculating the probability of selecting a red ball. This can be calculated by following the two paths in red as shown in the preceding figure. Hence, we have $P(Red\,Ball) = \frac{1}{2} \times \frac{7}{17} + \frac{1}{2} \times \frac{7}{11} = 0.524$.

Now, the probability of choosing a red ball from bag 1 is via the upper path only and is given as follows:

$$P\left(Red\ Ball\ from\ Bag\,1\right)=\frac{1}{2}\times\frac{7}{17}=\frac{7}{34}=0.206$$

And, the probability of choosing a red ball from bag 2 is given as follows:

$$P\left(Red\ Ball\ from\ Bag\,2\right)=\frac{1}{2}\times\frac{7}{17}=\frac{7}{22}=0.318$$

Note that this probability can be written as follows:

$$P\left(Red\ Ball,Bag\,1\right)=P\left(Red\ Ball\,|\,Bag\,1\right)*P\left(Bag\,1\right)$$

By inspection we can see that $P\left(Bag\,1\right)=1/2$, and the final branch of the tree is only traversed if the ball is firstly in bag 1 and is a red ball. Hence, intuitively we'll get the following outcome:

$$P\left(Bag\,1\,|\,Red\ Ball\right)=\frac{P\left(Red\ Ball,Bag\,1\right)}{P\left(Red\ Ball\right)}$$
$$=\frac{P\left(Red\ Ball\,|\,Bag\,1\right)*P\left(Bag\,1\right)}{P\left(Red\ Ball\right)}$$
$$=\frac{0.206}{0.524}=0.393$$

Mathematical framework for Bayesian statistics

With Bayesian methods we present an alternative method of making a statistical inference. We first introduce the Bayes theorem, the fundamental equation from which all Bayesian inference is derived.

A couple of definitions about probability are in order:

- A, B: These are events that can occur with a certain probability.

- $P(A)$ and $P(B)$: This is the probability of the occurrence of a particular event.

- $P(A|B)$: This is the probability of A happening, given that B has occurred. This is known as a conditional probability.

- $P(AB) = P(A \text{ and } B)$: This is the probability of A and B occurring together.

We begin with the basic assumption, as follows:

$$P(AB) = P(B) * P(A|B)$$

The preceding equation relates the joint probability of P(AB) to the conditional probability P(A | B) and what is also known as the marginal probability P(B). If we rewrite the equation, we have the expression for conditional probability as follows:

$$P(A|B) = P(AB)/P(B)$$

This is somewhat intuitive—that the probability of A given B is obtained by dividing the probability of both A and B occurring by the probability that B occurred. The idea is that B is given, so we divide by its probability. A more rigorous treatment of this equation can be found at `http://bit.ly/1bCYXRd`, which is titled *Probability: Joint, Marginal and Conditional Probabilities*.

Similarly, by symmetry we have $P(AB) = P(BA) = P(A) * P(B|A)$. Thus, we have $P(A) * P(B|A) = P(B) * P(A|B)$. By dividing the expression by $P(B)$ on both sides and assuming P(B) !=0, we obtain this:

$$P(A|B) = P(A) * \frac{P(B|A)}{P(B)}$$

The preceding equation is referred to as Bayes theorem, the bedrock for all of Bayesian statistical inference. In order to link Bayes theorem to inferential statistics, we will recast the equation into what is called the *diachronic interpretation*, as follows:

$$P(H|D) = P(H) * \frac{P(D|H)}{P(D)}$$

where, H represents a hypothesis.

D represents an event that has already occurred, which we use in our statistical study, and is also referred to as *data*.

Then, (H) is the probability of our hypothesis *before* we observe the data. This is known as the prior probability. The use of prior probability is often touted as an advantage by Bayesian statisticians since prior knowledge or previous results can be used as input for the current model, resulting in increased accuracy. For more information on this, refer to `http://www.bayesian-inference.com/advantagesbayesian`.

$P(D)$ is the probability of obtaining the data that we observe regardless of the hypothesis. This is called the normalizing constant. The normalizing constant doesn't always need to be calculated, especially in many popular algorithms such as MCMC, which we will examine later in this chapter.

$P(H|D)$ is the probability that the hypothesis is true, given the data that we observe. This is called the posterior.

$P(D|H)$ is the probability of obtaining the data, considering our hypothesis. This is called the likelihood.

Thus, Bayesian statistics amounts to applying Bayes rule to solve problems in inferential statistics with H representing our hypothesis and D the data.

A Bayesian statistical model is cast in terms of parameters, and the uncertainty in these parameters is represented by probability distributions. This is different from the Frequentist approach where the values are regarded as deterministic. An alternative representation is as follows:

$$P(\theta|x)$$

where, θ is our unknown data and x is our observed data

In Bayesian statistics, we make assumptions about the prior data and use the likelihood to update to the posterior probability using the Bayes rule. As an illustration, let us consider the following problem. Here is a classic case of what is commonly known as the *urn problem*:

- Two urns contain colored balls
- Urn one contains 50 red and 50 blue balls

- Urn two contains 30 red and 70 blue balls
- One of the two urns is randomly chosen (50 percent probability) and then a ball is drawn at random from one of the two urns

If a red ball is drawn, what is the probability that it came from urn one? We want $P(H|D)$ that is $P(ball\,came\,from\,Urn1|Red\,ball\,is\,drawn)$.

Here, H denotes that the ball is drawn from Urn one, and D denotes that the drawn ball is red:

$$P(H) = P(ball\,is\,drawn\,from\,Urn1) = 0.5$$

We know that $P(H|D) = P(H)*P(D|H)/P(D)$, $P(D|H) = 0.5$, $P(D) = (50+30)/(100+100) = 0.4$, or
$$P(D) = P(H)P(D|H) + P(\sim H)P(D|\sim H) = 0.5*0.5 + 0.5*0.3 = 0.25 + 0.15$$
$$= 0.4$$

Hence, we conclude that $P(H|D) = 0.5*0.5/0.4 = 0.25/0.4 = 0.625$.

Bayes theory and odds

Bayes theorem can sometimes be represented by a more natural and convenient form by using an alternative formulation of probability called *odds*. Odds are generally expressed in terms of ratios and are used heavily. A 3 to 1 odds (written often as 3:1) of a horse winning a race represents the fact that the horse is expected to win with 75 percent probability.

Given a probability p, the odds can be computed as odds = $p:(1-p)$, which in the case of p=0.75 becomes 0.75:0.25, which is 3:1.

We can rewrite the form of Bayes theorem by using odds as:

$$o(A|D) = o(A)P(D|A)/P(D|B)$$

Applications of Bayesian statistics

Bayesian statistics can be applied to many problems that we encounter in classical statistics such as:

- Parameter estimation
- Prediction

- Hypothesis Testing
- Linear regression

There are many compelling reasons for studying Bayesian statistics; some of them being the use of prior information to better inform the current model. The Bayesian approach works with probability distributions rather than point estimates, thus producing more realistic predictions. Bayesian inference bases a hypothesis on the available data—*P(hypothesis|data)*. The Frequentist approach tries to fit the data based on a hypothesis. It can be argued that the Bayesian approach is the more logical and empirical one as it tries to base its belief on the facts rather than the other way round. For more information on this, refer to `http://www.bayesian-inference.com/advantagesbayesian`.

Probability distributions

In this section, we will briefly examine the properties of various probability distributions. Many of these distributions are used for Bayesian analysis; thus, a brief synopsis is needed. We will also illustrate how to generate and display these distributions using `matplotlib`. In order to avoid repeating import statements for every code snippet in each section, I will present the following standard set of Python code imports that need to be run before any of the code snippets mentioned in the following command. You only need to run these imports once per session. The imports are as follows:

```
In [1]: import pandas as pd
        import numpy as np
        from matplotlib import pyplot as plt
        from matplotlib import colors
        import matplotlib.pyplot as plt
        import matplotlib
        %matplotlib inline
```

Fitting a distribution

One of the steps that we have to take in a Bayesian analysis is to fit our data to a probability distribution. Selecting the correct distribution can be somewhat of an art and often requires statistical knowledge and experience, but we can follow a few guidelines to help us along the way; these are as follows:

- Determine whether the data is discrete or continuous
- Examine the skewness/symmetry of the data and if skewed, determine the direction

- Determine the lower and upper limits, if any
- Determine the likelihood of observing extreme values in the distribution

A statistical trial is a repeatable experiment with a set of well-defined outcomes that are known as the sample space. A Bernoulli trial is a Yes/No experiment where the random X variable is assigned the value of 1 in the case of a Yes and 0 in the case of a No. The event of tossing a coin and seeing whether it turns up heads is an example of a Bernoulli trial.

There are two classes of probability distributions: discrete and continuous. In the following sections, we will discuss the differences between these two classes of distributions and take a tour of the main distributions.

Discrete probability distributions

In this scenario, the variable can take only certain distinct values such as integers. An example of a discrete random variable is the number of heads obtained when we flip a coin 5 times; the possible values are {0,1,2,3,4,5}. We cannot obtain 3.82 heads for example. The range of values the random variable can take is specified by what is known as a **probability mass function** (**pmf**).

Discrete uniform distributions

The discrete uniform distribution is a distribution that models an event with a finite set of possible outcomes where each outcome is equally likely to be observed. For n outcomes, each has a probability of occurrence of $1/n$.

An example of this is throwing a fair die. The probability of any of the six outcomes is $1/6$. The PMF is given by $1/n$, and the expected value and variance are given by $(\max + \min)/2$ and $(n^2 - 1)/12$ respectively.

```
In [13]: from matplotlib import pyplot as plt

    import matplotlib.pyplot as plt

  X=range(0,11)

  Y=[1/6.0 if x in range(1,7) else 0.0 for x in X]

  plt.plot(X,Y,'go-', linewidth=0, drawstyle='steps-pre',

          label="p(x)=1/6")

  plt.legend(loc="upper left")

  plt.vlines(range(1,7),0,max(Y), linestyle='-')

  plt.xlabel('x')

  plt.ylabel('p(x)')
```

```
plt.ylim(0,0.5)
plt.xlim(0,10)
plt.title('Discrete uniform probability distribution with
          p=1/6')
plt.show()
```
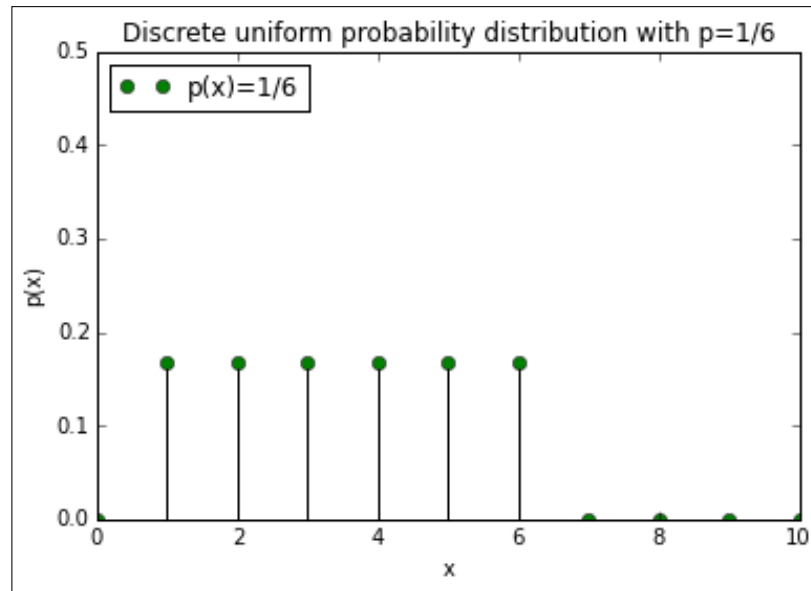

discrete uniform distribution

The Bernoulli distribution

The Bernoulli distribution measures the probability of success in a trial; for example, the probability that a coin toss turns up a head or a tail. This can be represented by a random X variable that takes a value of 1 if the coin turns up as heads and 0 if it is tails. The probability of turning up heads or tails is denoted by p and *q=1-p* respectively.

This can be represented by the following pmf:

$$f(k) = \begin{cases} 1-p, & k=0 \\ p, & k=1 \end{cases}$$

The expected value and variance are given by the following formula:

$$E(X) = p$$
$$Var(X) = p(1-p)$$

The reference for this information is at `http://en.wikipedia.org/wiki/Bernoulli_distribution`.

We now plot the Bernoulli distribution using `matplotlib` and `scipy.stats` as follows:

```
In [20]:import matplotlib
       from scipy.stats import bernoulli
     a = np.arange(2)

  colors = matplotlib.rcParams['axes.color_cycle']
  plt.figure(figsize=(12,8))
  for i, p in enumerate([0.0, 0.2, 0.5, 0.75, 1.0]):
          ax = plt.subplot(1, 5, i+1)
         plt.bar(a, bernoulli.pmf(a, p), label=p, color=colors[i],
alpha=0.5)
         ax.xaxis.set_ticks(a)

     plt.legend(loc=0)
        if i == 0:
            plt.ylabel("PDF at $k$")


     plt.suptitle("Bernoulli probability for various values of $p$")
Out[20]:
```

Bernoulli probability for various values of p
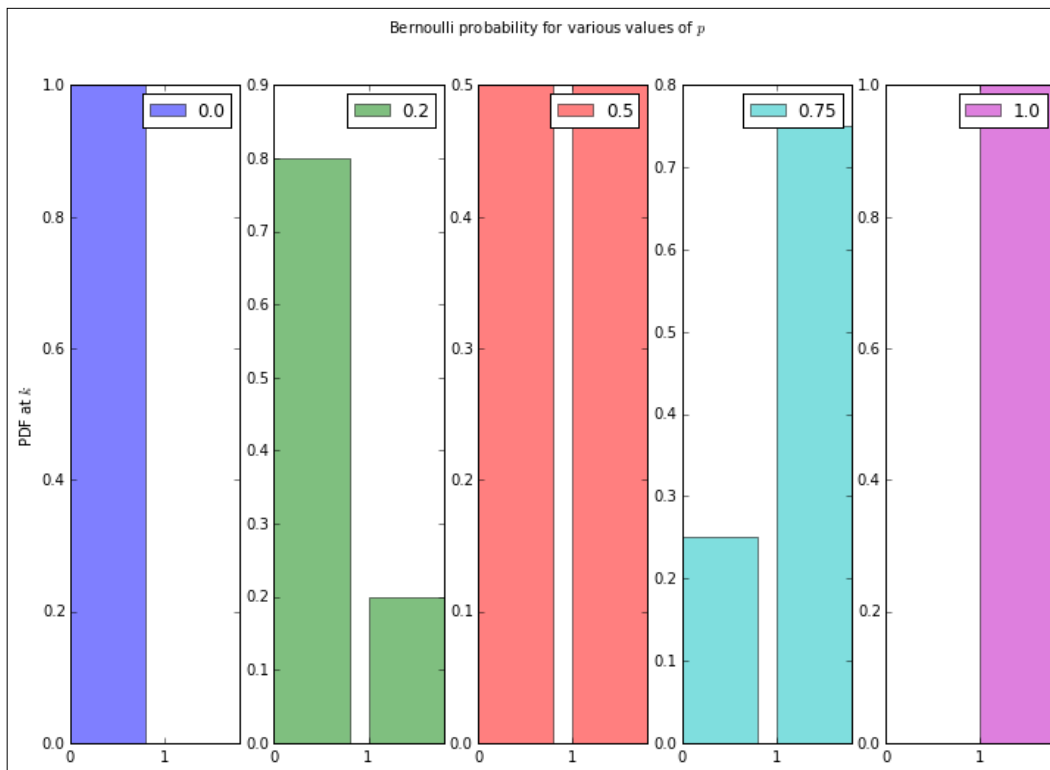
The binomial distribution

The binomial distribution is used to represent the number of successes in n independent Bernoulli trials that is, $Y = X_1 + X_2 + \cdots + X_n$.

Using the coin toss example, this distribution models the chance of getting X heads over n trials. For 100 tosses, the binomial distribution models the likelihood of getting 0 heads (extremely unlikely) to 50 heads (highest likelihood) to 100 heads (also extremely unlikely). This ends up making the binomial distribution symmetrical when the odds are perfectly even and skewed when the odds are far less even. The pmf is given by the following expression:

$$f(k) = \binom{n}{k} p^k q^{n-k}, 0 \leq k \leq n$$

The expectation and variance are given respectively by the following expression:

$$E(X) = np$$
$$Var(X) = np(1-p)$$

```
In [5]: from scipy.stats import binom
   clrs = ['blue','green','red','cyan','magenta']     plt.
figure(figsize=(12,6))
  k = np.arange(0, 22)
  for p, color in zip([0.001, 0.1, 0.3, 0.6, 0.999], clrs):
       rv = binom(20, p)
       plt.plot(k, rv.pmf(k), lw=2, color=color, label="$p$=" +
str(round(p,1)))

         plt.legend()
  plt.title("Binomial distribution PMF")
  plt.tight_layout()
  plt.ylabel("PDF at $k$")
  plt.xlabel("$k$")
Out[5]:
```
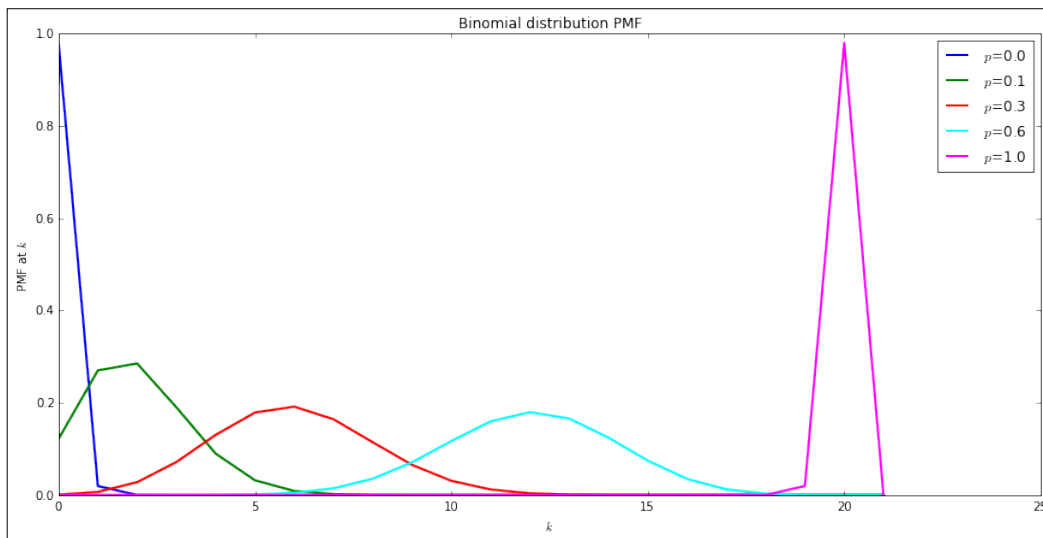

binomial distribution

The Poisson distribution

The Poisson distribution models the probability of a number of events within a given time interval, assuming that these events occur with a known average rate and successive events occur independent of the time since the previous event.

A concrete example of a process that can be modeled by a Poisson distribution would be if an individual received an average of, say, 23 e-mails per day. If we assume that the arrival times for the e-mails are independent of each other, the total number of e-mails an individual receives each day can be modeled by a Poisson distribution.

Another example could be the number of trains that stop at a particular station each hour. The pmf for a Poisson distribution is given by the following expression:

$$f(k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

Where λ is the rate parameter that represents the expected number of events/arrivals that occur per unit time, and k is the random variable that represents the number of events/arrivals.

The expectation and variance are given respectively by the following formula:

$$E(X) = \lambda$$

$$Var(X) = \lambda$$

For more information, refer to `http://en.wikipedia.org/wiki/Poisson_process`.

The pmf is plotted using `matplotlib` for various values as follows:

```
In [11]: %matplotlib inline
    import numpy as np
  import matplotlib
  import matplotlib.pyplot as plt
  from scipy.stats import poisson
  colors = matplotlib.rcParams['axes.color_cycle']
  k=np.arange(15)
  plt.figure(figsize=(12,8))
  for i, lambda_  in enumerate([1,2,4,6]):
```

```
        plt.plot(k, poisson.pmf(k, lambda_), '-o',
        label="$\lambda$=" + str(lambda_), color=colors[i])
        plt.legend()
    plt.title("Possion distribution PMF for various $\lambda$")
    plt.ylabel("PMF at $k$")
    plt.xlabel("$k$")
    plt.show()
Out[11]:
```
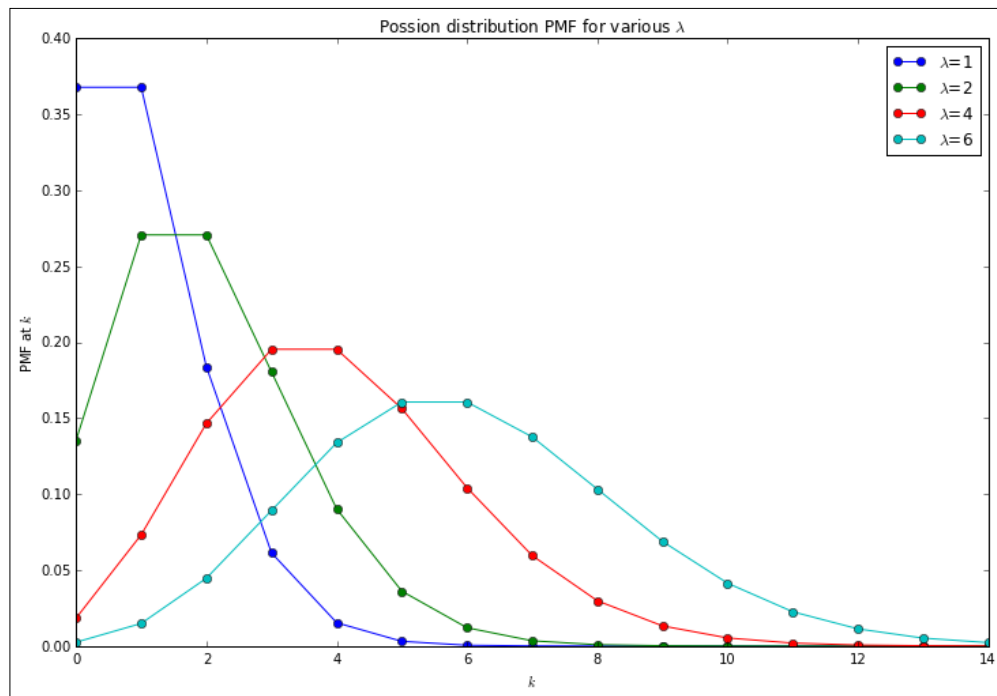

Poisson distribution

The Geometric distribution

For independent Bernoulli trials, the Geometric distribution measures the number of trials X needed to get one success. It can also represent the number of failures, $Y = X - 1$, before the first success.

The pmf is given by the following expression:

$$f(k) = p(1-p)^{k-1}$$

The preceding expression makes sense since $f(k) = P(X = k)$, and if it takes k trials to get one success (p), this means that we must have had $k-1$ failures which are equal to $(1-p)$.

The expectation and variance are given as follows:

$$E(X) = 1/p$$
$$Var(X) = (1-p)/p^2$$

The following command explains the preceding formula clearly:

```
In [12]: from scipy.stats import geom
    p_vals=[0.01,0.2,0.5,0.8,0.9]
  x = np.arange(geom.ppf(0.01,p),geom.ppf(0.99,p))
      colors = matplotlib.rcParams['axes.color_cycle']
      for p,color in zip(p_vals,colors):
        x = np.arange(geom.ppf(0.01,p),geom.ppf(0.99,p))
          plt.plot(x,geom.pmf(x,p),'-o',ms=8,label='$p$=' + str(p))
        plt.legend(loc='best')
      plt.ylim(-0.5,1.5)
      plt.xlim(0,7.5)
      plt.ylabel("Pmf at $k$")
      plt.xlabel("$k$")
      plt.title("Geometric distribution PMF")
Out[12]:
```
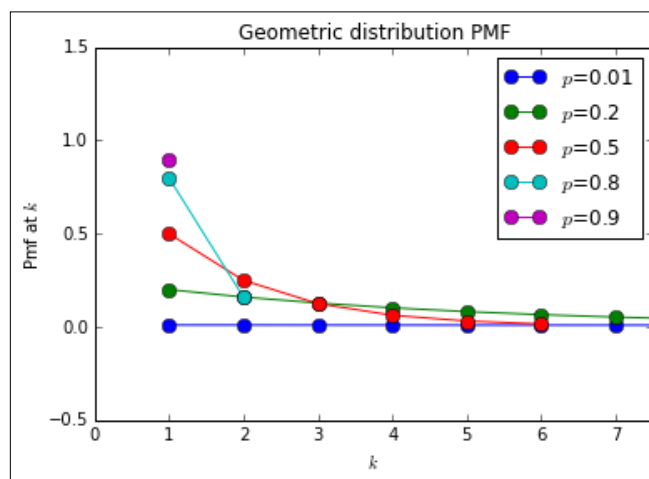

Geometric distribution

The negative binomial distribution

Also for independent Bernoulli trials, the negative binomial distribution measures the number of tries, $X = k$, before a specified number of successes, r, occur. An example would be the number of coin tosses it would take to obtain 5 heads. The pmf is given as follows:

$$P(X = k) = f(k) = \binom{k-1}{r-1} p^r (1-p)^{k-r}$$

The expectation and variance are given respectively by the following expression:

$$E(X) = \frac{pr}{1-p}$$

$$Var(X) = \frac{pr}{(1-p)^2}$$

We can see that the negative binomial is a generalization of the geometric distribution, with the geometric distribution being a special case of the negative binomial, where $r = 1$.

The code and plot are shown as follows:

```
In [189]: from scipy.stats import nbinom
       from matplotlib import colors
   clrs = matplotlib.rcParams['axes.color_cycle']
   x = np.arange(0,11)
   n_vals = [0.1,1,3,6]
   p=0.5
       for n, clr in zip(n_vals, clrs):
           rv = nbinom(n,p)
           plt.plot(x,rv.pmf(x), label="$n$=" + str(n), color=clr)
           plt.legend()
       plt.title("Negative Binomial Distribution PMF")
       plt.ylabel("PMF at $x$")
       plt.xlabel("$x$")
```
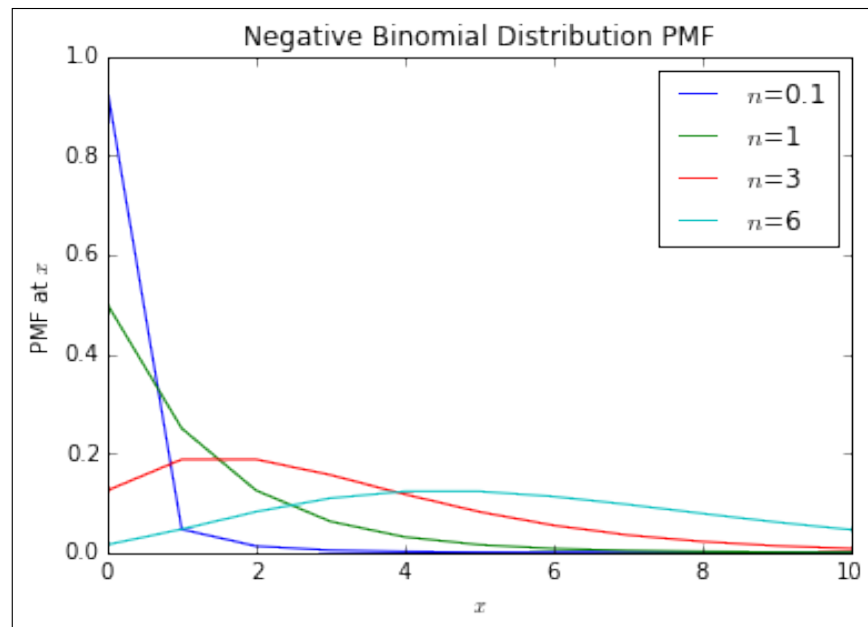
Continuous probability distributions

In a continuous probability distribution, the variable can take on any real number. It is not limited to a finite set of values as with the discrete probability distribution. For example, the average weight of a healthy newborn baby can range approximately between 6-9 lbs. Its weight can be 7.3 lbs for example. A continuous probability distribution is characterized by a **probability density function** (**PDF**).

The sum of all probabilities that the random variable can assume is 1. Thus, the area under the graph of the probability density function is 1.

The continuous uniform distribution

The uniform distribution models a random variable X that can take any value within the range $[a,b]$ with equal probability.

The PDF is given by $f(x) = \dfrac{1}{b-a}$, for $a \leq x \leq b$, and 0 otherwise.

The expectation and variance are given by the following expression:

$$E(x) = (a+b)/2$$

$$Var(x) = (b-a)^2/12$$

A continuous uniform probability distribution is generated and plotted for various sample sizes in the following code and figure:

```
In [11]: np.random.seed(100)   # seed the random number generator
                               # so plots are reproducible
         subplots = [111,211,311]
         ctr = 0
         fig, ax = plt.subplots(len(subplots), figsize=(10,12))
         nsteps=10
         for i in range(0,3):
           cud = np.random.uniform(0,1,nsteps) # generate distrib
           count, bins, ignored = ax[ctr].hist(cud,15,normed=True)
           ax[ctr].plot(bins,np.ones_like(bins),linewidth=2, color='r')
           ax[ctr].set_title('sample size=%s' % nsteps)
           ctr += 1
           nsteps *= 100
         fig.subplots_adjust(hspace=0.4)
         plt.suptitle("Continuous Uniform probability distributions for
various sample sizes" , fontsize=14)
```

Continuous Uniform probability distributions for various sample sizes
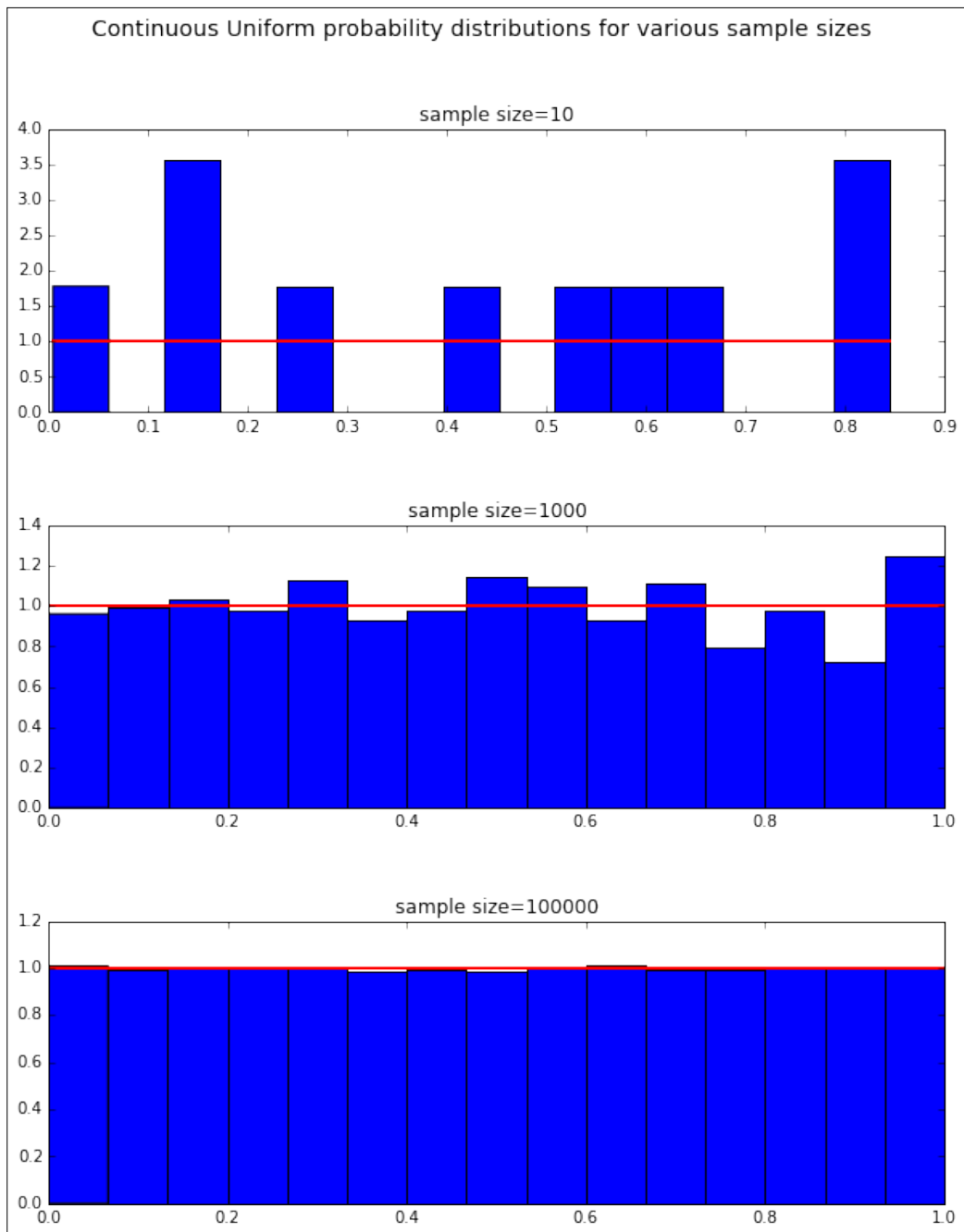
The exponential distribution

The exponential distribution models the waiting time between two events in a Poisson process. A Poisson process is a process that follows a Poisson distribution in which events occur unpredictably with a known average rate. The exponential distribution can be described as the *continuous limit* of the Geometric distribution and is also Markovian (memoryless).

A memoryless random variable exhibits the property whereby its future state depends only on relevant information about the current time and not the information from further in the past. An example of modeling a Markovian/memoryless random variable is modeling short-term stock price behavior and the idea that it follows a random walk. This leads to what is called the Efficient Market hypothesis in Finance. For more information, refer to `http://en.wikipedia.org/wiki/Random_walk_hypothesis`.

The PDF of the exponential distribution is given by $f(x) = \lambda e^{-\lambda x}$. The expectation and variance are given by the following expression:

$$E(X) = 1/\lambda$$
$$Var(X) = 1/\lambda^2$$

For a reference, refer to the link at `http://en.wikipedia.org/wiki/Exponential_distribution`.

The plot of the distribution and code is given as follows:

```
In [15]: import scipy.stats
         clrs = colors.cnames
           x = np.linspace(0,4, 100)
         expo = scipy.stats.expon
        lambda_ = [0.5, 1, 2, 5]
       plt.figure(figsize=(12,4))
       for l,c in zip(lambda_,clrs):
           plt.plot(x, expo.pdf(x, scale=1./l), lw=2,
                     color=c, label = "$\lambda = %.1f$"%l)
             plt.legend()
             plt.ylabel("PDF at $x$")
       plt.xlabel("$x$")
       plt.title("Pdf of an Exponential random variable for various $\
lambda$");
```
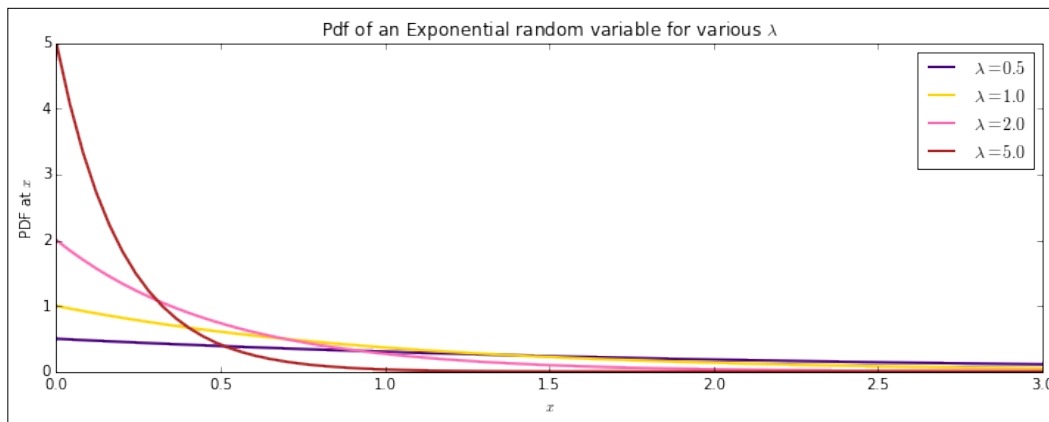
The normal distribution

The most important distribution in statistics is arguably the normal/Gaussian distribution. It models the probability distribution around a central value with no left or right bias. There are many examples of phenomena that follow the normal distribution, such as:

- The birth weights of babies
- Measurement errors
- Blood pressure
- Test scores

The normal distribution's importance is underlined by the central limit theorem, which states that the mean of many random variables drawn independently from the same distribution is approximately normal, regardless of the form of the original distribution. Its expected value and variance are given as follows:

$$E(X) = \mu$$
$$Var(X) = \sigma^2$$

The PDF of the normal distribution is given by the following expression:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

The following code and plot explains the formula:

```
In [54]: import matplotlib
    from scipy.stats import norm
        X = 2.5
    dx = 0.1
    R = np.arange(-X,X+dx,dx)

    L = list()
    sdL = (0.5,1,2,3)
    for sd in sdL:
        f = norm.pdf
        L.append([f(x,loc=0,scale=sd) for x in R])

    colors = matplotlib.rcParams['axes.color_cycle']
    for sd,c,P in zip(sdL,colors,L):
        plt.plot(R,P,zorder=1,lw=1.5,color=c,
                label="$\sigma$=" + str(sd))
        plt.legend()

    ax = plt.axes()
    ax.set_xlim(-2.1,2.1)
    ax.set_ylim(0,1.0)
    plt.title("Normal distribution Pdf")
    plt.ylabel("PDF at $\mu$=0, $\sigma$")
```

Normal distribution Pdf
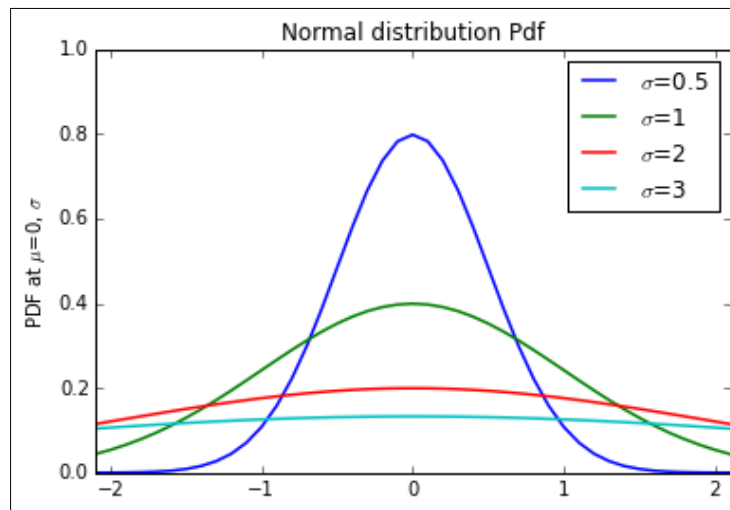
Reference for the Python code for the plotting of the distributions can be found at: `http://bit.ly/1E17nYx`.

The normal distribution can also be regarded as the continuous limit of the binomial distribution and other distributions as $n \rightarrow \infty$. We can see this for the binomial distribution in the command and plots as follows:

```
In [18]:from scipy.stats import binom
     from matplotlib import colors
  cols = colors.cnames
  n_values = [1, 5,10, 30, 100]

  subplots = [111+100*x for x in range(0,len(n_values))]
  ctr = 0
  fig, ax = plt.subplots(len(subplots), figsize=(6,12))
  k = np.arange(0, 200)
  p=0.5

  for n, color in zip(n_values, cols):
        k=np.arange(0,n+1)
        rv = binom(n, p)
        ax[ctr].plot(k, rv.pmf(k), lw=2, color=color)
        ax[ctr].set_title("$n$=" + str(n))
        ctr += 1
```

```
fig.subplots_adjust(hspace=0.5)
plt.suptitle("Binomial distribution PMF (p=0.5) for various values of
n", fontsize=14)
```
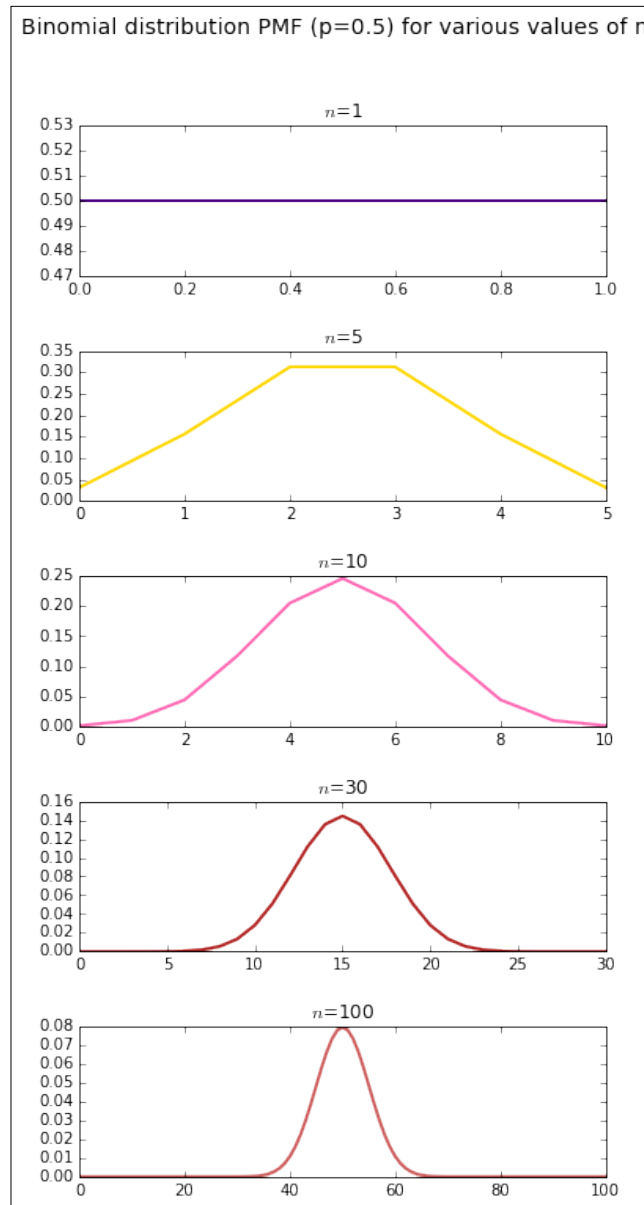

As n increases, the binomial distribution approaches the normal distribution. In fact, for *n>=30*, this is clearly seen in the preceding plots.

Bayesian statistics versus Frequentist statistics

In statistics today, there are two schools of thought as to how we interpret data and make statistical inferences. The classic and more dominant approach to date has been what is termed the Frequentist approach (refer to *Chapter 7*, *A Tour of Statistics – The Classical Approach*), while we are looking at the Bayesian approach in this chapter.

What is probability?

At the heart of the debate between the Bayesian and Frequentist worldview is the question—how do we define probability?

In the Frequentist worldview, probability is a notion that is derived from the frequencies of repeated events. For example, when we define the probability of getting heads when a fair coin is tossed as being equal to half. This is because when we repeatedly toss a fair coin, the number of heads divided by the total number of coin tosses approaches 0.5 when the number of coin tosses is sufficiently large.

The Bayesian worldview is different, and the notion of probability is that it is related to one's degree of belief in the event happening. Thus, for a Bayesian statistician, having a belief that the probability of a fair die turning up 5 is $1/6$ relates to our belief in the chances of that event occurring.

How the model is defined

From the model definition point of view Frequentists analyze how data and calculated metrics vary by making use of repeated experiments while keeping the model parameters fixed. Bayesians, on the other hand, utilize fixed experimental data but vary their degrees of belief in the model parameters, this is explained as follows:

- **Frequentists**: If the models are fixed, data varies
- **Bayesians**: If the data is fixed, models vary

The Frequentist approach uses what is known as the maximum likelihood method to estimate model parameters. It involves generating data from a set of independent and identically distributed observations and fitting the observed data to the model. The value of the model parameter that best fits the data is the **maximum likelihood estimator** (**MLE**), which can sometimes be a function of the observed data.

Bayesianism approaches the problem differently from a probabilistic framework. A probability distribution is used to describe the uncertainty in the values. Bayesian practitioners estimate probabilities using observed data. In order to compute these probabilities, they make use of a single estimator, which is the Bayes formula. This produces as distribution rather than just a point estimate, as in the case of the Frequentist approach.

Confidence (Frequentist) versus Credible (Bayesian) intervals

Let us compare what is meant by a 95 percent confidence interval, a term used by Frequentists with a 95 percent credible interval, used by Bayesian practitioners.

In a Frequentist framework, a 95 percent confidence interval means that if you repeat your experiment an infinite number of times, generating intervals in the process, 95 percent of these intervals would contain the parameter we're trying to estimate, which is often referred to as θ. In this case, the interval is the random variable and not the parameter estimate θ, which is fixed in the Frequentist worldview.

In the case of the Bayesian credible interval, we have an interpretation that is more in-line with the conventional interpretation ascribed to that of a Frequentist confidence interval. Thus, we have that $\Pr\left(a\left(Y\right) < \theta < b\left(Y\right) | \theta\right) = 0.95$. In this case, we can properly conclude that there is a 95 percent chance that θ lies within the interval.

For more information, refer to *Frequentism and Bayesianism: What's the Big Deal? | SciPy 2014 | Jake VanderPlas* at `https://www.youtube.com/watch?v=KhAUfqhLakw`.

Conducting Bayesian statistical analysis

Conducting a Bayesian statistical analysis involves the following steps:

1. **Specifying a probability model**: In this step, we fully describe the model using a probability distribution. Based on the distribution of a sample that we have taken, we try to fit a model to it and attempt to assign probabilities to unknown parameters.

2. **Calculating a posterior distribution**: The posterior distribution is a distribution that we calculate in light of observed data. In this case, we will directly apply Bayes formula. It will be specified as a function of the probability model that we specified in the previous step.

3. **Checking our model**: This is a necessary step where we review our model and its outputs before we make inferences. Bayesian inference methods use probability distributions to assign probabilities to possible outcomes.

Monte Carlo estimation of the likelihood function and PyMC

Bayesian statistics isn't just another method. It is an entirely alternative paradigm for practicing statistics. It uses probability models for making inferences, given the data that we have collected. This can be expressed in a fundamental expression as P(H|D).

Here, H is our hypothesis, that is, the thing we're trying to prove, and D is our data or observations.

As a reminder from our previous discussion, the diachronic form of Bayes' theorem is as follows:

$$P\left(\text{H}|\text{D}\right) = P\left(H\right) * \frac{P\left(\text{D}|\text{H}\right)}{P\left(D\right)}$$

Here, P(H) is an unconditional prior probability that represents what we know before we conduct our trial. P(D|H) is our likelihood function or probability of obtaining the data we observe, given that our hypothesis is true.

P(D) is the probability of the data, also known as the normalizing constant. This can be obtained by integrating the numerator over H.

The likelihood function is the most important piece in our Bayesian calculation and encapsulates all of the information concerning the unknowns in the data. It has some semblance to a reverse probability mass function.

One argument against adopting a Bayesian approach is that the calculation of the prior can be subjective. There are many arguments in favor of this approach; among them, one being that external prior information can be included as mentioned previously.

The likelihood value represents an unknown integral, which in simple cases can be obtained by analytic integration.

Monte Carlo (**MC**) integration is needed for more complicated use cases involving higher-dimensional integrals and can be used to compute the likelihood function.

MC integration can be computed via a variety of sampling methods, such as uniform sampling, stratified sampling, and importance sampling. In Monte Carlo Integration, we can approximate the integral as follows:

$$Pg = \int g dP$$

We can approximate the integral by the following finite sum:

$$P_n g = \frac{1}{n} \sum_{i=1}^{n} g(X_i)$$

where, x is a sample vector from g. The proof that this estimate is a good one can be obtained from the law of large numbers and by making sure that the simulation error is small.

In conducting Bayesian analysis in Python, we will need a module that will enable us to calculate the likelihood function using the Monte Carlo method that was described earlier. The `PyMC` library fulfills that need. It provides a Monte Carlo method known commonly as **Markov Chain Monte Carlo** (**MCMC**). I will not delve further into the technical details of MCMC, but the interested reader can find out more about MCMC implementation in `PyMC` at the following references:

- *Monte Carlo Integration in Bayesian Estimation* at `http://bit.ly/1bMALeu`
- *Markov Chain Monte Carlo Maximum Likelihood* at `http://bit.ly/1KBP8hH`
- *Bayesian Statistical Analysis Using Python-Part 1| SciPy 2014, Chris Fonnesbeck* at `http://www.youtube.com/watch?v=vOBB_ycQ0RA`

MCMC is not a universal panacea; there are some drawbacks to the approach, and one of them is the slow convergence of the algorithm.

Bayesian analysis example – Switchpoint detection

Here, we will try to use Bayesian inference and model an interesting dataset. The dataset in question consists of the author's **Facebook** (**FB**) post history over time. We have scrubbed the FB history data and saved the dates in the `fb_post_dates.txt` file. Here is what the data in the file looks like:

```
head -2 ../fb_post_dates.txt
Tuesday, September 30, 2014 | 2:43am EDT
Tuesday, September 30, 2014 | 2:22am EDT
```

Thus, we see a datetime series, representing the date and time at which the author posted on FB. First, we read the file into DataFrame, separating timestamp into Date and Time columns:

```
In [91]: filePath="./data/fb_post_dates.txt"

        fbdata_df=pd.read_csv(filePath,  sep='|', parse_dates=[0], heade
r=None,names=['Date','Time'])
```

Next, we inspect the data as follows:

```
In [92]: fbdata_df.head()  #inspect the data
Out[92]:   Date        Time
0  2014-09-30   2:43am EDT
1  2014-09-30   2:22am EDT
2  2014-09-30   2:06am EDT
3  2014-09-30   1:07am EDT
4  2014-09-28   9:16pm EDT
```

Now, we index the data by Date, creating a `DatetimeIndex` so that we can run resample on it to count by month as follows:

```
In [115]: fbdata_df_ind=fbdata_df.set_index('Date')
          fbdata_df_ind.head(5)
Out[115]:                       Time

        Date
        2014-09-30   2:43am EDT
        2014-09-30   2:22am EDT
        2014-09-30   2:06am EDT
        2014-09-30   1:07am EDT
        2014-09-28   9:16pm EDT
```

We display information about the index as follows:

```
In [116]: fbdata_df_ind.index
Out[116]: <class 'pandas.tseries.index.DatetimeIndex'>
          [2014-09-30, ..., 2007-04-16]
          Length: 7713, Freq: None, Timezone: None
```

We now obtain count of posts by month, using resample:

```
In [99]: fb_mth_count_=fbdata_df_ind.resample('M', how='count')

         fb_mth_count_.rename(columns={'Time':'Count'},

                                 inplace=True)    # Rename

         fb_mth_count_.head()
Out[99]:            Count

         Date

         2007-04-30  1

         2007-05-31  0

         2007-06-30  5

         2007-07-31  50

         2007-08-31  24
```

The Date format is shown as the last day of the month. Now, we create a scatter plot of FB post counts from 2007-2015, and we make the size of the dots proportional to the values in `matplotlib`:

```
In [108]: %matplotlib inline

          import datetime as dt
#Obtain the count data from the DataFrame as a dictionary

          year_month_count=fb_bymth_count.to_dict()['Count']

          size=len(year_month_count.keys())
#get dates as list of strings

          xdates=[dt.datetime.strptime(str(yyyymm),'%Y%m')

                 for yyyymm in year_month_count.keys()]

          counts=year_month_count.values()

          plt.scatter(xdates,counts,s=counts)

         plt.xlabel('Year')

          plt.ylabel('Number of Facebook posts')

          plt.show()
```
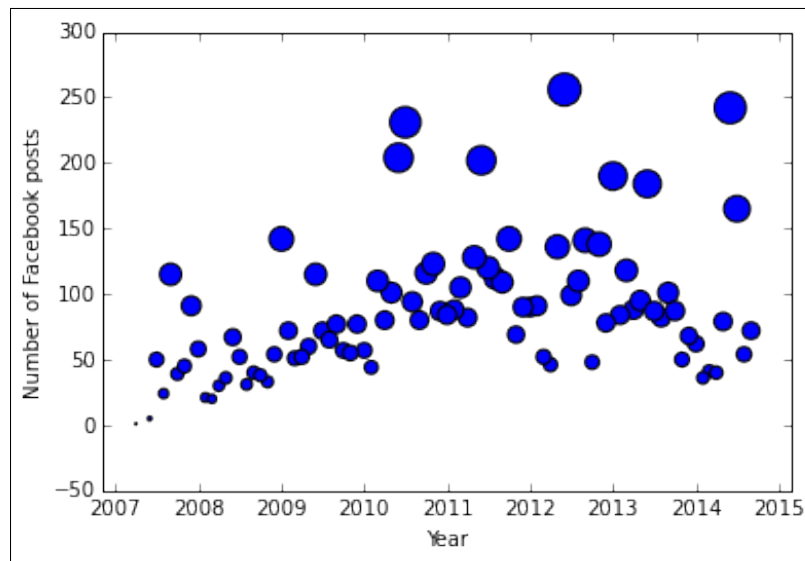
The question we would like to investigate is whether there was a change in behavior at some point over the time period. Specifically, we wish to identify whether there was a specific period at which the mean number of FB posts changed. This is often referred to as the Switchpoint or changepoint in a time series.

We can make use of the Poisson distribution to model this. You might recall that the Poisson distribution can be used to model time series count data. (Refer to `http://bit.ly/1JniIqy` for more about this.)

If we represent our monthly FB post count by C_i, we can represent our model as follows:

$$(C_i \mid s, e, l) \sim Poisson(r_i)$$

The r_i parameter is the rate parameter of the Poisson distribution, but we don't know what its value is. If we examine the scatter plot of the FB time series count data, we can see that there was a jump in the number of posts sometime around mid to late 2010, perhaps coinciding with the start of the 2010 World Cup in South Africa, which the author attended.

The s parameter is the Switchpoint, which is when the rate parameter changes, while e and l are the values of the r_i parameter before and after the Switchpoint respectively. This can be represented as follows:

$$r = \begin{cases} e & if \ i < s \\ l & if \ i \geq s \end{cases}$$

Note that the variables specified above C, s, e, r, l are all Bayesian random variables. For Bayesian random variables that represent one's beliefs about their values, we need to model them using a probability distribution. We would like to infer the values of e and l, which are unknown. In PyMC, we can represent random variables using the Stochastic and Deterministic classes. We note that the exponential distribution is the amount of time between Poisson events. Hence, in the case of e and l, we choose the exponential distribution to model them since they can be any positive number:

$$e \sim Exp(r)$$

$$l \sim Exp(r)$$

In the case of s, we will choose to model it using the uniform distribution, which reflects our belief that it is equally likely that the Switchpoint can occur on any day within the entire time period. Hence, we have this:

$$s \sim DiscreteUniform(t_0 \ t_f)$$

Here, t_0, t_f corresponds to the lower and upper boundaries of the year i. Let us now use PyMC to represent the model that we developed earlier. We will now use PyMC to see whether we can detect a Switchpoint in the FB post data. In addition to the scatter plot, we can also display the data in a bar chart. In order to do that first of all we need to obtain a count of FB posts ordered by month in a list:

```
In [69]: fb_activity_data = [year_month_count[k] for k in
                            sorted(year_month_count.keys())]
         fb_activity_data[:5]
```

```
Out[70]: [1, 0, 5, 50, 24]
```

```
In [71]: fb_post_count=len(fb_activity_data)
```

We render the bar plot using `matplotlib`:

```
In [72]: from IPython.core.pylabtools import figsize
         import matplotlib.pyplot as plt
           figsize(8, 5)
         plt.bar(np.arange(fb_post_count),
          fb_activity_data, color="#49a178")
         plt.xlabel("Time (months)")
         plt.ylabel("Number of FB posts")
         plt.title("Monthly Facebook posts over time")
         plt.xlim(0,fb_post_count);
```
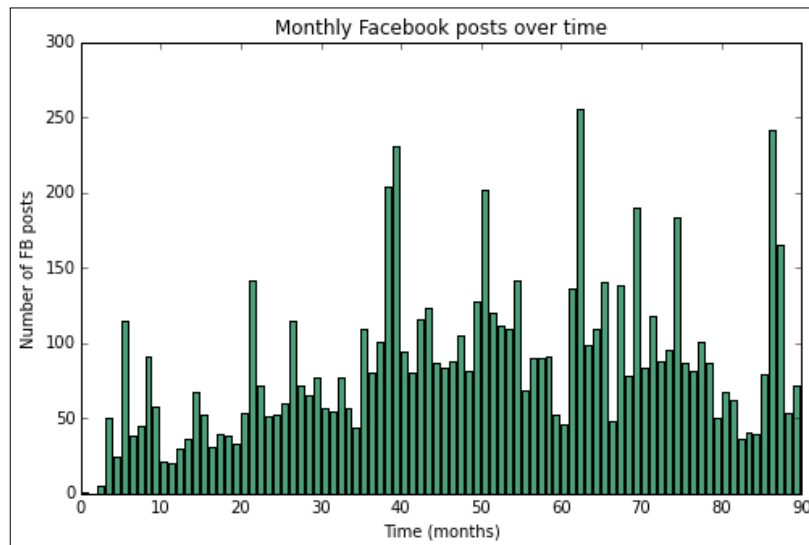

Looking at the preceding bar chart, can one conclude whether there was a change in FB frequency posting behavior over time? We can use `PyMC` on the model that we have developed to help us find out the change as follows:

```
In [88]: # Define data and stochastics
         import pymc as pm
         switchpoint = pm.DiscreteUniform('switchpoint',
                                          lower=0,
                              upper=len(fb_activity_data)-1,
                                doc='Switchpoint[month]')
         avg = np.mean(fb_activity_data)
         early_mean = pm.Exponential('early_mean', beta=1./avg)
```

```
        late_mean = pm.Exponential('late_mean', beta=1./avg)
        late_mean
Out[88]:<pymc.distributions.Exponential 'late_mean' at 0x10ee56d50>
```

Here, we define a method for the rate parameter, r, and we model the count data using a Poisson distribution as discussed previously:

```
In [89]: @pm.deterministic(plot=False)
        def rate(s=switchpoint, e=early_mean, l=late_mean):
            ''' Concatenate Poisson means '''
            out = np.zeros(len(fb_activity_data))
            out[:s] = e
            out[s:] = l
        return out


        fb_activity = pm.Poisson('fb_activity', mu=rate,
                        value=fb_activity_data, observed=True)
        fb_activity
Out[89]: <pymc.distributions.Poisson 'fb_activity' at 0x10ed1ee50>
```

In the preceding code snippet, `@pm.deterministic` is a decorator that denotes that the rate function is deterministic, meaning that its values are entirely determined by other variables—in this case, e, s, and l. The decorator is necessary in order to tell `PyMC` to convert the rate function into a deterministic object. If we do not specify the decorator, an error occurs. (For more information, refer to `http://bit.ly/1zj8U0o` for information on Python decorators.)

For more information, refer to the following web pages:

- `http://en.wikipedia.org/wiki/Poisson_process`
- `http://pymc-devs.github.io/pymc/tutorial.html`
- `https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers`

We now create a model with the FB Count data (`fb_activity`) and the e, s, l (`early_mean`, `late_mean`, and `rate` respectively) parameters.

Next, using `Pymc`, we create an MCMC object that enables us to fit our data using Markov Chain Monte Carlo methods. We then call the sample on the resulting MCMC object to do the fitting:

```
In [94]: fb_activity_model=pm.Model([fb_activity,early_mean,
                                    late_mean,rate])
```

```
In [95]: from pymc import MCMC

         fbM=MCMC(fb_activity_model)
In [96]: fbM.sample(iter=40000,burn=1000, thin=20)

         [----------------100%-----------------] 40000 of 40000

         complete in 11.0 sec
```

Fitting the model using MCMC involves using Markov-Chain Monte Carlo methods to generate a probability distribution for the posterior, *P(s,e,l | D)*. It uses the Monte Carlo process to repeatedly simulate sampling of the data and does this until the algorithm seems to converge to a steady state, based on multiple criteria. This is a Markov process because successive samples are dependent only on the previous sample. (For further reference on Markov chain convergence, refer to `http://bit.ly/1IETkhC`.)

The generated samples are referred to as traces. We can view what the marginal posterior distribution of the parameters looks like by viewing a histogram of its *trace*:

```
In [97]: from pylab import hist,show
         %matplotlib inline
       hist(fbM.trace('late_mean')[:])
Out[97]: (array([  15.,    61.,   214.,   421.,   517.,   426.,   202.,
                   70.,    21.,    3.]),
          array([ 102.29451192,  103.25158404,  104.20865616,
                  105.16572829,  106.12280041,  107.07987253,
                  108.03694465,  108.99401677,  109.95108889,
                  110.90816101,  111.86523313]),
    <a list of 10 Patch objects>)
```
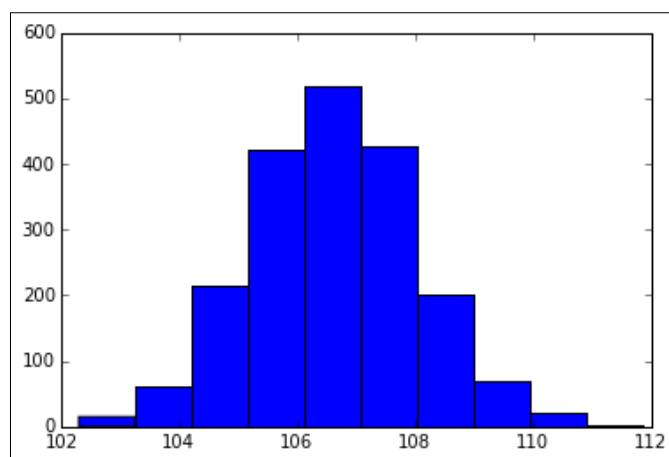
```
In [98]:plt.hist(fbM.trace('early_mean')[:])
Out[98]: (array([  20.,  105.,  330.,  489.,  470.,  314.,  147.,
                   60.,    3.,   12.]),
          array([ 49.19781192,  50.07760882,  50.95740571,
                  51.83720261,  52.71699951,  53.59679641,
                  54.47659331,  55.35639021,  56.2361871 ,
                  57.115984  ,  57.9957809 ]),
          <a list of 10 Patch objects>)
```
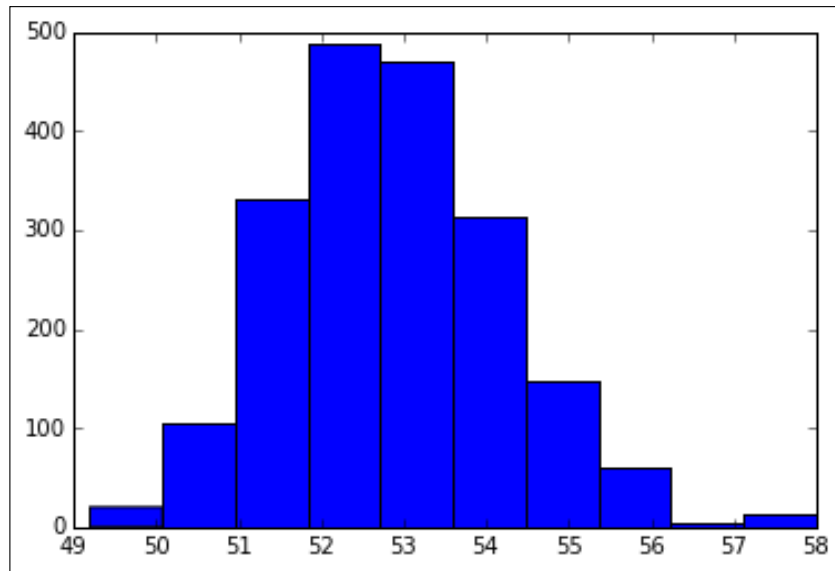

Here, we see what the Switchpoint in terms of number of months looks like:

```
In [99]: fbM.trace('switchpoint')[:]
Out[99]: array([38, 38, 38, ..., 35, 35, 35])

In [150]: plt.hist(fbM.trace('switchpoint')[:])
Out[150]: (array([ 1899.,     0.,     0.,     0.,     0.,     0.,
                      0., 0., 0.,    51.]),
           array([ 35. ,  35.3,  35.6,  35.9,  36.2,  36.5,  36.8,
                   37.1,  37.4,  37.7,  38. ]),
           <a list of 10 Patch objects>)
```
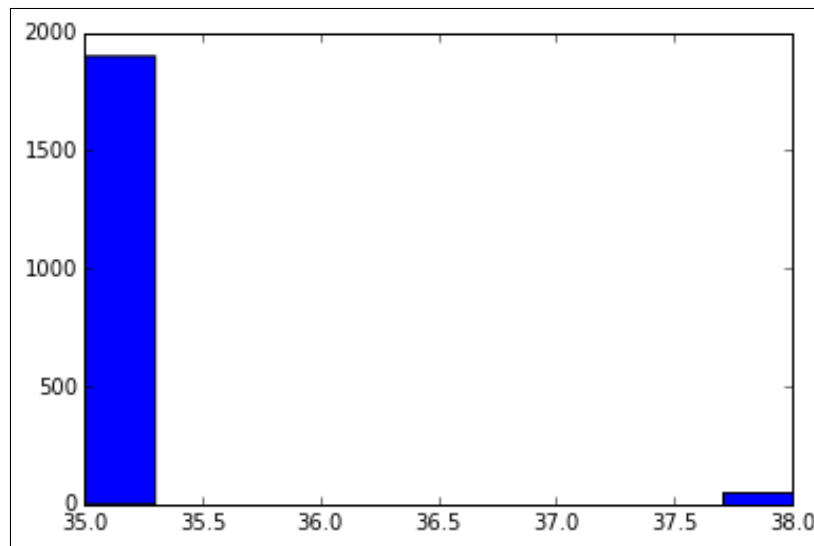
Hist Switchpoint

We can see that the Switchpoint is in the neighborhood of the months numbering 35-38. Here, we use `matplotlib` to display the marginal posterior distributions of e, s, and l in a single figure:

```
In [141]: early_mean_samples=fbM.trace('early_mean')[:]
          late_mean_samples=fbM.trace('late_mean')[:]
          switchpoint_samples=fbM.trace('switchpoint')[:]
In [142]: from IPython.core.pylabtools import figsize
      figsize(12.5, 10)
    # histogram of the samples:
    fig = plt.figure()
    fig.subplots_adjust(bottom=-0.05)

    n_mths=len(fb_activity_data)
    ax = plt.subplot(311)
    ax.set_autoscaley_on(False)

     plt.hist(early_mean_samples, histtype='stepfilled',
             bins=30, alpha=0.85, label="posterior of $e$",
             color="turquoise", normed=True)
    plt.legend(loc="upper left")
```

```
plt.title(r"""Posterior distributions of the variables
          $e, l, s$""",fontsize=16)
plt.xlim([40, 120])
 plt.ylim([0, 0.6])
plt.xlabel("$e$ value",fontsize=14)


 ax = plt.subplot(312)
ax.set_autoscaley_on(False)
plt.hist(late_mean_samples, histtype='stepfilled',
          bins=30, alpha=0.85, label="posterior of $l$",
          color="purple", normed=True)
plt.legend(loc="upper left")
plt.xlim([40, 120])
 plt.ylim([0, 0.6])
plt.xlabel("$l$ value",fontsize=14)
plt.subplot(313)
w = 1.0 / switchpoint_samples.shape[0] *
     np.ones_like(switchpoint_samples)
plt.hist(switchpoint_samples, bins=range(0,n_mths), alpha=1,
          label=r"posterior of $s$", color="green",
          weights=w, rwidth=2.)
plt.xlim([20, n_mths - 20])
plt.xlabel(r"$s$ (in days)",fontsize=14)
plt.ylabel("probability")
plt.legend(loc="upper left")


 plt.show()
```

Posterior distributions of the variables e, l, s
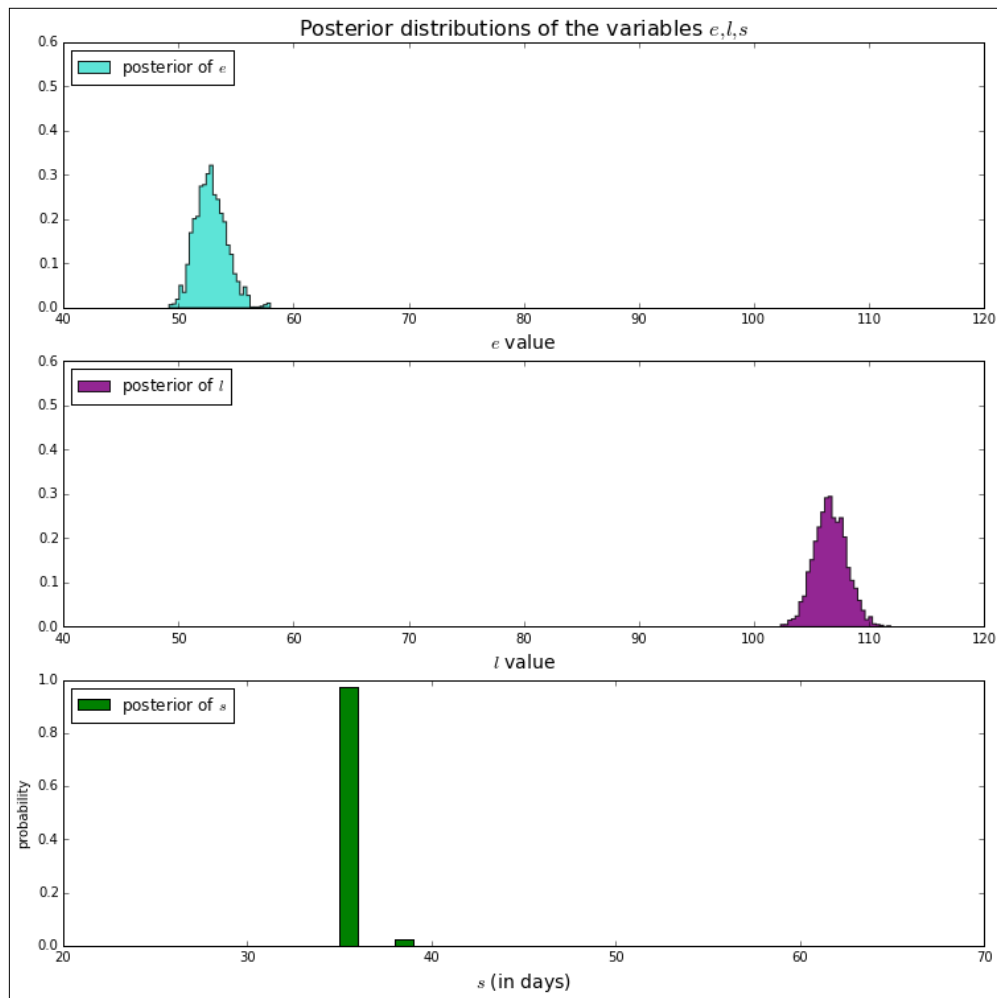
marginal posterior distributions

PyMC also has plotting functionality. (It uses `matplotlib`.) In the following plots, we display a time series plot, an autocorrelation plot (acorr), and a histogram of the samples drawn for the early mean, late mean, and the Switchpoint. The histogram is useful to visualize the posterior distribution. The autocorrelation plot shows whether values in the previous period are strongly related to values in the current period.

```
In [100]: from pymc.Matplot import plot
      plot(fbM)
   Plotting late_mean
   Plotting switchpoint
   Plotting early_mean
```

The following is the late mean plot:
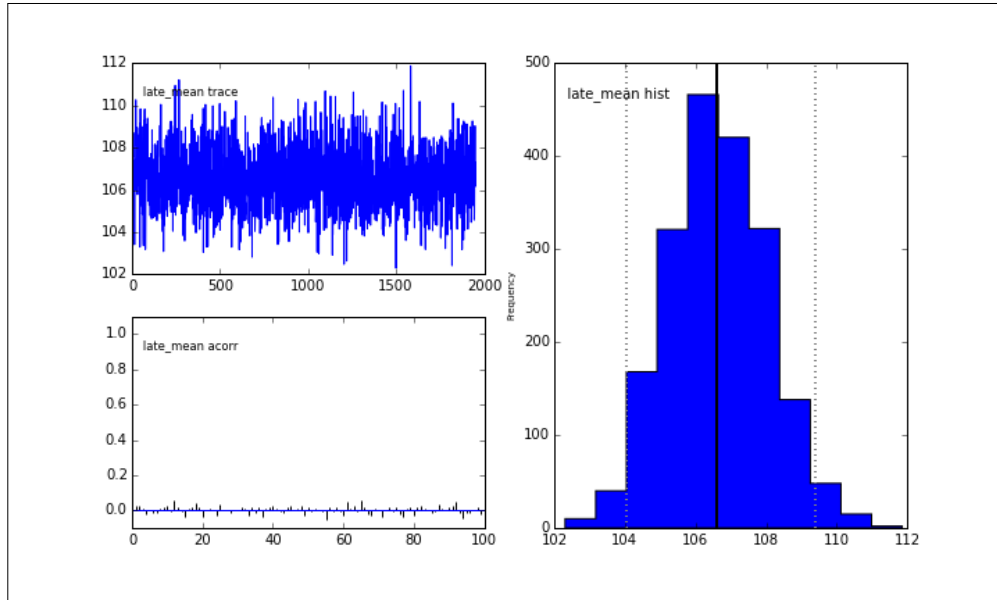
pymc_comprehensive_late_mean

Here, we display the Switchpoint plot:
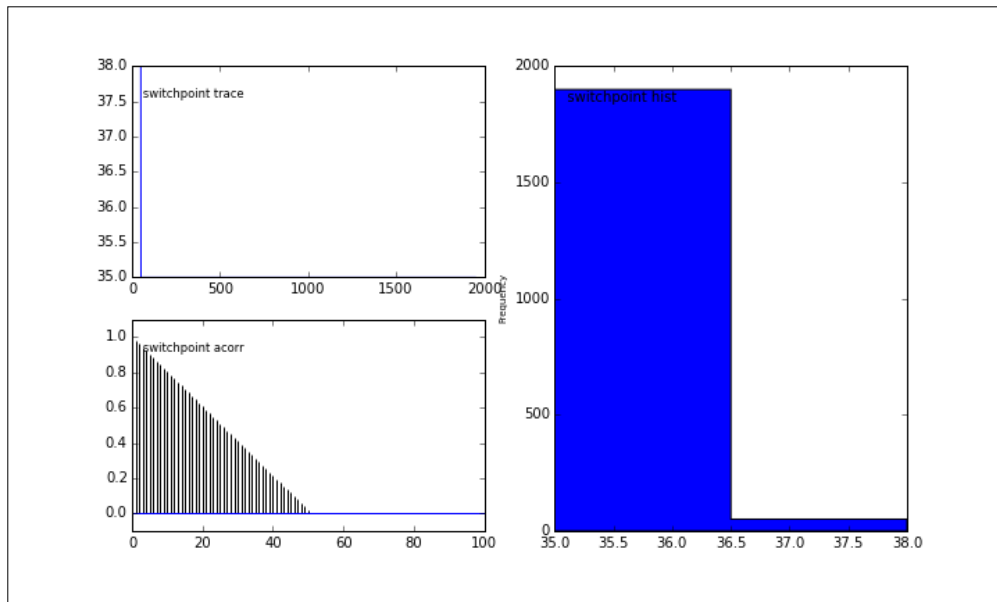
Pymc comprehensive Switchpoint

Here, we display the early mean plot:
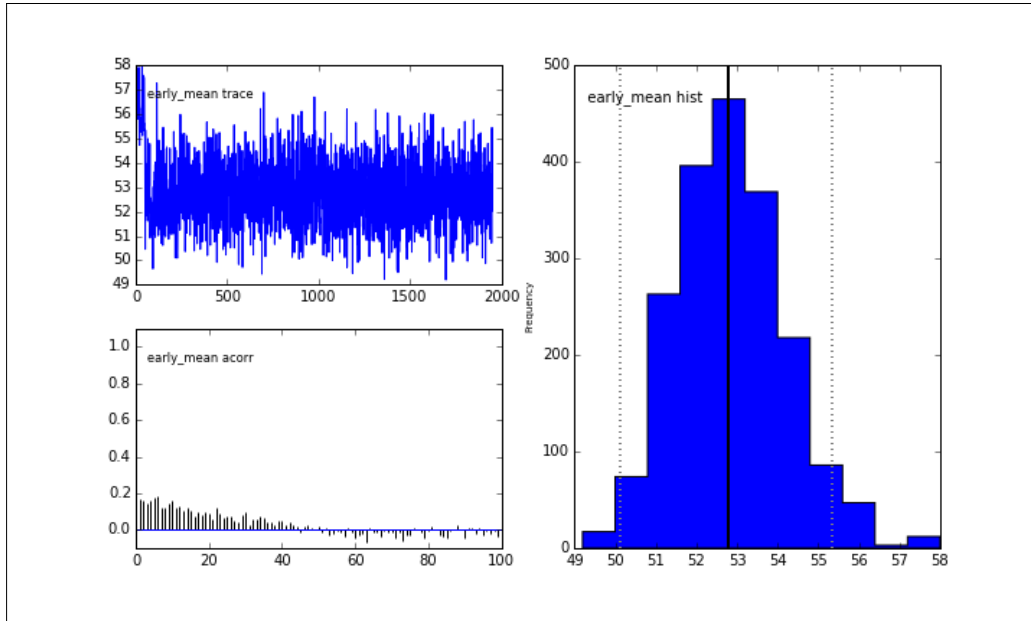
Pymc comprehensive early mean

From the output of PyMC, we can conclude that the Switchpoint is around 35-38 months from the start of the time series. This corresponds to sometime around March-July 2010. The author can testify that this was a banner year for him with respect to the use of FB since it was the year of the football (soccer) World Cup finals that were held in South Africa, which he attended.

References

For a more in-depth look at Bayesian statistics topics that we touched upon, please take a look at the following references:

- *Probabilistic Programming and Bayesian Methods for Hackers* at `https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers`

- *Bayesian Data Analysis*, *Third Edition*, *Andrew Gelman* at `http://www.amazon.com/Bayesian-Analysis-Chapman-Statistical-Science/dp/1439840954`

- *The Bayesian Choice*, *Christian P Robert* (this is more theoretical) at `http://www.springer.com/us/book/9780387952314`

- *PyMC documentation* at `http://pymc-devs.github.io/pymc/index.html`

Summary

In this chapter, we undertook a whirlwind tour of one of the hottest trends in statistics and data analysis in the past few years—the Bayesian approach to statistical inference. We covered a lot of ground here.

We examined what the Bayesian approach to statistics entails and discussed the various factors as to why the Bayesian view is a compelling one—facts over belief. We explained the key statistical distributions and showed how we can use the various statistical packages to generate and plot them in `matplotlib`.

We tackled a rather difficult topic without too much oversimplification and demonstrated how we can use the PyMC package and Monte Carlo simulation methods to showcase the power of Bayesian statistics to formulate models, do trend analysis, and make inferences on a real-world dataset (Facebook user posts). In the next chapter, we will discuss the pandas library architecture.

<div align="right">

9

</div>

The pandas Library
Architecture

In this chapter, we examine the various libraries that are available to pandas' users. This chapter is intended to be a short guide to help the user navigate and find their way around the various modules and libraries that pandas provide. It gives a breakup of how the library code is organized, and it also provides a brief description on the various modules. It will be most valuable to users who are interested to see the inner workings of pandas underneath, as well as to those who wish to make contributions to the code base. We will also briefly demonstrate how you can improve performance using Python extensions. The various topics that will be discussed are as follows:

- Introduction to pandas' library hierarchy
- Description of pandas' modules and files
- Improving performance using Python extensions

Introduction to pandas' file hierarchy

Generally, upon installation, pandas gets installed as a Python module in a standard location for third-party Python modules:

Platform	Standard installation location	Example
Unix/Mac OS	`prefix/lib/pythonX.Y/site-packages`	`/usr/local/lib/python2.7/site-packages`
Windows	`prefix\Lib\site-packages`	`C:\Python27\Lib\site-packages`

The installed files follow a specific hierarchy:

- `pandas/core`: This contains files for fundamental data structures such as Series/DataFrames and related functionality.
- `pandas/src`: This contains Cython and C code for implementing fundamental algorithms.
- `pandas/io`: This contains input/output tools (such as flat files, Excel, HDF5, SQL, and so on).
- `pandas/tools`: This contains auxiliary data algorithms merge and join routines, concatenation, pivot tables, and more.
- `pandas/sparse`: This contains sparse versions of Series, DataFrame, Panel and more.
- `pandas/stats`: This contains linear and Panel regression, and moving window regression. This should be replaced by functionality in statsmodels.
- `pandas/util`: This contains utilities, development, and testing tools.
- `pandas/rpy`: This contains RPy2 interface for connecting to R.

> For reference see: `http://pandas.pydata.org/developers.html`.

Description of pandas' modules and files

In this section, we provide brief descriptions of the various submodules and files that make up pandas' library.

pandas/core

This module contains the core submodules of pandas. They are discussed as follows:

- `api.py`: This imports some key modules for later use.
- `array.py`: This isolates pandas' exposure to numPy, that is, all direct numPy usage.
- `base.py`: This defines fundamental classes, such as `StringMixin`, `PandasObject` which is the base class for various pandas objects such as `Period, PandasSQLTable, sparse.array.SparseArray/SparseList, internals.Block, internals.BlockManager, generic.NDFrame, groupby. GroupBy, base.FrozenList, base.FrozenNDArray, io.sql.PandasSQL, io.sql.PandasSQLTable, tseries.period.Period, FrozenList, FrozenNDArray: IndexOpsMixin`, and `DatetimeIndexOpsMixin`.

- `common.py`: This defines common utility methods for handling data structures. For example `isnull` object detects missing values.
- `config.py`: This is the module for handling package-wide configurable objects. It defines the following classes: `OptionError`, `DictWrapper`, `CallableDynamicDoc`, `option_context`, `config_init`.
- `datetools.py`: This is a collection of functions that deal with dates in Python.
- `frame.py`: This defines pandas' DataFrame class and its various methods. DataFrame inherits from NDFrame. (see below).
- `generic.py`: This defines the generic NDFrame base class, which is a base class for pandas' DataFrame, Series, and Panel classes. NDFrame is derived from PandasObject, which is defined in `base.py`. An NDFrame can be regarded as an N-dimensional version of a pandas' DataFrame. For more information on this, go to `http://nullege.com/codes/search/pandas.core.generic.NDFrame`.
- `categorical.py`: This defines Categorical, which is a class that derives from PandasObject and represents categorical variables a la R/S-plus. (we will expand your knowledge on this a bit more later).
- `format.py`: This defines a whole host of Formatter classes such as `CategoricalFormatter`, `SeriesFormatter`, `TableFormatter`, `DataFrameFormatter`, `HTMLFormatter`, `CSVFormatter`, `ExcelCell`, `ExcelFormatter`, `GenericArrayFormatter`, `FloatArrayFormatter`, `IntArrayFormatter`, `Datetime64Formatter`, `Timedelta64Formatter`, and `EngFormatter`.
- `groupby.py`: This defines various classes that enable the `groupby` functionality. They are discussed as follows:
    - `Splitter classes`: This includes `DataSplitter`, `ArraySplitter`, `SeriesSplitter`, `FrameSplitter`, and `NDFrameSplitter`
    - `Grouper/Grouping classes`: This includes `Grouper`, `GroupBy`, `BaseGrouper`, `BinGrouper`, `Grouping`, `SeriesGroupBy`, `NDFrameGroupBy`
- `ops.py`: This defines an internal API for arithmetic operations on PandasObjects. It defines functions that add arithmetic methods to objects. It defines a `_create_methods` meta method, which is used to create other methods using arithmetic, comparison, and Boolean method constructors. The `add_methods` method takes a list of new methods, adds them to the existing list of methods, and binds them to their appropriate classes. The `add_special_arithmetic_methods` and `add_flex_arithmetic_methods` methods call `_create_methods` and `add_methods` to add arithmetic methods to a class.

It also defines the `_TimeOp` class, which is a wrapper for datetime-related arithmetic operations. It contains `Wrapper` functions for arithmetic, comparison, and Boolean operations on Series, DataFrame and Panel functions — `_arith_method_SERIES(..)`, `_comp_method_SERIES(..)`, `_bool_method_SERIES(..)`, `_flex_method_SERIES(..)`, `_arith_method_FRAME(..)`, `_comp_method_FRAME(..)`, `_flex_comp_method_FRAME(..)`, `_arith_method_PANEL(..)`, `_comp_method_PANEL(..)`.

- `index.py`: This defines the Index class and its related functionality. Index is used by all pandas' objects — Series, DataFrame, and Panel — to store axis labels. Underneath it is an immutable array that provides an ordered set that can be sliced.

- `internals.py`: This defines multiple object classes. These are listed as follows:
    - `Block`: This is a homogeneously typed N-dimensional `numpy.ndarray` object with additional functionality for pandas. For example, it uses `__slots__` to restrict the attributes of the object to 'ndim', 'values', and '_mgr_locs'. It acts as the base class for other `Block` subclasses.
    - `NumericBlock`: This is the base class for `Block`s with the numeric type.
    - `FloatOrComplexBlock`: This is base class for `FloatBlock` and `ComplexBlock` that inherits from `NumericBlock`
    - `ComplexBlock`: This is the class that handles the `Block` objects with the complex type.
    - `FloatBlock`: This is the class that handles the `Block` objects with the float type.
    - `IntBlock`: This is the class that handles the `Block` objects with the integer type.
    - `TimeDeltaBlock`, `BoolBlock`, and `DatetimeBlock`: These are the `Block` classes for timedelta, Boolean, and datetime.
    - `ObjectBlock`: This is the class that handles `Block` objects for user-defined objects.
    - `SparseBlock`: This is the class that handles sparse arrays of the same type.
    - `BlockManager`: This is the class that manages a set of `Block` objects. It is not a public API class.
    - `SingleBlockManager`: This is the class that manages a single `Block`.
    - `JoinUnit`: This is the utility class for `Block` objects.

- `matrix.py`: This imports DataFrame as `DataMatrix`.
- `nanops.py`: These are the classes and functionality for handling NaN values.
- `ops.py`: This defines arithmetic operations for pandas' objects. It is not a public API.
- `panel.py`, `panel4d.py`, and `panelnd.py`: These provide the functionality for the pandas' Panel object.
- `series.py`: This defines the pandas Series class and its various methods that Series inherits from NDFrame and IndexOpsMixin.
- `sparse.py`: This defines import for handling sparse data structures. Sparse data structures are *compressed* whereby data points matching NaN or missing values are omitted. For more information on this, go to `http://pandas.pydata.org/pandas-docs/stable/sparse.html`.
- `strings.py`: These have various functions for handling strings.
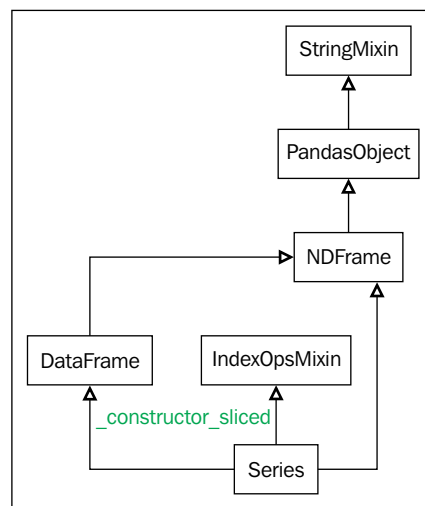
pandas/io

This module contains various modules for data I/O. These are discussed as follows:

- `api.py`: This defines various imports for the data I/O API.
- `auth.py`: This defines various methods dealing with authentication.
- `common.py`: This defines the common functionality for I/O API.

- `data.py`: This defines classes and methods for handling data. The `DataReader` method reads data from various online sources such as Yahoo and Google.

- `date_converters.py`: This defines date conversion functions.

- `excel.py`: This module parses and converts Excel data. This defines `ExcelFile` and `ExcelWriter` classes.

- `ga.py`: This is the module for the Google Analytics functionality.

- `gbq.py` : This is the module for Google's BigQuery.

- `html.py`: This is the module for dealing with HTML I/O.

- `json.py`: This is the module for dealing with json I/O in pandas. This defines the `Writer`, `SeriesWriter`, `FrameWriter`, `Parser`, `SeriesParser`, and `FrameParser` classes.

- `packer.py`: This is a msgpack serializer support for reading and writing pandas data structures to disk.

- `parsers.py`: This is the module that defines various functions and classes that are used in parsing and processing files to create pandas' DataFrames. All the three `read_*` functions discussed as follows have multiple configurable options for reading. See this reference for more details: `http://bit.ly/1e4Xqo1`.

    - `read_csv(..)`: This defines the `pandas.read_csv()` function that is useful to read the contents of a CSV file into a DataFrame.

    - `read_table(..)`: This reads a tab-separated table file into a DataFrame.

    - `read_fwf(..)`: This reads a fixed-width format file into a DataFrame.

    - `TextFileReader`: This is the class that is used for reading text files.

    - `ParserBase`: This is the base class for parser objects.

    - `CParserWrapper`, `PythonParser`: These are the parser for C and Python respectively. They both inherit from `ParserBase`.

    - `FixedWidthReader`: This is the class for reading fixed-width data. A fixed-width data file contains fields in specific positions within the file.

    - `FixedWithFieldParser`: This is the class for parsing fixed-width fields that have been inherited from `PythonParser`.

- `pickle.py`: This provides methods for pickling (serializing) pandas objects. These are discussed as follows:
    - `to_pickle(..)`: This serializes object to a file.
    - `read_pickle(..)`: This reads serialized object from file into pandas object. It should only be used with trusted sources.
- `pytables.py`: This is an interface to `PyTables` module for reading and writing pandas data structures to files on disk.
- `sql.py`: It is a collection of classes and functions used to enable the retrieval of data from relational databases that attempts to be database agnostic. These are discussed as follows:
    - `PandasSQL`: This is the base class for interfacing pandas with SQL. It provides dummy `read_sql` and `to_sql` methods that must be implemented by subclasses.
    - `PandasSQLAlchemy`: This is the subclass of `PandasSQL` that enables conversions between DataFrame and SQL databases using `SQLAlchemy`.
    - `PandasSQLTable` class: This maps pandas tables (DataFrame) to SQL tables.
    - `pandasSQL_builder(..)`: This returns the correct `PandasSQL` subclass based on the provided parameters.
    - `PandasSQLTableLegacy` class: This is the legacy support version of `PandasSQLTable`.
    - `PandasSQLLegacy` class: This is the legacy support version of `PandasSQLTable`.
    - `get_schema(..)`: This gets the SQL database table schema for a given frame.
    - `read_sql_table(..)`: This reads SQL db table into a DataFrame.
    - `read_sql_query(..)`: This reads SQL query into a DataFrame.
    - `read_sql(..)`: This reads SQL query/table into a DataFrame.
- `to_sql(..)`: This write records that are stored in a DataFrame to a SQL database.
- `stata.py`: This contains tools for processing `Stata` files into pandas DataFrames.
- `wb.py`: This is the module for downloading data from World Bank's website.

pandas/tools

- `util.py`: This has miscellaneous `util` functions defined such as `match(..)`, `cartesian_product(..)`, and `compose(..)`.

- `tile.py`: This has a set of functions that enable quantization of input data and hence `tile` functionality. Most of the functions are internal, except for `cut(..)` and `qcut(..)`.

- `rplot.py`: This is the module that provides the functionality to generate trellis plots in pandas.

- `plotting.py`: This provides a set of plotting functions that take a Series or DataFrame as an argument.

    - `scatter_matrix(..)`: This draws a matrix of scatter plots

    - `andrews_curves(..)`: This plots multivariate data as curves that are created using samples as coefficients for a Fourier series

    - `parallel_coordinates(..)`: This is a plotting technique that allows you to see clusters in data and visually estimate statistics

    - `lag_plot(..)`: This is used to check whether a dataset or a time series is random

    - `autocorrelation_plot(..)`: This is used for checking randomness in a time series

    - `bootstrap_plot(..)`: This plot is used to determine the uncertainty of a statistical measure such as mean or median in a visual manner

    - `radviz(..)`: This plot is used to visualize multivariate data

    > Reference for the preceding information is from:
    > `http://pandas.pydata.org/pandas-docs/`
    > `stable/visualization.html`

- `pivot.py`: This function is for handling pivot tables in pandas. It is the main function `pandas.tools.pivot_table(..)` which creates a spreadsheet-like pivot table as a DataFrame

    > Reference for the preceding information is from:
    > `http://pandas.pydata.org/pandas-docs/`
    > `stable/reshaping.html`

- `merge.py`: This provides functions for combining the Series, DataFrame, and Panel objects such as `merge(..)` and `concat(..)`

- `describe.py`: This provides a single `value_range(..)` function that returns the maximum and minimum of a DataFrame as a Series.

pandas/sparse

This is the module that provides *sparse* implementations of Series, DataFrame, and Panel. By sparse, we mean arrays where values such as missing or NA are omitted rather than kept as *0*.

For more information on this, go to `http://pandas.pydata.org/pandas-docs/version/stable/sparse.html`.

- `api.py`: It is a set of convenience imports

- `array.py`: It is an implementation of the SparseArray data structure

- `frame.py`: It is an implementation of the SparseDataFrame data structure

- `list.py`: It is an implementation of the SparseList data structure

- `panel.py`: It is an implementation of the SparsePanel data structure

- `series.py`: It is an implementation of the SparseSeries data structure

pandas/stats

- `api.py`: This is a set of convenience imports.

- `common.py`: This defines internal functions called by other functions in a module.

- `fama_macbeth.py`: This contains class definitions and functions for the Fama-Macbeth regression. For more information on FM regression, go to `http://en.wikipedia.org/wiki/Fama-MacBeth_regression`.

- `interface.py`: It defines `ols(..)` which returns an **Ordinary Least Squares** (**OLS**) regression object. It imports from `pandas.stats.ols` module.

- `math.py`: This has useful functions defined as follows:

    ° `rank(..)`, `solve(..)`, and `inv(..)`: These are used for matrix rank, solution, and inverse respectively

    ° `is_psd(..)`: This checks positive-definiteness of matrix

    ° `newey_west(..)`: This is for covariance matrix computation

    ° `calc_F(..)`: This computes F-statistic

- `misc.py`: This is used for miscellaneous functions.
- `moments.py`: This provides rolling and expanding statistical measures including moments that are implemented in Cython. These methods include: `rolling_count(..)`, `rolling_cov(..)`, `rolling_corr(..)`, `rolling_corr_pairwise(..)`, `rolling_quantile(..)`, `rolling_apply(..)`, `rolling_window(..)`, `expanding_count(..)`, `expanding_quantile(..)`, `expanding_cov(..)`, `expanding_corr(..)`, `expanding_corr_pairwise(..)`, `expanding_apply(..)`, `ewma(..)`, `ewmvar(..)`, `ewmstd(..)`, `ewmcov(..)`, and `ewmcorr(..)`.
- `ols.py`: This implements OLS and provides the OLS and `MovingOLS` classes. OLS runs a full sample Ordinary Least-Squares Regression, whereas `MovingOLS` generates a rolling or an expanding simple OLS.
- `plm.py`: This provides linear regression objects for Panel data. These classes are discussed as follows:
    - `PanelOLS`: This is the OLS for Panel object
    - `MovingPanelOLS`: This is the rolling/expanded OLS for Panel object
    - `NonPooledPanelOLS`:- This is the nonpooled OLS for Panel object
- `var.py`: This provides vector auto-regression classes discussed as follows:
    - `VAR`: This is the vector auto-regression on multi-variate data in Series and DataFrames
    - `PanelVAR`: This is the vector auto-regression on multi-variate data in Panel objects

> For more information on vector autoregression, go to:
> `http://en.wikipedia.org/wiki/Vector_autoregression`

pandas/util

- `testing.py`: This provides the assertion, debug, unit test, and other classes/ functions for use in testing. It contains many special assert functions that make it easier to check whether Series, DataFrame, or Panel objects are equivalent. Some of these functions include `assert_equal(..)`, `assert_series_equal(..)`, `assert_frame_equal(..)`, and `assert_panelnd_equal(..)`. The `pandas.util.testing` module is especially useful to the contributors of the pandas code base. It defines a `util.TestCase` class. It also provides utilities for handling locales, console debugging, file cleanup, comparators, and so on for testing by potential code base contributors.

- `terminal.py`: This function is mostly internal and has to do with obtaining certain specific details about the terminal. The single exposed function is `get_terminal_size()`.

- `print_versions.py`: This defines the `get_sys_info()` function that returns a dictionary of systems information, and the `show_versions(..)` function that displays the versions of available Python libraries.

- `misc.py`: This defines a couple of miscellaneous utilities.

- `decorators.py`: This defines some decorator functions and classes.

> The Substitution and Appender classes are decorators that perform substitution and appending on function `docstrings` and for more information on Python decorators, go to `http://bit.ly/1zj8U0o`.

- `clipboard.py`: This contains cross-platform clipboard methods to enable the copy and paste functions from the keyboard. The pandas I/O API include functions such as `pandas.read_clipboard()` and `pandas.to_clipboard(..)`.

pandas/rpy

This module attempts to provide an interface to the R statistical package if it is installed in the machine. It is deprecated in Version 0.16.0 and later. It's functionality is replaced by the `rpy2` module that can be accessed from `http://rpy.sourceforge.net`.

- `base.py`: This defines a class for the well-known `lm` function in R
- `common.py`: This provides many functions to enable the conversion of pandas objects into their equivalent R versions
- `mass.py`: This is an unimplemented version of `rlm`—R's `lm` function
- `var.py`: This contains an unimplemented class `VAR`

pandas/tests

This is the module that provides many tests for various objects in pandas. The names of the specific library files are fairly self-explanatory, and I will not go into further details here, except inviting the reader to explore this.

pandas/compat

The functionality related to compatibility are explained as follows:

- `chainmap.py`, `chainmap_impl.py`: This provides a `ChainMap` class that can group multiple `dicts` or mappings, in order to produce a single view that can be updated

- `pickle_compat.py`: This provides functionality for pickling pandas objects in the versions that are earlier than 0.12

- `openpyxl_compat.py`: This checks the compatibility of `openpyxl`

pandas/computation

This is the module that provides functionality for computation and is discussed as follows:

- `api.py`: This contains imports for `eval` and `expr`.

- `align.py`: This implements functions for data alignment.

- `common.py`: This contains a couple of internal functions.

- `engines.py`: This defines Abstract Engine, `NumExprEngine`, and `PythonEngine`. `PythonEngine` evaluates an expression and is used mainly for testing purposes.

- `eval.py`: This defines the all-important `eval(..)` function and also a few other important functions.

- `expressions.py`: This provides fast expression evaluation through `numexpr`. The `numexpr` function is used to accelerate certain numerical operations. It uses multiple cores as well as smart chunking and caching speedups. It defines the `evaluate(..)` and `where(..)` methods.

- `ops.py`: This defines the operator classes used by `eval`. These are `Term`, `Constant`, `Op`, `BinOp`, `Div`, and `UnaryOp`.

- `pytables.py`: This provides a query interface for the `PyTables` query.

- `scope.py`: This is a module for scope operations. It defines a `Scope` class, which is an object to hold scope.

> For more information on `numexpr`, go to `https://code.google.com/p/numexpr/`. For information of the usage of this module, go to `http://pandas.pydata.org/pandas-docs/stable/computation.html`.

pandas/tseries

- `api.py`: This is a set of convenience imports
- `converter.py`: This defines a set of classes that are used to format and convert datetime-related objects. Upon import, pandas registers a set of unit converters with `matplotlib`.
    - ° This is done via the `register()` function explained as follows:

        ```
        In [1]: import matplotlib.units as munits

        In [2]: munits.registry

        Out[2]: {}


        In [3]: import pandas

        In [4]: munits.registry

        Out[4]:
        {pandas.tslib.Timestamp: <pandas.tseries.converter.
        DatetimeConverter instance at 0x7fbbc4db17e8>,

         pandas.tseries.period.Period: <pandas.tseries.converter.
        PeriodConverter instance at 0x7fbbc4dc25f0>,

         datetime.date: <pandas.tseries.converter.DatetimeConverter
        instance at 0x7fbbc4dc2fc8>,

         datetime.datetime: <pandas.tseries.converter.
        DatetimeConverter instance at 0x7fbbc4dc2a70>,

         datetime.time: <pandas.tseries.converter.TimeConverter
        instance at 0x7fbbc4d61e18>}
        ```

    - ° `Converter`: This class includes `TimeConverter`, `PeriodConverter`, and `DateTimeConverter`
    - ° `Formatters`: This class includes `TimeFormatter`, `PandasAutoDateFormatter`, and `TimeSeries_DateFormatter`
    - ° `Locators`: This class includes `PandasAutoDateLocator`, `MilliSecondLocator`, and `TimeSeries_DateLocator`

    > The `Formatter` and `Locator` classes are used for handling ticks in matplotlib plotting.

- `frequencies.py`: This defines the code for specifying frequencies—daily, weekly, quarterly, monthly, annual, and so on—of time series objects.

- `holiday.py`: This defines functions and classes for handling holidays — `Holiday`, `AbstractHolidayCalendar`, and `USFederalHolidayCalendar` are among the classes defined.
- `index.py`: This defines the `DateTimeIndex` class.
- `interval.py`: This defines the `Interval`, `PeriodInterval`, and `IntervalIndex` classes.
- `offsets.py`: This defines various classes including Offsets that deal with time-related periods. These are explained as follows:
    - `DateOffset`: This is an interface for classes that provide the time period functionality such as `Week`, `WeekOfMonth`, `LastWeekOfMonth`, `QuarterOffset`, `YearOffset`, `Easter`, `FY5253`, and `FY5253Quarter`.
    - `BusinessMixin`: This is the `mixin` class for business objects to provide functionality with time-related classes. This will be inherited by the `BusinessDay` class. The `BusinessDay` subclass is derived from `BusinessMixin` and `SingleConstructorOffset` and provides an offset in business days.
    - `MonthOffset`: This is the interface for classes that provide the functionality for month time periods such as `MonthEnd`, `MonthBegin`, `BusinessMonthEnd`, and `BusinessMonthBegin`.
    - `MonthEnd` and `MonthBegin`: This is the date offset of one month at the end or the beginning of a month.
    - `BusinessMonthEnd` and `BusinessMonthBegin`: This is the date offset of one month at the end or the beginning of a business day calendar.
    - `YearOffset`: This offset is subclassed by classes that provide year period functionality — `YearEnd`, `YearBegin`, `BYearEnd`, `BYearBegin`
    - `YearEnd` and `YearBegin`: This is the date offset of one year at the end or the beginning of a year.
    - `BYearEnd` and `BYearBegin`: This is the date offset of one year at the end or the beginning of a business day calendar.
    - `Week`: This provides the offset of 1 week.
    - `WeekDay`: This provides mapping from weekday (Tue) to day of week (=2).
    - `WeekOfMonth` and `LastWeekOfMonth`: This describes dates in a week of a month
    - `QuarterOffset`: This is subclassed by classes that provide quarterly period functionality — `QuarterEnd`, `QuarterrBegin`, `BQuarterEnd`, and `BQuarterBegin`.

- ° `QuarterEnd, QuarterrBegin, BQuarterEnd,` and `BQuarterBegin`: This is same as for `Year*` classes except that the period is quarter instead of year.

- ° `FY5253, FY5253Quarter`: These classes describe a 52-53 week fiscal year. This is also known as a 4-4-5 calendar. You can get more information on this at `http://en.wikipedia.org/wiki/4-4-5_ calendar`.

- ° `Easter`: This is the `DateOffset` for the Easter holiday.

- ° `Tick`: This is the base class for Time unit classes such as `Day`, `Hour`, `Minute, Second, Milli, Micro,` and `Nano`.

- `period.py`: This defines the `Period` and `PeriodIndex` classes for pandas `TimeSeries`.

- `plotting.py`: This defines various plotting functions such as `tsplot(..)`, which plots a Series.

- `resample.py`: This defines `TimeGrouper`, a custom `groupby` class for time-interval grouping.

- `timedeltas.py`: This defines the `to_timedelta(..)` method, which converts its argument into a `timedelta` object.

- `tools.py`: This defines utility functions such as `to_datetime(..)`, `parse_time_string(..), dateutil_parse(..),` and `format(..)`.

- `util.py`: This defines more utility functions as follows:

  - ° `isleapyear(..)`: This checks whether the year is a leap year

  - ° `pivot_annual(..)`: This groups a series by years, accounting for leap years

pandas/sandbox

This module handles the integration of pandas DataFrame into the PyQt framework. For more information on PyQt, go to

Improving performance using Python extensions

One of the gripes of Python and pandas users is that the ease of use and expressiveness of the language and module comes with a significant downside—the performance—especially when it comes to numeric computing.

According to the programming benchmarks site, Python is often slower than compiled languages, such as C/C++ for many algorithms or data structure operations. An example of this would be binary tree operations. In the following reference, Python3 ran 104x slower than the fastest C++ implementation of an n-body simulation calculation: `http://bit.ly/1dm4JqW`.

So, how can we solve this legitimate yet vexing problem? We can mitigate this slowness in Python while maintaining the things that we like about it—clarity and productivity—by writing the parts of our code that are performance sensitive. For example numeric processing, algorithms in C/C++ and having them called by our Python code by writing a Python extension module: `http://docs.python.org/2/extending/extending.html`

Python extension modules enable us to make calls out to user-defined C/C++ code or library functions from Python, thus enabling us to boost our code performance but still benefit from the ease of using Python.

To help us understand what a Python extension module is, consider what happens in Python when we import a module. An import statement *imports* a module, but what does this really mean? There are three possibilities, which are as follows:

- Some Python extension modules are linked to the interpreter when it is built.
- An import causes Python to load a `.pyc` file into memory. The `.pyc` files contain Python bytecode.For example to the following command:

```
In [3]: import pandas
            pandas.__file__
Out[3]: '/usr/lib/python2.7/site-packages/pandas/__init__.pyc'
```

- The import statement causes a Python extension module to be loaded into the memory. The `.so` (shared object) file is comprised of machine code. For example refer to the following command:

```
In [4]: import math
          math.__file__
Out[4]: '/usr/lib/python2.7/lib-dynload/math.so'
```

We will focus on the third possibility. Even though we are dealing with a binary-shared object compiled from C, we can import it as a Python module, and this shows the power of Python extensions—applications can import modules from Python machine code or machine code and the interface is the same. Cython and SWIG are the two most popular methods of writing extensions in C and C++. In writing an extension, we wrap up C/C++ machine code and turn it into Python extension modules that behave like pure Python code. In this brief discussion, we will only focus on Cython, as it was designed specifically for Python.

Cython is a superset of Python that was designed to significantly improve Python's performance by allowing us to call externally compiled code in C/C++ as well as declare types on variables.

The Cython command generates an optimized C/C++ source file from a Cython source file, and compiles this optimized C/C++ source into a Python extension module. It offers built-in support for NumPy and combines C's performance with Python's usability.

We will give a quick demonstration of how we can use Cython to significantly speed up our code. Let's define a simple Fibonacci function:

```
In [17]: def fibonacci(n):
         a,b=1,1
         for i in range(n):
             a,b=a+b,a
         return a
In [18]: fibonacci(100)
Out[18]: 927372692193078999176L
In [19]: %timeit fibonacci(100)
         100000 loops, best of 3: 18.2 µs per loop
```

Using the `timeit` module, we see that it takes 18.2 µs per loop.

Let's now rewrite the function in Cython, specifying types for the variables by using the following steps:

1. First, we import the Cython magic function to IPython as follows:

   ```
   In [22]: %load_ext cythonmagic
   ```

2. Next, we rewrite our function in Cython, specifying types for our variables:

   ```
   In [24]: %%cython
            def cfibonacci(int n):
                cdef int i, a,b
                for i in range(n):
                    a,b=a+b,a
                return a
   ```

3. Let's time our new Cython function:

```
In [25]: %timeit cfibonacci(100)
         1000000 loops, best of 3: 321 ns per loop


In [26]: 18.2/0.321
Out[26]: 56.69781931464174
```

4. Thus, we can see that the Cython version is 57x faster than the pure Python version!

For more references on writing Python extensions using Cython/SWIG or other options, please refer to the following references:

- The pandas documentation titled *Enhancing Performance* at `http://pandas.pydata.org/pandas-docs/stable/enhancingperf.html`
- Scipy Lecture Notes titled *Interfacing with C* at `https://scipy-lectures.github.io/advanced/interfacing_with_c/interfacing_with_c.html`
- Cython documentation at `http://docs.cython.org/index.html`
- SWIG Documentation at `http://www.swig.org/Doc2.0/SWIGDocumentation.html`

Summary

To summarize this chapter, we took a tour of the library hierarchy of pandas in an attempt to illustrate the internal workings of the library. We also touched on the benefits of speeding up our code performance by using a Python extension module.

10
R and pandas Compared

This chapter focuses on comparing pandas with R, the statistical package on which much of pandas' functionality is modeled. It is intended as a guide for R users who wish to use pandas, and for users who wish to replicate functionality that they have seen in the R code in pandas. It focuses on some key features available to R users and shows how to achieve similar functionality in pandas by using some illustrative examples. This chapter assumes that you have the R statistical package installed. If not, it can be downloaded and installed from here: `http://www.r-project.org/`.

By the end of the chapter, data analysis users should have a good grasp of the data analysis capabilities of R as compared to pandas, enabling them to transition to or use pandas, should they need to. The various topics addressed in this chapter include the following:

- R data types and their pandas equivalents
- Slicing and selection
- Arithmetic operations on datatype columns
- Aggregation and GroupBy
- Matching
- Split-apply-combine
- Melting and reshaping
- Factors and categorical data

R data types

R has five primitive or atomic types:

- Character
- Numeric

- Integer
- Complex
- Logical/Boolean

It also has the following, more complex, container types:

- **Vector**: This is similar to `numpy.array`. It can only contain objects of the same type.
- **List**: It is a heterogeneous container. Its equivalent in pandas would be a series.
- **DataFrame**: It is a heterogeneous 2D container, equivalent to a pandas DataFrame
- **Matrix**:- It is a homogeneous 2D version of a vector. It is similar to a `numpy.matrix`.

For this chapter, we will focus on list and DataFrame, which have pandas equivalents as series and DataFrame.

> For more information on R data types, refer to the following document at: `http://www.statmethods.net/input/datatypes.html`.
>
> For NumPy data types, refer to the following document at: `http://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html` and `http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html`.

R lists

R lists can be created explicitly as a list declaration as shown here:

```
>h_lst<- list(23,'donkey',5.6,1+4i,TRUE)
>h_lst
[[1]]
[1] 23


[[2]]
[1] "donkey"


[[3]]
[1] 5.6
```

```
[[4]]
[1] 1+4i


[[5]]
[1] TRUE


>typeof(h_lst)
[1] "list"
```

Here is its series equivalent in pandas with the creation of a list and the creation of a series from it:

```
In [8]: h_list=[23, 'donkey', 5.6,1+4j, True]
In [9]: import pandas as pd

        h_ser=pd.Series(h_list)
In [10]: h_ser
Out[10]: 0           23
         1       donkey
         2          5.6
         3        (1+4j)
         4         True
dtype: object
```

Array indexing starts from 0 in pandas as opposed to R, where it starts at 1. Following is an example of this:

```
In [11]: type(h_ser)
Out[11]: pandas.core.series.Series
```

R DataFrames

We can construct an R DataFrame as follows by calling the `data.frame()` constructor and then display it as follows:

```
>stocks_table<- data.frame(Symbol=c('GOOG','AMZN','FB','AAPL',
                                     'TWTR','NFLX','LINKD'),
                           Price=c(518.7,307.82,74.9,109.7,37.1,
                                   334.48,219.9),
MarketCap=c(352.8,142.29,216.98,643.55,23.54,20.15,27.31))
```

```
>stocks_table
Symbol  PriceMarketCap
1    GOOG 518.70     352.80
2    AMZN 307.82     142.29
3      FB  74.90     216.98
4    AAPL 109.70     643.55
5    TWTR  37.10      23.54
6    NFLX 334.48      20.15
7   LINKD 219.90      27.31
```

Here, we construct a pandas DataFrame and display it:

```
In [29]: stocks_df=pd.DataFrame({'Symbol':['GOOG','AMZN','FB','AAPL',
                                'TWTR','NFLX','LNKD'],
                          'Price':[518.7,307.82,74.9,109.7,37.1,
        334.48,219.9],
'MarketCap($B)' : [352.8,142.29,216.98,643.55,
                                          23.54,20.15,27.31]
                          })
stocks_df=stocks_df.reindex_axis(sorted(stocks_df.columns,reverse=True),a
xis=1)
stocks_df
Out[29]:
Symbol  PriceMarketCap($B)
0        GOOG    518.70  352.80
1        AMZN    307.82  142.29
2        FB       74.90  216.98
3        AAPL    109.70  643.55
4        TWTR     37.10  23.54
5        NFLX    334.48  20.15
6        LNKD219.90  27.31
```

Slicing and selection

In R, we slice objects in the following three ways:

- `[`: This always returns an object of the same type as the original and can be used to select more than one element.

- `[[`: This is used to extract elements of list or DataFrame; and can only be used to extract a single element,: the type of the returned element will not necessarily be a list or DataFrame.

- `$`: This is used to extract elements of a list or DataFrame by name and is similar to `[[`.

Here are some slicing examples in R and their equivalents in pandas:

R-matrix and NumPy array compared

Let's see matrix creation and selection in R:

```
>r_mat<- matrix(2:13,4,3)
>r_mat
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    3    7   11
[3,]    4    8   12
[4,]    5    9   13
```

To select first row, we write:

```
>r_mat[1,]
[1]  2  6 10
```

To select second column, we use the following command:

```
>r_mat[,2]
[1] 6 7 8 9
```

Let's now see NumPy array creation and selection:

```
In [60]: a=np.array(range(2,6))
         b=np.array(range(6,10))
         c=np.array(range(10,14))
In [66]: np_ar=np.column_stack([a,b,c])
np_ar
```

```
Out[66]: array([[ 2,  6, 10],
[ 3,  7, 11],
[ 4,  8, 12],
[ 5,  9, 13]])
```

To select first row, write the following command:

```
In [79]: np_ar[0,]
Out[79]: array([ 2,  6, 10])
```

> Indexing is different in R and pandas/NumPy.
>
> In R, indexing starts at 1, while in pandas/NumPy, it starts at 0. Hence, we have to subtract 1 from all indexes when making the translation from R to pandas/NumPy.

To select second column, write the following command:

```
In [81]: np_ar[:,1]
Out[81]: array([6, 7, 8, 9])
```

Another option is to transpose the array first and then select the column, as follows:

```
In [80]: np_ar.T[1,]
Out[80]: array([6, 7, 8, 9])
```

R lists and pandas series compared

Here is an example of list creation and selection in R:

```
>cal_lst<- list(weekdays=1:8, mth='jan')
>cal_lst
$weekdays
[1] 1 2 3 4 5 6 7 8

$mth
[1] "jan"

>cal_lst[1]
$weekdays
[1] 1 2 3 4 5 6 7 8
```

```
>cal_lst[[1]]
[1] 1 2 3 4 5 6 7 8


>cal_lst[2]
$mth
[1] "jan"
```

Series creation and selection in pandas is done as follows:

```
In [92]: cal_df= pd.Series({'weekdays':range(1,8), 'mth':'jan'})
In [93]: cal_df
Out[93]: mthjan
weekdays    [1, 2, 3, 4, 5, 6, 7]
dtype: object


In [97]: cal_df[0]
Out[97]: 'jan'


In [95]: cal_df[1]
Out[95]: [1, 2, 3, 4, 5, 6, 7]


In [96]: cal_df[[1]]
Out[96]: weekdays    [1, 2, 3, 4, 5, 6, 7]
dtype: object
```

Here, we see a difference between an R-list and a pandas series from the perspective of the `[]` and `[[]]` operators. We can see the difference by considering the second item, which is a character string.

In the case of R, the `[]` operator produces a container type, that is, a list containing the string, while the `[[]]` produces an atomic type: in this case, a character as follows:

```
>typeof(cal_lst[2])
[1] "list"
>typeof(cal_lst[[2]])
[1] "character"
```

In the case of pandas, the opposite is true: `[]` produces the atomic type, while `[[]]` results in a complex type, that is, a series as follows:

```
In [99]: type(cal_df[0])
Out[99]: str


In [101]: type(cal_df[[0]])
Out[101]: pandas.core.series.Series
```

In both R and pandas, the column name can be specified in order to obtain an element.

Specifying column name in R

In R, this can be done with the column name preceded by the `$` operator as follows:

```
>cal_lst$mth
[1] "jan"
> cal_lst$'mth'
[1] "jan"
```

Specifying column name in pandas

In pandas, we subset elements in the usual way with the column name in square brackets:

```
In [111]: cal_df['mth']
Out[111]: 'jan'
```

One area where R and pandas differ is in the subsetting of nested elements. For example, to obtain day 4 from weekdays, we have to use the `[[]]` operator in R:

```
>cal_lst[[1]][[4]]
[1] 4


>cal_lst[[c(1,4)]]
[1] 4
```

However, in the case of pandas, we can just use a double `[]`:

```
In [132]: cal_df[1][3]
Out[132]: 4
```

R's DataFrames versus pandas' DataFrames

Selecting data in R DataFrames and pandas DataFrames follows a similar script. The following section explains on how we perform multi-column selects from both.

Multicolumn selection in R

In R, we specify the multiple columns to select by stating them in a vector within square brackets:

```
>stocks_table[c('Symbol','Price')]
Symbol  Price
1    GOOG 518.70
2    AMZN 307.82
3      FB  74.90
4    AAPL 109.70
5    TWTR  37.10
6    NFLX 334.48
7   LINKD 219.90


>stocks_table[,c('Symbol','Price')]
Symbol  Price
1    GOOG 518.70
2    AMZN 307.82
3      FB  74.90
4    AAPL 109.70
5    TWTR  37.10
6    NFLX 334.48
7   LINKD 219.90
```

Multicolumn selection in pandas

In pandas, we subset elements in the usual way with the column names in square brackets:

```
In [140]: stocks_df[['Symbol','Price']]
Out[140]:Symbol Price
0       GOOG   518.70
1       AMZN   307.82
2       FB     74.90
```

```
3        AAPL    109.70
4        TWTR    37.10
5        NFLX    334.48
6        LNKD    219.90


In [145]: stocks_df.loc[:,['Symbol','Price']]
Out[145]: Symbol  Price
0         GOOG    518.70
1         AMZN    307.82
2         FB      74.90
3         AAPL    109.70
4         TWTR    37.10
5         NFLX    334.48
6         LNKD    219.90
```

Arithmetic operations on columns

In R and pandas, we can apply arithmetic operations in data columns in a similar manner. Hence, we can perform arithmetic operations such as addition or subtraction on elements in corresponding positions in two or more DataFrames.

Here, we construct a DataFrame in R with columns labeled x and y, and subtract column y from column x:

```
>norm_df<- data.frame(x=rnorm(7,0,1), y=rnorm(7,0,1))
>norm_df$x - norm_df$y
[1] -1.3870730  2.4681458 -4.6991395  0.2978311 -0.8492245  1.5851009
-1.4620324
```

The `with` operator in R also has the same effect as arithmetic operations:

```
>with(norm_df,x-y)
[1] -1.3870730  2.4681458 -4.6991395  0.2978311 -0.8492245  1.5851009
-1.4620324
```

In pandas, the same arithmetic operations on columns can be done and the equivalent operator is `eval`:

```
In [10]: import pandas as pd
         import numpy as np
```

```
df = pd.DataFrame({'x': np.random.normal(0,1,size=7), 'y': np.random.
normal(0,1,size=7)})


In [11]: df.x-df.y
Out[11]: 0    -0.107313
         1     0.617513
         2    -1.517827
         3     0.565804
         4    -1.630534
         5     0.101900
         6     0.775186
dtype: float64


In [12]: df.eval('x-y')
Out[12]: 0    -0.107313
         1     0.617513
         2    -1.517827
         3     0.565804
         4    -1.630534
         5     0.101900
         6     0.775186
dtype: float64
```

Aggregation and GroupBy

Sometimes, we may wish to split data into subsets and apply a function such as the mean, max, or min to each subset. In R, we can do this via the `aggregate` or `tapply` functions.

Here, we will use the example of a dataset of statistics on the top five strikers of the four clubs that made it to the semi-final of the European Champions League Football tournament in 2014. We will use it to illustrate aggregation in R and its equivalent GroupBy functionality in pandas.

Aggregation in R

In R aggregation is done using the following command:

```
> goal_stats=read.csv('champ_league_stats_semifinalists.csv')
>goal_stats
```

	Club	Player	Goals	GamesPlayed
1	Atletico Madrid	Diego Costa	8	9
2	Atletico Madrid	ArdaTuran	4	9
3	Atletico Madrid	RaúlGarcía	4	12
4	Atletico Madrid	AdriánLópez	2	9
5	Atletico Madrid	Diego Godín	2	10
6	Real Madrid	Cristiano Ronaldo	17	11
7	Real Madrid	Gareth Bale	6	12
8	Real Madrid	Karim Benzema	5	11
9	Real Madrid	Isco	3	12
10	Real Madrid	Ángel Di María	3	11
11	Bayern Munich	Thomas Müller	5	12
12	Bayern Munich	ArjenRobben	4	10
13	Bayern Munich	Mario Götze	3	11
14	Bayern Munich	Bastian Schweinsteiger	3	8
15	Bayern Munich	Mario Mandžuki	3	10
16	Chelsea	Fernando Torres	4	9
17	Chelsea	Demba Ba	3	6
18	Chelsea	Samuel Eto'o	3	9
19	Chelsea	Eden Hazard	2	9
20	Chelsea	Ramires	2	10

We can now compute the goals per game ratio for each striker, to measure their deadliness in front of a goal:

```
>goal_stats$GoalsPerGame<- goal_stats$Goals/goal_stats$GamesPlayed
>goal_stats
```

	Club	Player	Goals	GamesPlayed	GoalsPerGame
1	Atletico Madrid	Diego Costa	8	9	0.8888889
2	Atletico Madrid	ArdaTuran	4	9	0.4444444
3	Atletico Madrid	RaúlGarcía	4	12	0.3333333
4	Atletico Madrid	AdriánLópez	2	9	0.2222222

5	Atletico Madrid	Diego Godín	2	10	0.2000000
6	Real Madrid	Cristiano Ronaldo	17	11	1.5454545
7	Real Madrid	Gareth Bale	6	12	0.5000000
8	Real Madrid	Karim Benzema	5	11	0.4545455
9	Real Madrid	Isco	3	12	0.2500000
10	Real Madrid	Ángel Di María	3	11	0.2727273
11	Bayern Munich	Thomas Müller	5	12	0.4166667
12	Bayern Munich	ArjenRobben	4	10	0.4000000
13	Bayern Munich	MarioGötze	3	11	0.2727273
14	Bayern Munich	Bastian Schweinsteiger	3	8	0.3750000
15	Bayern Munich	MarioMandžuki	3	10	0.3000000
16	Chelsea	Fernando Torres	4	9	0.4444444
17	Chelsea	Demba Ba	3	6	0.5000000
18	Chelsea	Samuel Eto'o	3	9	0.3333333
19	Chelsea	Eden Hazard	2	9	0.2222222
20	Chelsea	Ramires	2	10	0.2000000

Let's suppose that we wanted to know the highest goals per game ratio for each team. We would calculate this as follows:

```
>aggregate(x=goal_stats[,c('GoalsPerGame')], by=list(goal_
stats$Club),FUN=max)
          Group.1           x
1 Atletico Madrid 0.8888889
2   Bayern Munich 0.4166667
3         Chelsea 0.5000000
4     Real Madrid 1.5454545
```

The `tapply` function is used to apply a function to a subset of an array or vector that is defined by one or more columns. The `tapply` function can also be used as follows:

```
>tapply(goal_stats$GoalsPerGame,goal_stats$Club,max)
Atletico Madrid   Bayern Munich        Chelsea     Real Madrid
      0.8888889       0.4166667      0.5000000       1.5454545
```

The pandas' GroupBy operator

In pandas, we can achieve the same result by using the `GroupBy` function:

```
In [6]: import pandas as pd
importnumpy as np
In [7]: goal_stats_df=pd.read_csv('champ_league_stats_semifinalists.csv')


In [27]: goal_stats_df['GoalsPerGame']=    goal_stats_df['Goals']/goal_
stats_df['GamesPlayed']


In [27]: goal_stats_df['GoalsPerGame']= goal_stats_df['Goals']/goal_
stats_df['GamesPlayed']


In [28]: goal_stats_df
```

Out[28]:	Club	Player	Goals	GamesPlayed	GoalsPerGame
0	Atletico Madrid	Diego Costa	8	9	0.888889
1	Atletico Madrid	ArdaTuran	4	9	0.444444
2	Atletico Madrid	RaúlGarcía	4	12	0.333333
3	Atletico Madrid	AdriánLópez	2	9	0.222222
4	Atletico Madrid	Diego Godín	2	10	0.200000
5	Real Madrid	Cristiano Ronaldo	17	11	1.545455
6	Real Madrid	Gareth Bale	6	12	0.500000
7	Real Madrid	Karim Benzema	5	11	0.454545
8	Real Madrid	Isco	3	12	0.250000
9	Real Madrid	Ángel Di María	3	11	0.272727
10	Bayern Munich	Thomas Müller	5	12	0.416667
11	Bayern Munich	ArjenRobben	4	10	0.400000
12	Bayern Munich	Mario Götze	3	11	0.272727
13	Bayern Munich	BastianSchweinsteiger	3	8	0.375000
14	Bayern Munich	MarioMandžuki	3	10	0.300000
15	Chelsea	Fernando Torres	4	9	0.444444
16	Chelsea	Demba Ba	3	6	0.500000
17	Chelsea	Samuel Eto'o	3	9	0.333333
18	Chelsea	Eden Hazard	2	9	0.222222
19	Chelsea	Ramires	2	10	0.200000

```
In [30]: grouped = goal_stats_df.groupby('Club')


In [17]: grouped['GoalsPerGame'].aggregate(np.max)
Out[17]: Club
         Atletico Madrid    0.888889
         Bayern Munich      0.416667
         Chelsea            0.500000
         Real Madrid        1.545455
         Name: GoalsPerGame, dtype: float64


In [22]: grouped['GoalsPerGame'].apply(np.max)

Out[22]: Club
         Atletico Madrid    0.888889
         Bayern Munich      0.416667
         Chelsea            0.500000
         Real Madrid        1.545455
         Name: GoalsPerGame, dtype: float64
```

Comparing matching operators in R and pandas

Here, we will demonstrate the equivalence of matching operators between R (`%in%`) and pandas (`isin()`). In both cases, a logical vector or series (pandas) is produced, which indicates the position at which a match was found.

R %in% operator

Here, we will demonstrate the use of the `%in%` operator in R:

```
>stock_symbols=stocks_table$Symbol
>stock_symbols
[1] GOOG  AMZN  FB  AAPL  TWTR  NFLX  LINKD
Levels: AAPL AMZN FB GOOG LINKD NFLX TWTR


>stock_symbols %in% c('GOOG','NFLX')
[1]  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
```

The pandas isin() function

Here is an example of using the pandas `isin()` function:

```
In [11]: stock_symbols=stocks_df.Symbol
stock_symbols
Out[11]: 0    GOOG
         1    AMZN
         2      FB
         3    AAPL
         4    TWTR
         5    NFLX
         6    LNKD
         Name: Symbol, dtype: object
In [10]: stock_symbols.isin(['GOOG','NFLX'])
Out[10]: 0     True
         1    False
         2    False
         3    False
         4    False
         5     True
         6    False
         Name: Symbol, dtype: bool
```

Logical subsetting

In R as well as in pandas, there is more than one way to perform logical subsetting. Suppose that we wished to display all players with the average goals per game ratio of greater than or equal to 0.5; that is, they average at least one goal every two games.

Logical subsetting in R

Here's how we can do this in R:

- Via a logical slice:

```
>goal_stats[goal_stats$GoalsPerGame>=0.5,]
    Club            Player          Goals GamesPlayedGoalsPerGame
  1 Atletico Madrid Diego Costa      8          9      0.8888889
  6 Real Madrid Cristiano Ronaldo   17         11      1.5454545
```

```
7   Real Madrid          Gareth Bale    6           12      0.5000000
17  Chelsea              Demba Ba       3           6       0.5000000
```

- Via the `subset()` function:

```
>subset(goal_stats,GoalsPerGame>=0.5)
    Club              Player        Goals GamesPlayedGoalsPerGame
1   Atletico Madrid Diego Costa      8           9       0.8888889
6   Real Madrid Cristiano Ronaldo 17           11      1.5454545
7   Real Madrid        Gareth Bale   6           12      0.5000000
17  Chelsea            Demba Ba      3           6       0.5000000
```

Logical subsetting in pandas

In pandas, we also do something similar:

- Logical slicing:

```
In [33]: goal_stats_df[goal_stats_df['GoalsPerGame']>=0.5]
Out[33]:       Club          Player          Goals
GamesPlayedGoalsPerGame
0     Atletico Madrid Diego Costa      8     9            0.888889
5     Real Madrid    Cristiano Ronaldo 17    11           1.545455
6     Real Madrid      Gareth Bale      6     12           0.500000
16    Chelsea          Demba Ba         3     6            0.500000
```

- `DataFrame.query()` operator:

```
In [36]:  goal_stats_df.query('GoalsPerGame>= 0.5')
Out[36]:
Club               Player    Goals GamesPlayedGoalsPerGame
0     Atletico Madrid Diego Costa   8     9            0.888889
5     Real Madrid  Cristiano Ronaldo 17   11           1.545455
6     Real Madrid      Gareth Bale   6     12           0.500000
16    Chelsea          Demba Ba      3     6            0.500000
```

Split-apply-combine

R has a library called `plyr` for a split-apply-combine data analysis. The `plyr` library has a function called `ddply`, which can be used to apply a function to a subset of a DataFrame, and then, combine the results into another DataFrame.

> For more information on ddply, you can refer to the following:
> `http://www.inside-r.org/packages/cran/plyr/docs/ddply`

To illustrate, let us consider a subset of a recently created dataset in R, which contains data on flights departing NYC in 2013: `http://cran.r-project.org/web/packages/nycflights13/index.html`.

Implementation in R

Here, we will install the package in R and instantiate the library:

```
>install.packages('nycflights13')
...

>library('nycflights13')
>dim(flights)
[1] 336776    16

>head(flights,3)
year month day dep_timedep_delayarr_timearr_delay carrier tailnum flight
1 2013    1   1     517         2     830         11       UA   N14228
1545
2 2013    1   1     533         4     850         20       UA   N24211
1714
3 2013    1   1     542         2     923         33       AA   N619AA
1141
origindestair_time distance hour minute
1    EWR  IAH     227     1400    5     17
2    LGA  IAH     227     1416    5     33
3    JFK  MIA     160     1089    5     42

> flights.data=na.omit(flights[,c('year','month','dep_delay','arr_
delay','distance')])
>flights.sample<- flights.data[sample(1:nrow(flights.
data),100,replace=FALSE),]
```

```
>head(flights.sample,5)
year month dep_delayarr_delay distance
155501 2013      3           2         5       184
2410   2013      1           0         4       762
64158  2013     11          -7       -27       509
221447 2013      5          -5       -12       184
281887 2013      8          -1       -10       937
```

The `ddply` function enables us to summarize the departure delays (mean, standard deviation) by year and month:

```
>ddply(flights.sample,.(year,month),summarize, mean_dep_
delay=round(mean(dep_delay),2), s_dep_delay=round(sd(dep_delay),2))
year month mean_dep_delaysd_dep_delay
1   2013     1          -0.20          2.28
2   2013     2          23.85         61.63
3   2013     3          10.00         34.72
4   2013     4           0.88         12.56
5   2013     5           8.56         32.42
6   2013     6          58.14        145.78
7   2013     7          25.29         58.88
8   2013     8          25.86         59.38
9   2013     9          -0.38         10.25
10  2013    10           9.31         15.27
11  2013    11          -1.09          7.73
12  2013    12           0.00          8.58
```

Let us save the `flights.sample` dataset to a CSV file so that we can use the data to show us how to do the same thing in pandas:

```
>write.csv(flights.sample,file='nycflights13_sample.csv',
quote=FALSE,row.names=FALSE)
```

Implementation in pandas

In order to do the same thing in pandas, we read the CSV file saved in the preceding section:

```
In [40]: flights_sample=pd.read_csv('nycflights13_sample.csv')


In [41]: flights_sample.head()
```

```
Out[41]: year    month   dep_delayarr_delay        distance
0        2013    3       2         5         184
1        2013    1       0         4         762
2        2013    11      -7        -27       509
3        2013    5       -5        -12       184
4        2013    8       -1        -10       937
```

We achieve the same effect as `ddply` by making use of the `GroupBy()` operator:

```
In [44]: pd.set_option('precision',3)
In [45]: grouped = flights_sample_df.groupby(['year','month'])

In [48]: grouped['dep_delay'].agg([np.mean, np.std])

Out[48]:        mean    std
year    month
2013    1       -0.20   2.28
        2       23.85   61.63
        3       10.00   34.72
        4       0.88    12.56
        5       8.56    32.42
        6       58.14   145.78
        7       25.29   58.88
        8       25.86   59.38
        9       -0.38   10.25
        10      9.31    15.27
        11      -1.09   7.73
        12      0.00    8.58
```

Reshaping using melt

The `melt` function converts data into a wide format to a single column consisting of unique ID-variable combinations.

The R melt() function

Here, we demonstrate the use of the `melt()` function in R. It produces long-format data in which the rows are unique variable-value combinations:

```
>sample4=head(flights.sample,4)[c('year','month','dep_delay','arr_
delay')]
> sample4
        year month dep_delay arr_delay
  155501 2013     3         2         5
  2410   2013     1         0         4
  64158  2013    11        -7       -27
  221447 2013     5        -5       -12


>melt(sample4,id=c('year','month'))
      year month  variable value
    1 2013     3 dep_delay     2
    2 2013     1 dep_delay     0
    3 2013    11 dep_delay    -7
    4 2013     5 dep_delay    -5
    5 2013     3 arr_delay     5
    6 2013     1 arr_delay     4
    7 2013    11 arr_delay   -27
    8 2013     5 arr_delay   -12
>
```

For more information, you can refer to the following: `http://www.statmethods.net/management/reshape.html`.

The pandas melt() function

In pandas, the `melt` function is similar:

```
In [55]: sample_4_df=flights_sample_df[['year','month','dep_delay', \
'arr_delay']].head(4)
In [56]: sample_4_df
Out[56]:    year   month dep_delay arr_delay
        0   2013   3        2         5
        1   2013   1        0         4
```

```
        2    2013    11      -7       -27
        3    2013    5       -5       -12


In [59]: pd.melt(sample_4_df,id_vars=['year','month'])
Out[59]: year    month    variable        value
        0    2013    3       dep_delay       2
        1    2013    1       dep_delay       0
        2    2013    11      dep_delay       -7
        3    2013    5       dep_delay       -5
        4    2013    3       arr_delay       5
        5    2013    1       arr_delay       4
        6    2013    11      arr_delay       -27
        7    2013    5       arr_delay       -12
```

The reference for this information is from: `http://pandas.pydata.org/pandas-docs/stable/reshaping.html#reshaping-by-melt`.

Factors/categorical data

R refers to categorical variables as factors, and the `cut()` function enables us to break a continuous numerical variable into ranges, and treat the ranges as factors or categorical variables, or to classify a categorical variable into a larger bin.

An R example using cut()

Here is an example in R:

```
clinical.trial<- data.frame(patient = 1:1000,
age = rnorm(1000, mean = 50, sd = 5),
year.enroll = sample(paste("19", 80:99, sep = ""),
                                1000, replace = TRUE))


>clinical.trial<- data.frame(patient = 1:1000,
+                                age = rnorm(1000, mean = 50, sd = 5),
+                                year.enroll = sample(paste("19", 80:99,
sep = ""),
+                                1000, replace = TRUE))
>summary(clinical.trial)
patient                age            year.enroll
```

```
 Min.   :    1.0   Min.   :31.14   1995   : 61
 1st Qu.: 250.8    1st Qu.:46.77   1989   : 60
Median : 500.5     Median :50.14   1985   : 57
 Mean   : 500.5    Mean   :50.14   1988   : 57
 3rd Qu.: 750.2    3rd Qu.:53.50   1990   : 56
 Max.   :1000.0    Max.   :70.15   1991   : 55
                                   (Other):654
>ctcut<- cut(clinical.trial$age, breaks = 5)> table(ctcut)
ctcut
(31.1,38.9] (38.9,46.7] (46.7,54.6] (54.6,62.4] (62.4,70.2]
        15          232         558         186           9
```

The reference for the preceding data can be found at: `http://www.r-bloggers.com/r-function-of-the-day-cut/`.

The pandas solution

Here is the equivalent of the earlier explained `cut()` function in pandas (only applies to Version 0.15+):

```
In [79]: pd.set_option('precision',4)
clinical_trial=pd.DataFrame({'patient':range(1,1001),
                                        'age' : np.random.
normal(50,5,size=1000),
                'year_enroll': [str(x) for x in np.random.choice(range(1
980,2000),size=1000,replace=True)]})


In [80]: clinical_trial.describe()
Out[80]:         age         patient
count   1000.000  1000.000
mean    50.089     500.500
std     4.909      288.819
min     29.944     1.000
        25%     46.572     250.750
        50%     50.314     500.500
        75%     53.320     750.250
max     63.458     1000.000
```

```
In [81]: clinical_trial.describe(include=['O'])
Out[81]:        year_enroll
count    1000
unique   20
top      1992
freq     62


In [82]: clinical_trial.year_enroll.value_counts()[:6]
Out[82]: 1992     62
         1985     61
         1986     59
         1994     59
         1983     58
         1991     58
dtype: int64
In [83]: ctcut=pd.cut(clinical_trial['age'], 5)
In [84]: ctcut.head()
Out[84]: 0     (43.349, 50.052]
         1     (50.052, 56.755]
         2     (50.052, 56.755]
         3     (43.349, 50.052]
         4     (50.052, 56.755]
         Name: age, dtype: category
         Categories (5, object): [(29.91, 36.646] < (36.646, 43.349] <
(43.349, 50.052] < (50.052, 56.755] < (56.755, 63.458]]


In [85]: ctcut.value_counts().sort_index()
Out[85]: (29.91, 36.646]       3
         (36.646, 43.349]      82
     (43.349, 50.052]      396
     (50.052, 56.755]      434
   (56.755, 63.458]      85
dtype: int64
```

Summary

In this chapter, we have attempted to compare key features in R with their pandas equivalents in order to achieve the following objectives:

- To assist R users who may wish to replicate the same functionality in pandas
- To assist any users who upon reading some R code may wish to rewrite the code in pandas

In the next chapter, we will conclude the book by giving a brief introduction to the `scikit-learn` library for doing machine learning and show how pandas fits within that framework. The reference documentation for this chapter can be found here: `http://pandas.pydata.org/pandas-docs/stable/comparison_with_r.html`.

11
Brief Tour of Machine Learning

This chapter takes the user on a whirlwind tour of machine learning, focusing on using the pandas library as a tool that can be used to preprocess data used by machine learning programs. It also introduces the user to the `scikit-learn` library, which is the most popular machine learning toolkit in Python.

In this chapter, we illustrate machine learning techniques by applying them to a well-known problem about classifying which passengers survived the Titanic disaster at the turn of the last century. The various topics addressed in this chapter include the following:

- Role of pandas in machine learning
- Installation of `scikit-learn`
- Introduction to machine learning concepts
- Application of machine learning – Kaggle Titanic competition
- Data analysis and preprocessing using pandas
- Naïve approach to Titanic problem
- `scikit-learn` ML classifier interface
- Supervised learning algorithms
- Unsupervised learning algorithms

Role of pandas in machine learning

The library we will be considering for machine learning is called `scikit-learn`. The `scikit-learn` Python library provides an extensive library of machine learning algorithms that can be used to create adaptive programs that learn from data inputs.

However, before this data can be used by `scikit-learn`, it must undergo some preprocessing. This is where pandas comes in. The pandas can be used to preprocess and filter data before passing it to the algorithm implemented in `scikit-learn`.

Installation of scikit-learn

As was mentioned in *Chapter 2*, *Installation of pandas and the Supporting Software*, the easiest way to install pandas and its accompanying libraries is to use a third-party distribution such as Anaconda and be done with it. Installing `scikit-learn` should be no different. I will briefly highlight the steps for installation on various platforms and third-party distributions starting with Anaconda. The `scikit-learn` library requires the following libraries:

- Python 2.6.x or higher
- NumPy 1.6.1 or higher
- SciPy 0.9 or higher

Assuming that you have already installed pandas as described in *Chapter 2*, *Installation of pandas and the Supporting Software*, these dependencies should already be in place.

Installing via Anaconda

You can install `scikit-learn` on Anaconda by running the conda Python package manager:

```
conda install scikit-learn
```

Installing on Unix (Linux/Mac OS X)

For Unix, it is best to install from the source (C compiler is required). Assuming that pandas and NumPy are already installed and the required dependent libraries are already in place, you can install `scikit-learn` via Git by running the following commands:

```
git clone https://github.com/scikit-learn/scikit-learn.git
cd scikitlearn
python setup.py install
```

The pandas can also be installed on Unix by using `pip` from `PyPi`:

```
pip install pandas
```

Installing on Windows

To install on Windows, you can open a console and run the following:

```
pip install -U scikit-learn
```

> For more in-depth information on installation, you can take a look at the official `scikit-learn` docs at: `http://scikit-learn.org/stable/install.html`.
>
> You can also take a look at the README file for the `scikit-learn` Git repository at: `https://github.com/scikit-learn/scikit-learn/blob/master/README.rst`.

Introduction to machine learning

Machine learning is the art of creating software programs that learn from data. More formally, it can be defined as the practice of building adaptive programs that use tunable parameters to improve predictive performance. It is a sub-field of artificial intelligence.

We can separate machine learning programs based on the type of problems they are trying to solve. These problems are appropriately called learning problems.

The two categories of these problems, broadly speaking, are referred to as supervised and unsupervised learning problems. Further, there are some hybrid problems that have aspects that involve both categories.

The input to a learning problem consists of a dataset of n rows. Each row represents a sample and may involve one or more fields referred to as attributes or features.

A dataset can be canonically described as consisting of n samples, each consisting of m features. A more detailed introduction to machine learning is given in the following paper:

A Few Useful Things to Know about Machine Learning at `http://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf`

Supervised versus unsupervised learning

For supervised learning problems, the input to a learning problem is a dataset consisting of *labeled* data. By this we mean that we have outputs whose values are known. The learning program is fed input samples and their corresponding outputs and its goal is to decipher the relationship between them. Such input is known as labeled data. Supervised learning problems include the following:

- **Classification**: The learned attribute is categorical (nominal) or discrete
- **Regression**: The learned attribute is numeric/continuous

In unsupervised learning or data mining, the learning program is fed inputs but no corresponding outputs. This input data is referred to as unlabeled data. The learning program's goal is to learn or decipher the hidden label. Such problems include the following:

- Clustering
- Dimensionality reduction

Illustration using document classification

A common usage of machine learning techniques is in the area of document classification. The two main categories of machine learning can be applied to this problem - supervised and unsupervised learning.

Supervised learning

Each document in the input collection is assigned to a category, that is, a label. The learning program/algorithm uses the input collection of documents to learn how to make predictions for another set of documents with no labels. This method is known as *classification*.

Unsupervised learning

The documents in the input collection are not assigned to categories; hence, they are unlabeled. The learning program takes this as input and tries to *cluster* or discover groups of related or similar documents. This method is known as *clustering*.

How machine learning systems learn

Machine learning systems utilize what is known as a classifier in order to learn from data. A *classifier* is an interface that takes a matrix of what is known as *feature values* and produces an output vector, also known as the class. These feature values may be discrete or continuously valued. There are three core components of classifiers:

- **Representation**: What type of classifier is it?
- **Evaluation**: How good is the classifier?
- **Optimization**: How to search among the alternatives?

Application of machine learning – Kaggle Titanic competition

In order to illustrate how we can use pandas to assist us at the start of our machine learning journey, we will apply it to a classic problem, which is hosted on the Kaggle website (`http://www.kaggle.com`). **Kaggle** is a competition platform for machine learning problems. The idea behind Kaggle is to enable companies that are interested in solving predictive analytics problems with their data to post their data on Kaggle and invite data scientists to come up with the proposed solutions to their problems. The competition can be ongoing over a period of time, and the rankings of the competitors are posted on a leader board. At the end of the competition, the top-ranked competitors receive cash prizes.

The classic problem that we will study in order to illustrate the use of pandas for machine learning with `scikit-learn` is the Titanic: machine learning from disaster problem hosted on Kaggle as their classic introductory machine learning problem. The dataset involved in the problem is a raw dataset. Hence, pandas is very useful in the preprocessing and cleansing of the data before it is submitted as input to the machine learning algorithm implemented in `scikit-learn`.

The titanic: machine learning from disaster problem

The dataset for the Titanic consists of the passenger manifest for the doomed trip, along with various features and an indicator variable telling whether the passenger survived the sinking of the ship or not. The essence of the problem is to be able to predict, given a passenger and his/her associated features, whether this passenger survived the sinking of the Titanic or not. Please delete this sentence.

The data consists of two datasets: one training dataset and the other test dataset. The training dataset consists of 891 passenger cases, and the test dataset consists of 491 passenger cases.

The training dataset also consists of 11 variables, of which 10 are features and 1 dependent/indicator variable Survived, which indicates whether the passenger survived the disaster or not.

The feature variables are as follows:

- PassengerID
- Cabin
- Sex
- Pclass (passenger class)
- Fare
- Parch (number of parents and children)
- Age
- Sibsp (number of siblings)
- Embarked

We can make use of pandas to help us preprocess data in the following ways:

- Data cleaning and categorization of some variables
- Exclusion of unnecessary features, which obviously have no bearing on the survivability of the passenger, for example, their name
- Handling missing data

There are various algorithms that we can use to tackle this problem. They are as follows:

- Decision trees
- Neural networks
- Random forests
- Support vector machines

The problem of overfitting

Overfitting is a well-known problem in machine learning, whereby the program memorizes the specific data that it is fed as input, leading to perfect results on the training data and abysmal results on the test data.

In order to prevent overfitting, the 10-fold cross-validation technique can be used to introduce variability in the data during the training phase.

Data analysis and preprocessing using pandas

In this section, we will utilize pandas to do some analysis and preprocessing of the data before submitting it as input to `scikit-learn`.

Examining the data

In order to start our preprocessing of the data, let us read in the training dataset and examine what it looks like.

Here, we read in the training dataset into a pandas DataFrame and display the first rows:

```
In [2]: import pandas as pd
        import numpy as np
# For .read_csv, always use header=0 when you know row 0 is the header
row
        train_df = pd.read_csv('csv/train.csv', header=0)
In [3]: train_df.head(3)
```

The output is as follows:

```
In [3]: from matplotlib import pyplot as plt
        import matplotlib.pyplot as plt
        import pandas as pd
        import numpy as np
        import patsy as pt

In [4]: # For .read_csv, always use header=0 when you know row 0 is the header row
        train_df = pd.read_csv('csv/train.csv', header=0)
        test_df = pd.read_csv('csv/test.csv', header=0)

In [5]: train_df.head(3)
```

Out[5]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.9250	NaN	S

Thus, we can see the various features: **PassengerId**, **PClass**, **Name**, **Sex**, **Age**, **Sibsp**, **Parch**, **Ticket**, **Fare**, **Cabin**, and **Embarked**. One question that springs to mind immediately is this: which of the features are likely to influence whether a passenger survived or not?

It should seem obvious that PassengerID, Ticket Code, and Name should not be influencers on survivability since they're *identifier* variables. We will skip these in our analysis.

Handling missing values

One issue that we have to deal with in datasets for machine learning is how to handle missing values in the training set.

Let's visually identify where we have missing values in our feature set.

For that, we can make use of an equivalent of the `missmap` function in R, written by Tom Augspurger. The next graphic shows how much data is missing for the various features in an intuitively pleasing manner:
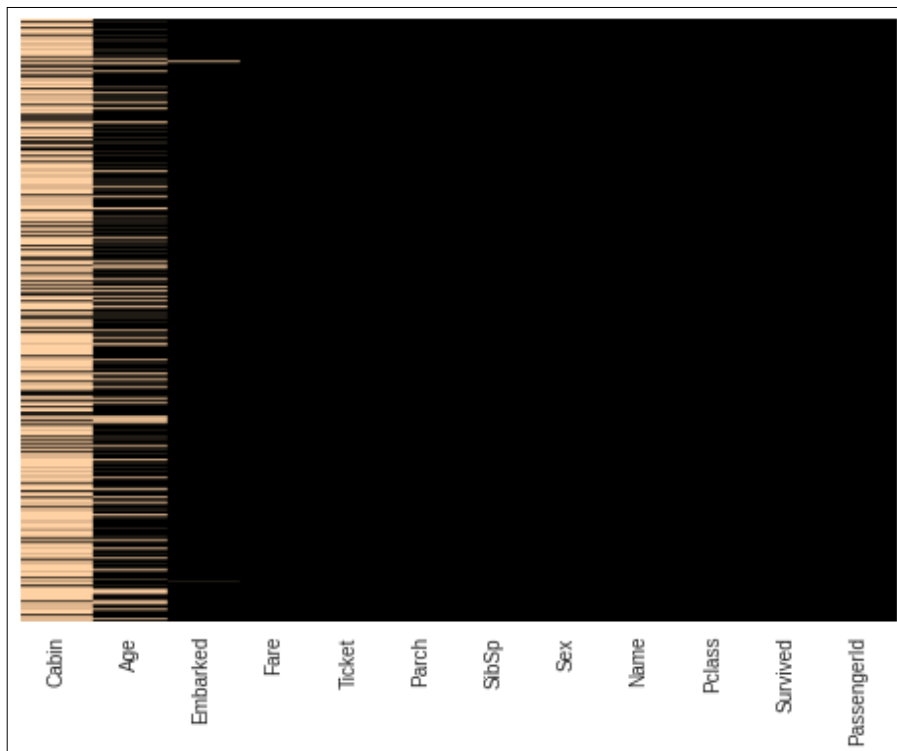
For more information and the code used to generate this data, see the following:
`http://bit.ly/1C0a24U`.

We can also calculate how much data is missing for each of the features:

```
In [83]: missing_perc=train_df.apply(lambda x: 100*(1-x.count().sum()/
(1.0*len(x))))
In [85]: sorted_missing_perc=missing_perc.order(ascending=False)
         sorted_missing_perc
Out[85]: Cabin          77.104377
         Age            19.865320
         Embarked        0.224467
         Fare            0.000000
         Ticket          0.000000
         Parch           0.000000
         SibSp           0.000000
         Sex             0.000000
         Name            0.000000
         Pclass          0.000000
         Survived        0.000000
         PassengerId     0.000000
         dtype: float64
```

Thus, we can see that most of the `Cabin` data is missing (77%), while around 20% of the `Age` data is missing. We then decide to drop the `Cabin` data from our learning feature set as the data is too sparse to be of much use.

Let us do a further breakdown of the various features that we would like to examine. In the case of categorical/discrete features, we use bar plots; for continuous valued features, we use histograms:

```
In [137]:  import random
             bar_width=0.1
             categories_map={'Pclass':{'First':1,'Second':2,
'Third':3},
             'Sex':{'Female':'female','Male':'male'},
             'Survived':{'Perished':0,'Survived':1},
             'Embarked':{'Cherbourg':'C','Queenstown':'Q','Southampton
':'S'},
             'SibSp': { str(x):x for x in [0,1,2,3,4,5,8]},
```

```
                'Parch': {str(x):x for x in range(7)}
                }
           colors=['red','green','blue','yellow','magenta','orange']
           subplots=[111,211,311,411,511,611,711,811]
           cIdx=0
          fig,ax=plt.subplots(len(subplots),figsize=(10,12))


             keyorder = ['Survived','Sex','Pclass','Embarked','SibSp',
'Parch']


for category_key,category_items in sorted(categories_map.iteritems(),
                                    key=lambda i:keyorder.
index(i[0])):
    num_bars=len(category_items)
    index=np.arange(num_bars)
    idx=0
    for cat_name,cat_val in sorted(category_items.iteritems()):
        ax[cIdx].bar(idx,len(train_df[train_df[category_key]==cat_val]),
label=cat_name,
                color=np.random.rand(3,1))
        idx+=1
    ax[cIdx].set_title('%s Breakdown' % category_key)
    xlabels=sorted(category_items.keys())
    ax[cIdx].set_xticks(index+bar_width)
    ax[cIdx].set_xticklabels(xlabels)
    ax[cIdx].set_ylabel('Count')
    cIdx +=1
fig.subplots_adjust(hspace=0.8)
for hcat in ['Age','Fare']:
    ax[cIdx].hist(train_df[hcat].dropna(),color=np.random.rand(3,1))
    ax[cIdx].set_title('%s Breakdown' % hcat)
    #ax[cIdx].set_xlabel(hcat)
    ax[cIdx].set_ylabel('Frequency')
    cIdx +=1


fig.subplots_adjust(hspace=0.8)
plt.show()
```
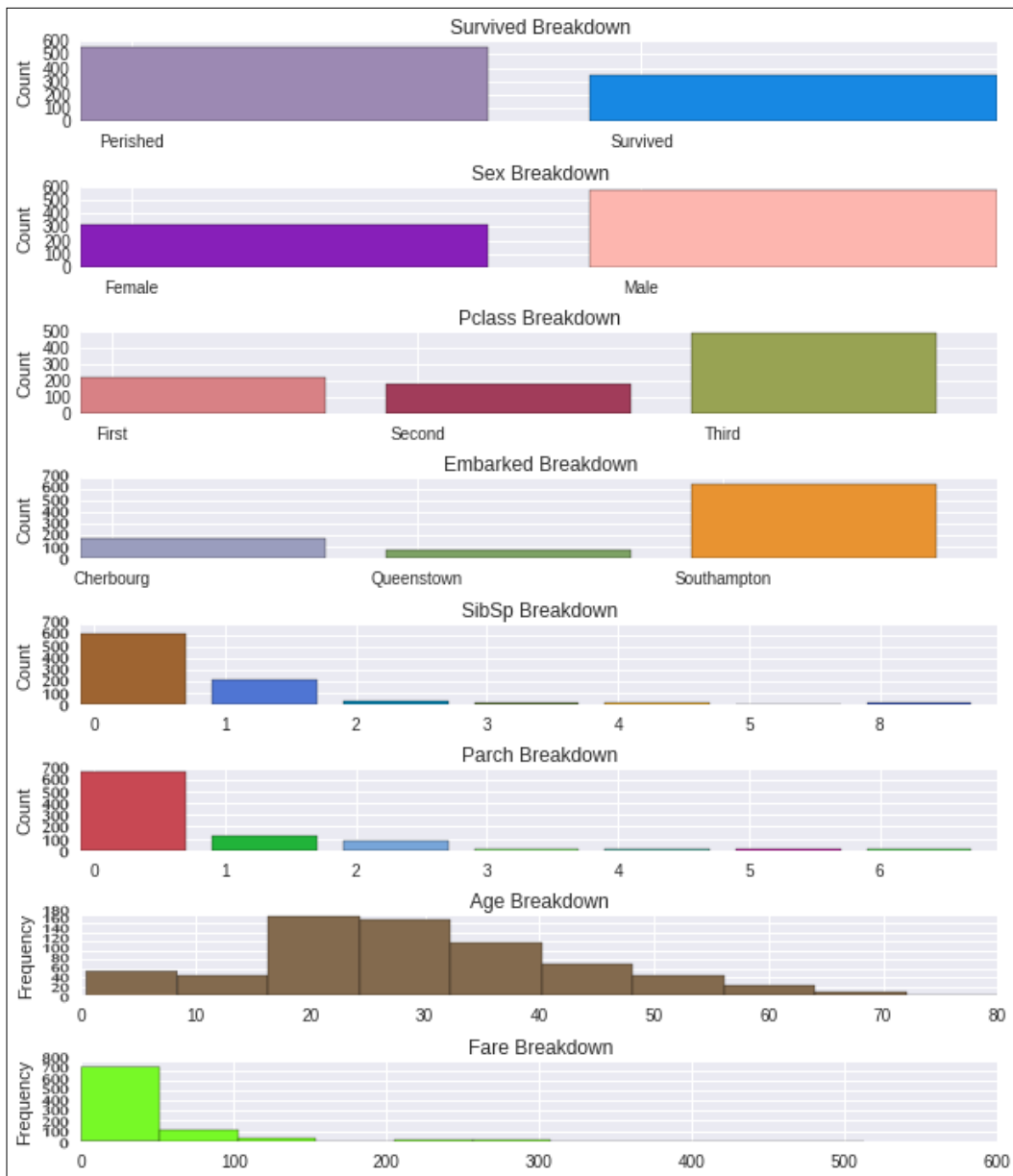
From the data and illustration in the preceding figure, we can observe the following:

- About twice as many passengers perished than survived (62% vs. 38%).
- There were about twice as many male passengers as female passengers (65% versus 35%).

- There were about 20% more passengers in the third class versus the first and second together (55% versus 45%).

- Most passengers were solo, that is, had no children, parents, siblings, or spouse on board.

These observations might lead us to dig deeper and investigate whether there is some correlation between chances of survival and gender and also fare class, particularly if we take into account the fact that the Titanic had a women-and-children-first policy (http://en.wikipedia.org/wiki/Women_and_children_first) and the fact that the Titanic was carrying fewer lifeboats (20) than it was designed to (32).

In light of this, let us further examine the relationships between survival and some of these features. We start with gender:

```
In [85]: from collections import OrderedDict

         num_passengers=len(train_df)

         num_men=len(train_df[train_df['Sex']=='male'])

         men_survived=train_df[(train_df['Survived']==1 ) & (train_
df['Sex']=='male')]

         num_men_survived=len(men_survived)

         num_men_perished=num_men-num_men_survived

         num_women=num_passengers-num_men

         women_survived=train_df[(train_df['Survived']==1) & (train_
df['Sex']=='female')]

         num_women_survived=len(women_survived)

         num_women_perished=num_women-num_women_survived

         gender_survival_dict=OrderedDict()

         gender_survival_dict['Survived']={'Men':num_men_
survived,'Women':num_women_survived}

         gender_survival_dict['Perished']={'Men':num_men_
perished,'Women':num_women_perished}

         gender_survival_dict['Survival Rate']= {'Men' :
           round(100.0*num_men_survived/num_men,2),
           'Women':round(100.0*num_women_survived/num_women,2)}
pd.DataFrame(gender_survival_dict)
Out[85]:
```

Gender	Survived	Perished	Survival Rate
Men	109	468	18.89
Women	233	81	74.2

We now illustrate this data in a bar chart using the following command:

```
In [76]: #code to display survival by gender
        fig = plt.figure()
        ax = fig.add_subplot(111)
        perished_data=[num_men_perished, num_women_perished]
        survived_data=[num_men_survived, num_women_survived]
        N=2
        ind = np.arange(N)     # the x locations for the groups
        width = 0.35


        survived_rects = ax.barh(ind, survived_data,
width,color='green')
        perished_rects = ax.barh(ind+width, perished_data,
width,color='red')


        ax.set_xlabel('Count')
        ax.set_title('Count of Survival by Gender')
        yTickMarks = ['Men','Women']
        ax.set_yticks(ind+width)
        ytickNames = ax.set_yticklabels(yTickMarks)
        plt.setp(ytickNames, rotation=45, fontsize=10)


        ## add a legend
        ax.legend((survived_rects[0], perished_rects[0]), ('Survived',
'Perished') )
        plt.show()
```

The preceding code produces the following bar graph:

From the preceding plot, we can see that a majority of the women survived (74%), while most of the men perished (only 19% survived).

This leads us to the conclusion that the gender of the passenger may be a contributing factor to whether a passenger survived or not.

Next, let us look at passenger class. First, we generate the survived and perished data for each of the three passenger classes, as well as survival rates and show them in a table:

```
In [86]:
from collections import OrderedDict
num_passengers=len(train_df)
num_class1=len(train_df[train_df['Pclass']==1])
class1_survived=train_df[(train_df['Survived']==1 ) & (train_
df['Pclass']==1)]
num_class1_survived=len(class1_survived)
num_class1_perished=num_class1-num_class1_survived
num_class2=len(train_df[train_df['Pclass']==2])
```

```
class2_survived=train_df[(train_df['Survived']==1) & (train_
df['Pclass']==2)]

num_class2_survived=len(class2_survived)

num_class2_perished=num_class2-num_class2_survived

num_class3=num_passengers-num_class1-num_class2

class3_survived=train_df[(train_df['Survived']==1 ) & (train_
df['Pclass']==3)]

num_class3_survived=len(class3_survived)

num_class3_perished=num_class3-num_class3_survived

pclass_survival_dict=OrderedDict()

pclass_survival_dict['Survived']={'1st Class':num_class1_survived,
                                  '2nd Class':num_class2_survived,
                                  '3rd Class':num_class3_survived}

pclass_survival_dict['Perished']={'1st Class':num_class1_perished,
                                  '2nd Class':num_class2_perished,
                                  '3rd Class':num_class3_perished}

pclass_survival_dict['Survival Rate']= {'1st Class' : round(100.0*num_
class1_survived/num_class1,2),

            '2nd Class':round(100.0*num_class2_survived/num_class2,2),

            '3rd Class':round(100.0*num_class3_survived/num_
class3,2),}

pd.DataFrame(pclass_survival_dict)
```

```
Out[86]:
```

Passenger Class	Survived	Perished	Survival Rate
First Class	136	80	62.96
Second Class	87	97	47.28
Third Class	119	372	24.24

We can then plot the data by using matplotlib in a similar manner to that for the survivor count by gender as described earlier:

```
In [186]:

fig = plt.figure()

ax = fig.add_subplot(111)

perished_data=[num_class1_perished, num_class2_perished, num_class3_
perished]
```

```
survived_data=[num_class1_survived, num_class2_survived, num_class3_
survived]
N=3
ind = np.arange(N)                    # the x locations for the groups
width = 0.35
survived_rects = ax.barh(ind, survived_data, width,color='blue')
perished_rects = ax.barh(ind+width, perished_data, width,color='red')
ax.set_xlabel('Count')
ax.set_title('Survivor Count by Passenger class')
yTickMarks = ['1st Class','2nd Class', '3rd Class']
ax.set_yticks(ind+width)
ytickNames = ax.set_yticklabels(yTickMarks)
plt.setp(ytickNames, rotation=45, fontsize=10)
## add a legend
ax.legend( (survived_rects[0], perished_rects[0]), ('Survived',
'Perished'),
          loc=10 )
plt.show()
```

This produces the following bar plot:
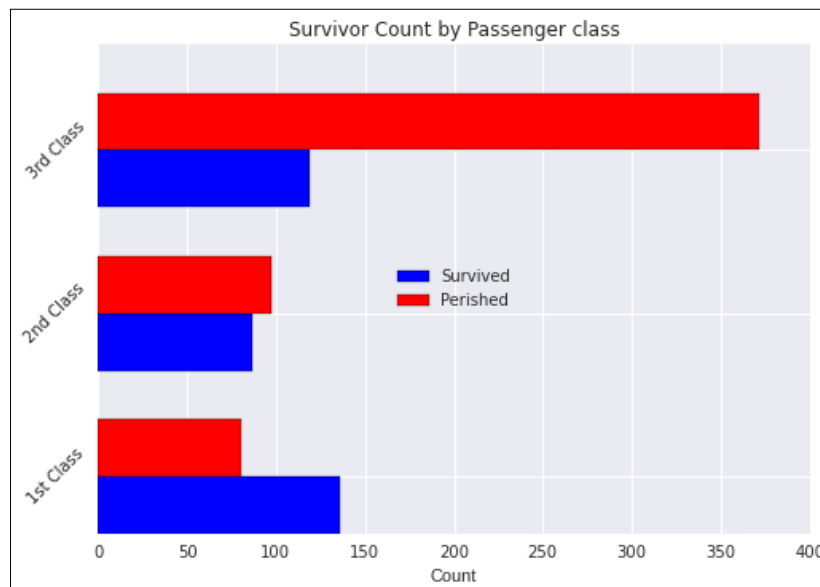
It seems clear from the preceding data and illustration that the higher the passenger fare class is, the greater are one's chances of survival.

Given that both gender and fare class seem to influence the chances of a passenger's survival, let's see what happens when we combine these two features and plot a combination of both. For this, we shall use the `crosstab` function in pandas.

```
In [173]: survival_counts=pd.crosstab([train_df.Pclass,train_
df.Sex],train_df.Survived.astype(bool))
          survival_counts
Out[173]:              Survived False  True

            Pclass     Sex

            1          female     3     91

                       male      77     45

            2          female     6     70

                       male      91     17

            3          female    72     72

                       male     300     47
```

Let us now display this data using `matplotlib`. First, let's do some re-labeling for display purposes:

```
In [183]: survival_counts.index=survival_counts.index.set_levels([['1st',
'2nd', '3rd'], ['Women', 'Men']])
In [184]: survival_counts.columns=['Perished','Survived']
```

Now, we plot the data by using the `plot` function of a pandas DataFrame:

```
In [185]: fig = plt.figure()
          ax = fig.add_subplot(111)
          ax.set_xlabel('Count')
          ax.set_title('Survivor Count by Passenger class, Gender')
          survival_counts.plot(kind='barh',ax=ax,width=0.75,
                              color=['red','black'], xlim=(0,400))
Out[185]: <matplotlib.axes._subplots.AxesSubplot at 0x7f714b187e90>
```

A naïve approach to Titanic problem

Our first attempt at classifying the Titanic data is to use a *naïve*, yet very intuitive, approach. This approach involves the following steps:

1. Select a set of features *S*, which influence whether a person survives or not.

2. For each possible combination of features, use the training data to indicate whether the majority of cases survived or not. This can be evaluated in what is known as a survival matrix.

3. For each test example that we wish to predict survival, look up the combination of features that corresponds to the values of its features and assign its predicted value to the survival value in the survival table. This approach is a naive K-nearest neighbor approach.

Based on what we have seen earlier in our analysis, there are three features that seem to have the most influence on the survival rate:

- Passenger class
- Gender
- Passenger fare (bucketed)

We include passenger fare as it is related to passenger class.

The survival table looks something similar to the following:

	NumberOfPeople	Pclass	PriceBucket	Sex	Survived
0	0	1	0	female	0
1	1	1	0	male	0
2	0	1	1	female	0
3	0	1	1	male	0
4	7	1	2	female	1
5	34	1	2	male	0
6	1	1	3	female	1
7	19	1	3	male	0
8	0	2	0	female	0
9	0	2	0	male	0
10	35	2	1	female	1
11	63	2	1	male	0
12	31	2	2	female	1
13	25	2	2	male	0
14	4	2	3	female	1
15	6	2	3	male	0
16	64	3	0	female	1
17	256	3	0	male	0
18	43	3	1	female	1
19	38	3	1	male	0
20	21	3	2	female	0
21	24	3	2	male	0
22	10	3	3	female	0
23	5	3	3	male	0

The code for generating this table can be found in the file `survival_data.py` which is attached. To see how we use this table, let us take a look at a snippet of our test data:

```
In [192]: test_df.head(3)[['PassengerId','Pclass','Sex','Fare']]
Out[192]: PassengerId   Pclass   Sex      Fare
       0         892       3     male     7.8292
       1         893       3     female   7.0000
       2         894       2     male     9.6875
```

For passenger 892, we see that he is male, his ticket price was 7.8292, and he travelled in the third class.

Hence, the key for survival table lookup for this passenger is *{Sex='male', Pclass=3, PriceBucket=0 (since 7.8292 falls in bucket 0)}.*

If we look up the survival value corresponding to this key in our survival table (row 17), we see that the value is 0 = Perished; this is the value that we will predict.

Similarly, for passenger 893, we have *key={Sex='female', Pclass=3, PriceBucket=0}.*

This corresponds to row 16, and hence, we will predict 1, that is, survived, and her predicted survival is 1, that is, survived.

Thus, our results look like the following command:

```
> head -4 csv/surv_results.csv
PassengerId,Survived
892,0
893,1
894,0
```

The source of this information is at: `http://bit.ly/1FU7mXj`.

Using the survival table approach outlined earlier, one is able to achieve an accuracy of 0.77990 on Kaggle (`http://www.kaggle.com`).

The survival table approach, while intuitive, is a very basic approach that represents only the tip of the iceberg of possibilities in machine learning.

In the following sections, we will take a whirlwind tour of various machine learning algorithms that will help you, the reader, to get a feel for what is available in the machine learning universe.

The scikit-learn ML/classifier interface

We'll be diving into the basic principles of machine learning and demonstrate the use of these principles via the `scikit-learn` basic API.

The `scikit-learn` library has an estimator interface. We illustrate it by using a linear regression model. For example, consider the following:

```
In [3]: from sklearn.linear_model import LinearRegression
```

The estimator interface is instantiated to create a model, which is a linear regression model in this case:

```
In [4]: model = LinearRegression(normalize=True)

In [6]: print model
    LinearRegression(copy_X=True, fit_intercept=True, normalize=True)
```

Here, we specify `normalize=True`, indicating that the x-values will be normalized before regression. **Hyperparameters** (estimator parameters) are passed on as arguments in the model creation. This is an example of creating a model with tunable parameters.

The estimated parameters are obtained from the data when the data is fitted with an estimator. Let us first create some sample training data that is normally distributed about `y = x/2`. We first generate our `x` and `y` values:

```
In [51]: sample_size=500
         x = []
         y = []


         for i in range(sample_size):
             newVal = random.normalvariate(100,10)
             x.append(newVal)
             y.append(newVal / 2.0 + random.normalvariate(50,5))
```

`sklearn` takes a 2D array of `num_samples` x `num_features` as input, so we convert our x data into a 2D array:

```
In [67]: X = np.array(x)[:,np.newaxis]
         X.shape
Out[67]: (500, 1)
```

In this case, we have 500 samples and 1 feature, x. We now train/fit the model and display the slope (coefficient) and the intercept of the regression line, which is the prediction:

```
In [71]: model.fit(X,y)
         print "coeff=%s, intercept=%s" % (model.coef_,model.intercept_)
         coeff=[ 0.47071289], intercept=52.7456611783
```

This can be visualized as follows:

```
In [65]: plt.title("Plot of linear regression line and training data")
         plt.xlabel('x')
         plt.ylabel('y')
         plt.scatter(X,y,marker='o', color='green', label='training
data');
         plt.plot(X,model.predict(X), color='red', label='regression
line')
         plt.legend(loc=2)


Out[65]: [<matplotlib.lines.Line2D at 0x7f11b0752350]
```
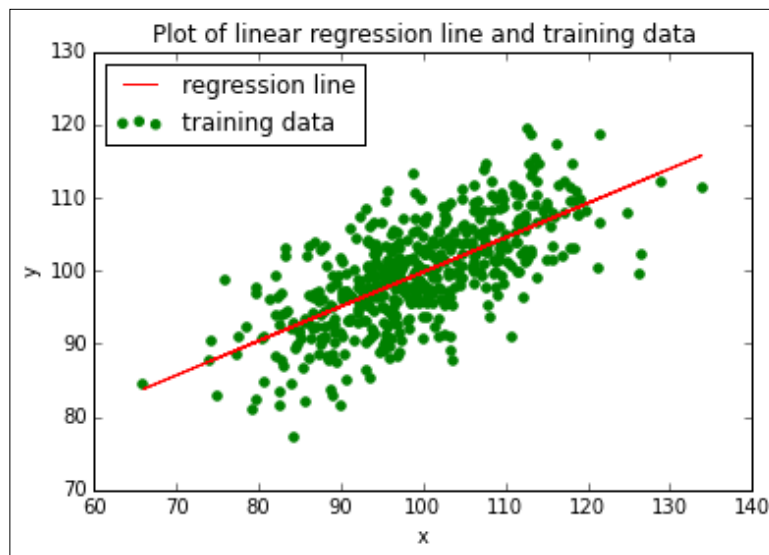

To summarize the basic use of estimator interface, follow these steps:

1. Define your model - *LinearRegression*, *SupportVectorMachine*, *DecisionTrees*, and so on. You can specify the needed hyperparameters in this step. For example, `normalize=True` as specified earlier.

2. Once the model has been defined, you can train your model on your data by calling the `fit(..)` method on the model defined in the previous step.

3. Once we have fit the model, we can call the `predict(..)` method on test data in order to make predictions or estimations.

4. In the case of a supervised learning problem, the `predict(X)` method is given unlabeled observations `X` and returns predicted labels `y`.

> For extra reference, please see the following: `http://bit.ly/1FU7mXj` and `http://bit.ly/1QqFN2V`.

Supervised learning algorithms

We will take a brief tour of some well-known supervised learning algorithms and see how we can apply them to the Titanic survival prediction problem described earlier.

Constructing a model using Patsy for scikit-learn

Before we start our tour of the machine learning algorithms, we need to know a little bit about the `Patsy` library. We will make use of `Patsy` to design features that will be used in conjunction with `scikit-learn`. `Patsy` is a package for creating what are known as design matrices. These design matrices are transformations of the features in our input data. The transformations are specified by expressions known as formulas, which correspond to a specification of what features we wish the machine learning program to utilize in learning.

A simple example of this is as follows:

Suppose that we want a linear regression of y against some other variables of x, a, and b and the interaction between a and b; then, we can specify the model as follows:

```
import patsy as pts
pts.dmatrices("y ~ x + a + b + a:b", data)
```

In the preceding line of code, the formula is specified by the following expression: `y ~ x + a + b + a:b`.

> For further reference, look at: `http://patsy.readthedocs.org/en/latest/overview.html`

General boilerplate code explanation

In this section, we will introduce boilerplate code for the implementation of the various following algorithms by using `Patsy` and `scikit-learn`. The reason for doing this is that most of the code for the following algorithms is repeatable.

In the following sections, the workings of the algorithms will be described and the code specific to each algorithm will be provided as attachments to the chapter.

1. First, let's make sure that we're in the correct folder by using the following command line. Assuming that the working directory is located at `~/devel/ Titanic`, we have:

```
In [17]: %cd ~/devel/Titanic

         /home/youruser/devel/sandbox/Learning/Kaggle/Titanic
```

2. Here, we import the needed packages and read in our training and test datasets:

```
In [18]: import matplotlib.pyplot as plt

         import pandas as pd

         import numpy as np

         import patsy as pt

In [19]: train_df = pd.read_csv('csv/train.csv', header=0)

         test_df = pd.read_csv('csv/test.csv', header=0)
```

3. Next, we specify the formulas we would like to submit to `Patsy`:

```
In [21]: formula1 = 'C(Pclass) + C(Sex) + Fare'

         formula2 = 'C(Pclass) + C(Sex)'

         formula3 = 'C(Sex)'

         formula4 = 'C(Pclass) + C(Sex) + Age + SibSp + Parch'

         formula5 = 'C(Pclass) + C(Sex) + Age + SibSp + Parch +
C(Embarked)'

         formula6 = 'C(Pclass) + C(Sex) + Age + SibSp +
C(Embarked)'

         formula7 = 'C(Pclass) + C(Sex) + SibSp + Parch +
C(Embarked)'

         formula8 = 'C(Pclass) + C(Sex) + SibSp + Parch +
C(Embarked)'


In [23]: formula_map = {'PClass_Sex_Fare' : formula1,

                        'PClass_Sex' : formula2,
```

```
                         'Sex' : formula3,
                         'PClass_Sex_Age_Sibsp_Parch' : formula4,
                         'PClass_Sex_Age_Sibsp_Parch_Embarked' :
     formula5,
               'PClass_Sex_Embarked' : formula6,
               'PClass_Sex_Age_Parch_Embarked' : formula7,
               'PClass_Sex_SibSp_Parch_Embarked' : formula8
                  }
```

We will define a function that helps us handle missing values. The following function finds the cells within the DataFrame that have null values, obtains the set of similar passengers, and sets the null value to the mean value of that feature for the set of similar passengers. Similar passengers are defined as those having the same gender and passenger class as the passengers with the null feature value.

```
In [24]:
def fill_null_vals(df,col_name):
    null_passengers=df[df[col_name].isnull()]
    passenger_id_list = null_passengers['PassengerId'].tolist()
    df_filled=df.copy()
    for pass_id in passenger_id_list:
        idx=df[df['PassengerId']==pass_id].index[0]
        similar_passengers = df[(df['Sex']==
        null_passengers['Sex'][idx]) &
        (df['Pclass']==null_passengers['Pclass'][idx])]
        mean_val = np.mean(similar_passengers[col_name].dropna())
        df_filled.loc[idx,col_name]=mean_val
    return df_filled
```

Here, we create filled versions of our training and test DataFrames.

Our test DataFrame is what the fitted `scikit-learn` model will generate predictions on to produce output that will be submitted to Kaggle for evaluation:

```
In [28]: train_df_filled=fill_null_vals(train_df,'Fare')
         train_df_filled=fill_null_vals(train_df_filled,'Age')
         assert len(train_df_filled)==len(train_df)

         test_df_filled=fill_null_vals(test_df,'Fare')
         test_df_filled=fill_null_vals(test_df_filled,'Age')
         assert len(test_df_filled)==len(test_df)
```

Here is the actual implementation of the call to `scikit-learn` to learn from the training data by fitting a model and then generate predictions on the test dataset. Note that even though this is boilerplate code, for the purpose of illustration, an actual call is made to a specific algorithm, in this case, `DecisionTreeClassifier`.

The output data is written to files with descriptive names, for example, `csv/dt_PClass_Sex_Age_Sibsp_Parch_1.csv` and `csv/dt_PClass_Sex_Fare_1.csv`.

```
In [29]:

from sklearn import metrics,svm, tree
for formula_name, formula in formula_map.iteritems():
        print "name=%s formula=%s" % (formula_name,formula)
     y_train,X_train = pt.dmatrices('Survived ~ ' + formula,
                                        train_df_filled,return_
type='dataframe')
    y_train = np.ravel(y_train)
    model = tree.DecisionTreeClassifier(criterion='entropy',
            max_depth=3,min_samples_leaf=5)
    print "About to fit..."
    dt_model = model.fit(X_train, y_train)
    print "Training score:%s" % dt_model.score(X_train,y_train)
    X_test=pt.dmatrix(formula,test_df_filled)
    predicted=dt_model.predict(X_test)
    print "predicted:%s" % predicted[:5]
    assert len(predicted)==len(test_df)
    pred_results = pd.Series(predicted,name='Survived')
    dt_results = pd.concat([test_df['PassengerId'],
              pred_results],axis=1)
    dt_results.Survived = dt_results.Survived.astype(int)
    results_file = 'csv/dt_%s_1.csv' % (formula_name)
    print "output file: %s\n" % results_file
    dt_results.to_csv(results_file,index=False)
```

The preceding code follows a standard recipe, and the synopsis is as follows:

1. Read in the training and test datasets
2. Fill in any missing values for the features we wish to consider in both datasets

3. Define formulas for the various feature combinations we wish to generate machine learning models for in `Patsy`

4. For each formula, perform the following set of steps:

    1. Call `Patsy` to create design matrices for our training feature set and training label set (designated by `X_train` and `y_train`).

    2. Instantiate the appropriate `scikit-learn` classifier. In this case, we use `DecisionTreeClassifier`.

    3. Fit the model by calling the `fit(..)` method.

    4. Make a call to `Patsy` to create a design matrix (`X_test`) for our predicted output via a call to `patsy.dmatrix(..)`.

    5. Predict on the `X_test` design matrix, and save the results in the variable predicted.

    6. Write our predictions to an output file, which will be submitted to Kaggle.

We will consider the following supervised learning algorithms:

- Logistic regression
- Support vector machine
- Decision tree
- Random forest

Logistic regression

In logistic regression, we attempt to predict the outcome of a categorical, that is, discrete-valued dependent, variable on the basis of one or more input predictor variables.

Logistic regression can be thought of as the equivalent of applying linear regression but on discrete or categorical variables. However, in the case of binary logistic regression (which applies to the Titanic problem), the function to which we're trying to fit is not a linear one as we're only trying to predict an outcome that can take only two values – 0 and 1. Using a linear function for our regression doesn't make sense as the output cannot take values between 0 and 1. Ideally, what we need to model for the regression of a binary valued output is some sort of step function for values 0 and 1. However, such a function is not well-defined and not differentiable, so an approximation with nicer properties was defined: the logistic function. The logistic function takes values between 0 and 1 but is skewed towards the extreme values of 0 and 1 and can be used as a good approximation for the regression of categorical variables.

The formal definition of the logistic regression function is as follows:

$$f(x) = \frac{1}{\left(1 + e^{-ax}\right)}$$

The following graph is a good illustration as to why the logistic function is suitable for binary logistic regression:
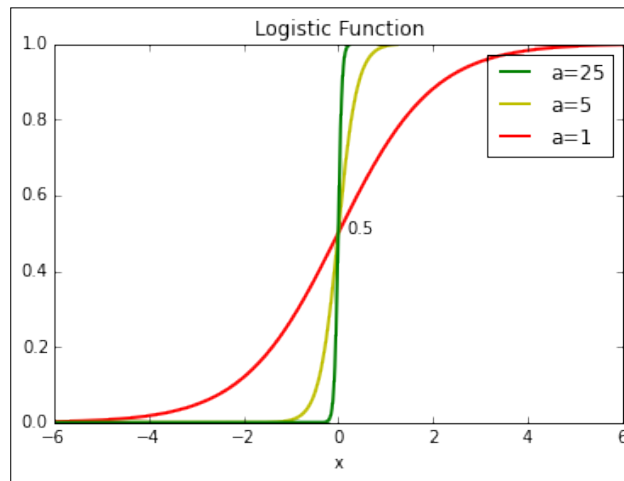
We can see that as we increase the value of our parameter a, we can get closer to taking on the 0 to 1 values and to the step function we wish to model. A simple application of the preceding function would be to set the output value to 0, if *f(x)* <*0.5*, and 1 if not.

The code for plotting the function is included in `plot_logistic.py`.

> A more detailed examination of the logistic regression may be found here at: `http://en.wikipedia.org/wiki/Logit` and `http://logisticregressionanalysis.com/86-what-is-logistic-regression`.

In applying logistic regression to the Titanic problem, we wish to predict a binary outcome, that is, whether a passenger survived or not.

We adapted the boilerplate code to use the `sklearn.linear_model.LogisticRegression` class of `scikit-learn`.

Upon submitting our data to Kaggle, the following results were obtained:

Formula	Kaggle Score
C(Pclass) + C(Sex) + Fare	0.76077
C(Pclass) + C(Sex)	0.76555
C(Sex)	0.76555
C(Pclass) + C(Sex) + Age + SibSp + Parch	0.74641
C(Pclass) + C(Sex) + Age + Sibsp + Parch + C(Embarked)	0.75598

The code implementing logistic regression can be found in the `run_logistic_regression_titanic.py` file.

Support vector machine

Support vector machine (**SVM**) is a powerful supervised learning algorithm used for classification and regression. It is a discriminative classifier–it draws a boundary between clusters or classifications of data, so new points can be classified on the basis of the cluster that they fall into.

SVMs do not just find a boundary line; they also try to determine margins for the boundary on either side. The SVM algorithm tries to find the boundary with the largest possible margin around it.

Support vectors are points that define the largest margin around the boundary–remove these points, and possibly, a larger margin can be found.

Hence the name, support, as they support the margin around the boundary line. The support vectors matter. This is illustrated in the following diagram:
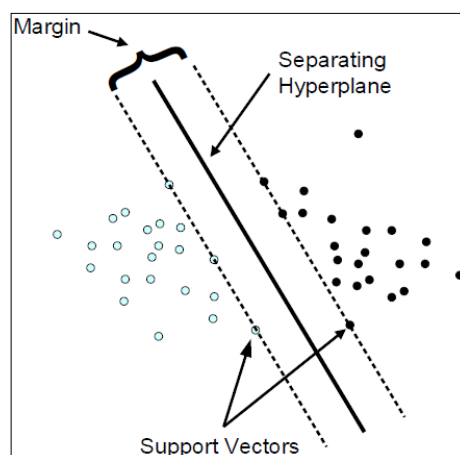
> For more information on this, refer to `http://winfwiki.wi-fom.de/images/c/cf/Support_vector_2.png`.

To use the SVM algorithm for classification, we specify one of the following three kernels: linear, poly, and rbf (also known as radial basis functions).

Then, we import the **support vector classifier** (**SVC**):

```
from sklearn import svm
```

We then instantiate an SVM classifier, fit the model, and predict the following:

```
model = svm.SVC(kernel=kernel)

svm_model = model.fit(X_train, y_train)

X_test = pt.dmatrix(formula, test_df_filled)

. . .
```

Upon submitting our data to Kaggle, the following results were obtained:

Formula	Kernel Type	Kaggle Score
C(Pclass) + C(Sex) + Fare	poly	0.71292
C(Pclass) + C(Sex)	poly	0.76555
C(Sex)	poly	0.76555
C(Pclass) + C(Sex) + Age + SibSp + Parch	poly	0.75598
C(Pclass) + C(Sex) + Age + Parch + C(Embarked)	poly	0.77512
C(Pclass) + C(Sex) + Age + Sibsp + Parch + C(embarked)	poly	0.79426
C(Pclass) + C(Sex) + Age + Sibsp + Parch + C(Embarked)	rbf	0.7512

The code can be seen in its entirety in the following file: `run_svm_titanic.py`.

Here, we see that the SVM with a kernel type of poly (polynomial) and the combination of Pclass, Sex, Age, Sibsp, and Parch features produces the best results when submitted to Kaggle. Surprisingly, it seems as if the embarkation point (Embarked) and whether the passenger travelled alone or with family members (Sibsp + Parch) do have a material effect on a passenger's chances of survival.

The latter effect was probably due to the women-and-children-first policy on the Titanic.

Decision trees

The basic idea behind decision trees is to use the training dataset to create a tree of decisions in order to make a prediction.

It recursively splits the training dataset into subsets on the basis of the value of a single feature. Each split corresponds to a node in the decision tree. The splitting process is continued until every subset is pure, that is, all elements belong to a single class. This always works except in cases where there are duplicate training examples that fall into different classes. In this case, the majority class wins.

The end result is a rule set for making predictions on the test dataset.

Decision trees encode a sequence of binary choices in a process that mimics how a human might classify things, but decide which question is most useful at each step by using the information criteria.

An example of this would be if you wished to determine whether an animal x is a mammal, fish, or a reptile; in this case, we would ask the following questions:

```
- Does x have fur?
Yes: x is a mammal
No: Does x have feathers?
Yes: x is a bird
No: Does x have scales?
Yes: Does x have gills?
Yes: x is a fish
No: x is a reptile
No: x is an amphibian
```

This generates a decision tree that looks similar to the following:

Decision tree for classification of vertibrates
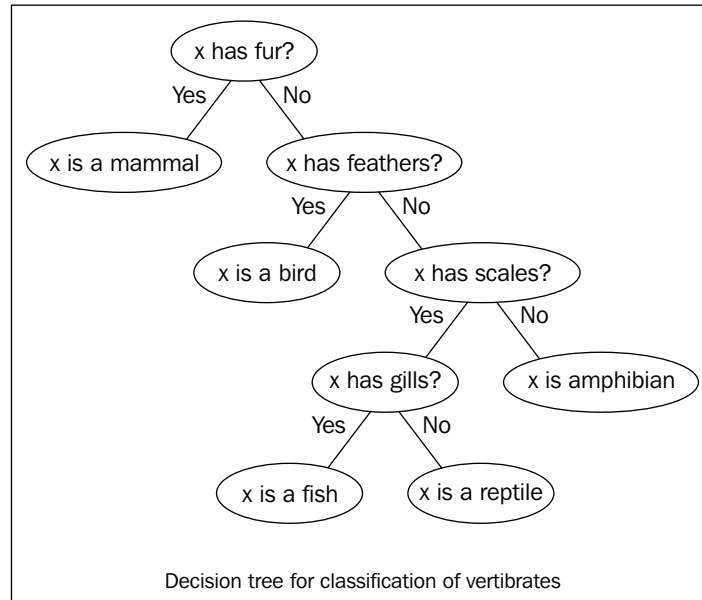
Refer to the following link for more information:
`http://bit.ly/1C0cM2e`.

The binary splitting of questions at each node is the essence of a decision tree algorithm. A major drawback of decision trees is that they can *overfit* the data.

They are so flexible that given a large depth, they can memorize the inputs, and this results in poor results when they are used to classify unseen data.

The way to fix this is to use multiple decision trees, and this is known as using an ensemble estimator. An example of an ensemble estimator is the random forest algorithm, which we will address next.

To use a decision tree in `scikit-learn`, we import the `tree` module:

```
from sklearn import tree
```

We then instantiate an SVM classifier, fit the model, and predict the following:

```
model = tree.DecisionTreeClassifier(criterion='entropy',
            max_depth=3,min_samples_leaf=5)
dt_model = model.fit(X_train, y_train)
X_test = dt.dmatrix(formula, test_df_filled)
#. . .
```

Upon submitting our data to Kaggle, the following results are obtained:

Formula	Kaggle Score
C(Pclass) + C(Sex) + Fare	0.77033
C(Pclass) + C(Sex)	0.76555
C(Sex)	0.76555
C(Pclass) + C(Sex) + Age + SibSp + Parch	0.76555
C(Pclass) + C(Sex) + Age + Parch + C(Embarked)	0.78947
C(Pclass) + C(Sex) + Age + Sibsp + Parch + C(Embarked)	0.79426

Random forest

The random forest is an example of a non-parametric model as are decision trees. Random forests are based on decision trees. The decision boundary is learned from the data itself. It doesn't have to be a line or a polynomial or radial basis function. The random forest model builds upon the decision tree concept by producing a large number of or a forest of decision trees. It takes a random sample of the data and identifies a set of features to grow each decision tree. The error rate of the model is compared across sets of decision trees to find the set of features that produces the strongest classification model.

To use a random forest in scikit-learn, we import the RandomForestClassifier module:

```
from sklearn import RandomForestClassifier
```

We then instantiate a `random forest` classifier, fit the model, and predict the following:

```
model = RandomForestClassifier(n_estimators=num_estimators,
                               random_state=0)
rf_model = model.fit(X_train, y_train)
X_test = dt.dmatrix(formula, test_df_filled)
. . .
```

Upon submitting our data to Kaggle (Formula: *C(Pclass) + C(Sex) + Age + Sibsp + Parch + C(Embarked)*), the following results are obtained:

Formula	Kaggle Score
10	0.74163
100	0.76077
1000	0.76077
10000	0.77990
100000	0.77990

Unsupervised learning algorithms

There are two tasks that we are mostly concerned with in unsupervised learning: dimensionality reduction and clustering.

Dimensionality reduction

Dimensionality reduction is used to help visualize higher-dimensional data in a systematic way. This is useful because our human brains can visualize only three spatial dimensions (and possibly, a temporal one), but most datasets involve much higher dimensions.

The typical technique used in dimensionality reduction is **Principal Component Analysis** (**PCA**). PCA involves using linear algebra techniques to project higher-dimensional data onto a lower-dimensional space. This inevitably involves the loss of information, but often by projecting along the correct set and number of dimensions, the information loss can be minimized. A common dimensionality reduction technique is to find the combination of variables that explain the most variance (proxy for information) in our data and project along these dimensions.

In the case of unsupervised learning problems, we do not have the set of labels (Y), and so, we only call `fit()` on the input data X itself, and for PCA, we call `transform()` instead of `predict()` as we're trying to transform the data into a new representation.

One of the datasets that we will be using to demonstrate USL is the iris dataset, possibly the most famous dataset in all of machine learning.

The `scikit-learn` library provides a set of pre-packaged datasets, which are available via the `sklearn.datasets` modules. The iris dataset is one of them.

The iris dataset consists of 150 samples of data from three different species of iris flowers - versicolor, setosa, and virginica with 50 samples of each type. The dataset consists of four features/dimensions:

- petal length
- petal width
- sepal length
- sepal width

The length and width values are in centimeters. It can be loaded as follows:

```
from sklearn.datasets import load_iris

iris = load_iris()
```

In our examination of unsupervised learning, we will be focusing on how to visualize and cluster this data.

Before discussing unsupervised learning, let us examine the iris data a bit. The `load_iris()` command returns what is known as a bunch object, which is essentially a dictionary with keys in addition to the key containing the data. Hence, we have the following:

```
In [2]: iris_data.keys()
Out[2]: ['target_names', 'data', 'target', 'DESCR', 'feature_names']
```

Further, the data itself looks similar to the following:

```
In [3]: iris_data.data.shape
Out[3]: (150, 4)
```

This corresponds to 150 samples of four features. These four features are shown as follows:

```
In [4]: print iris_data.feature_names
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal
width (cm)']
```

We can also take a peek at the actual data:

```
In [9]: print iris_data.data[:2]
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]]
```

Our target names (what we're trying to predict) look similar to the following:

```
In [10]: print iris_data.target_names
        ['setosa' 'versicolor' 'virginica']
```

As noted earlier, the iris feature set corresponds to five-dimensional data and we cannot visualize this on a color plot. One thing that we can do is pick two features and plot them against each other, while using color to differentiate between the species feature. We do this next for all the possible combinations of features, selecting two at a time for a set of six different possibilities. These combinations are as follows:

- Sepal width versus sepal length
- Sepal width versus petal width
- Sepal width versus petal length
- Sepal length versus petal width
- Sepal length versus petal length
- Petal width versus petal length
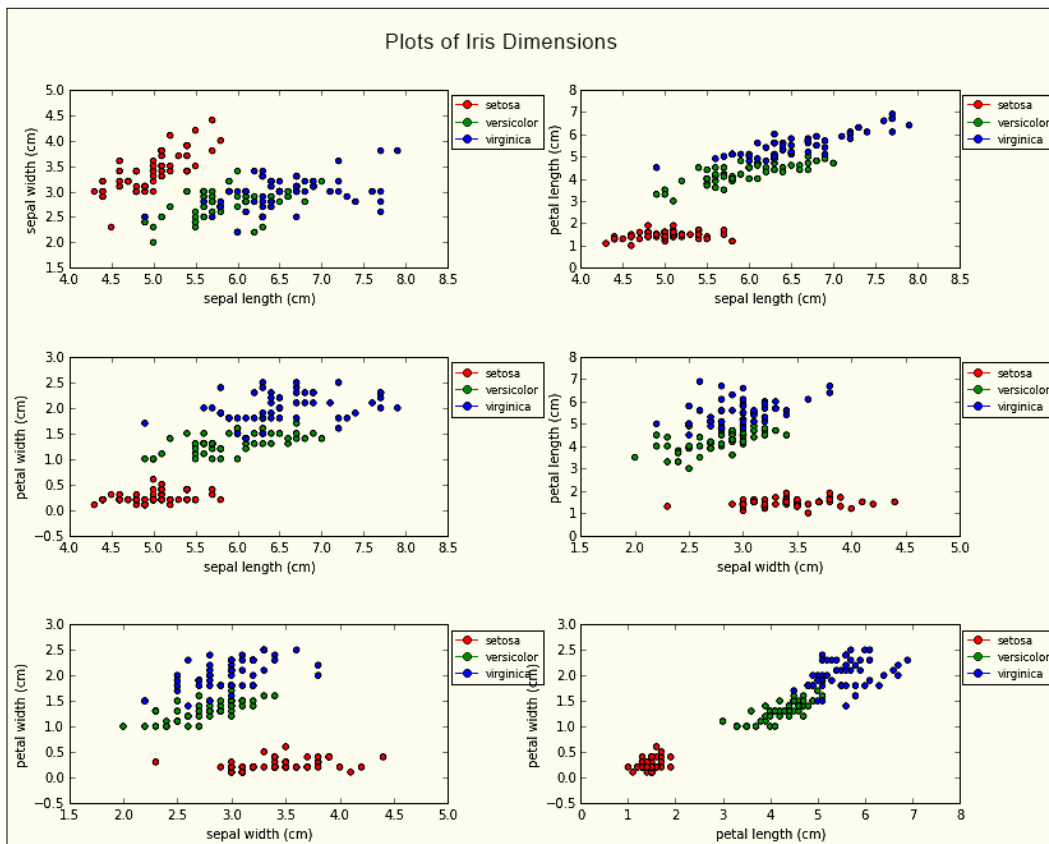
The code for this may be found in the following file: `display_iris_dimensions.py`. From the preceding plots, we can observe that the setosa points tend to be clustered by themselves, while there is a bit of overlap between the virginica and the versicolor points. This may lead us to conclude that the latter two species are more closely related to one another than to the setosa species.

These are, however, two-dimensional slices of data. What if we wanted a somewhat more holistic view of the data, with some representation of all four sepal and petal dimensions?

What if there were some hitherto undiscovered connection between the dimensions that our two-dimensional plot wasn't showing? Is there a means of visualizing this? Enter dimensionality reduction. We will use dimensionality reduction to extract two combinations of sepal and petal dimensions to help visualize it.

We can apply dimensionality reduction to do this as follows:

```
In [118]: X, y = iris_data.data, iris_data.target
          from sklearn.decomposition import PCA
          pca = PCA(n_components=2)
          pca.fit(X)
          X_red=pca.transform(X)
          print "Shape of reduced dataset:%s" % str(X_red.shape)


        Shape of reduced dataset:(150, 2)
```

Thus, we see that the reduced dataset is now in two dimensions. Let us display the data visually in two dimensions as follows:

```
In [136]: figsize(8,6)
          fig=plt.figure()
          fig.suptitle("Dimensionality reduction on iris data")
          ax=fig.add_subplot(1,1,1)
          colors=['red','yellow','magenta']
          cols=[colors[i] for i in iris_data.target]
          ax.scatter(X_red[:,0],X[:,1],c=cols)
Out[136]:
<matplotlib.collections.PathCollection at 0x7fde7fae07d0>
```
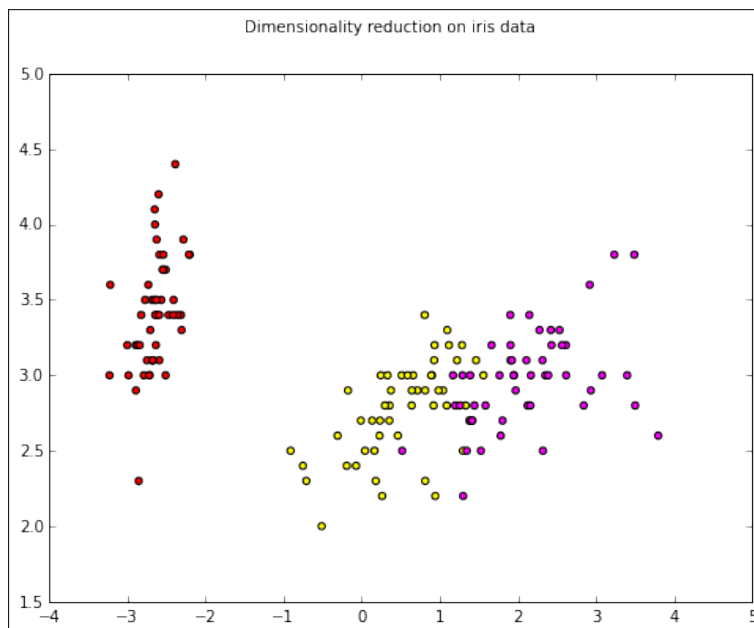
We can examine the makeup of the PCA-reduced two dimensions as follows:

```
In [57]:
print "Dimension Composition:"
idx=1
for comp in pca.components_:
    print "Dim %s" % idx
    print " + ".join("%.2f x %s" % (value, name)
                     for value, name in zip(comp, iris_data.feature_
names))
    idx += 1


Dimension Composition:
Dim 1
0.36 x sepal length (cm) + -0.08 x sepal width (cm) + 0.86 x petal length
(cm) + 0.36 x petal width (cm)
Dim 2
-0.66 x sepal length (cm) + -0.73 x sepal width (cm) + 0.18 x petal
length (cm) + 0.07 x petal width (cm)
```

Thus, we can see that the two reduced dimensions are a linear combination of all four sepal and petal dimensions.

The source of this information is at: `https://github.com/jakevdp/sklearn_pycon2014`.

K-means clustering

The idea behind clustering is to group together similar points in a dataset on the basis of a given criterion, thus finding clusters in the data.

The K-means algorithm aims to partition a set of data points into K clusters such that each data point belongs to the cluster with the nearest mean point or centroid.

To illustrate K-means clustering, we can apply it to the set of reduced iris data that we obtained via PCA, but in this case, we do not pass the actual labels to the `fit(..)` method as we do for supervised learning:

```
In [142]: from sklearn.cluster import KMeans
          k_means = KMeans(n_clusters=3, random_state=0)
          k_means.fit(X_red)
          y_pred = k_means.predict(X_red)
```

We now display the clustered data as follows:

```
In [145]: figsize(8,6)
          fig=plt.figure()
          fig.suptitle("K-Means clustering on PCA-reduced iris data,
K=3")
          ax=fig.add_subplot(1,1,1)
          ax.scatter(X_red[:, 0], X_red[:, 1], c=y_pred);
```
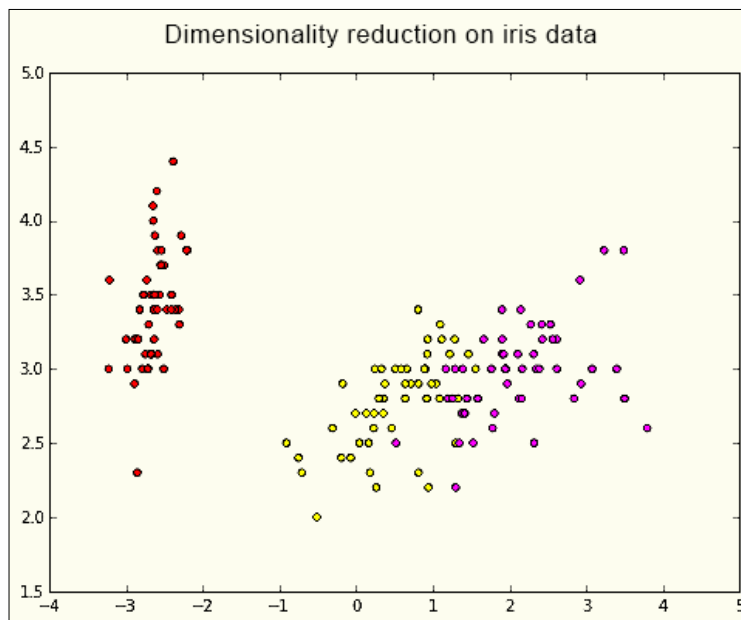

Note that our K-means algorithm clusters do not exactly correspond to the dimensions obtained via PCA. The source code is available at `https://github.com/jakevdp/sklearn_pycon2014`.

> More information on K-means clustering in `scikit-learn` and, in general, can be found here at:
>
> `http://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_iris.html` and `http://en.wikipedia.org/wiki/K-means_clustering`.

Summary

In this chapter, we embarked on a whirlwind tour of machine learning, examining the role of pandas in feature extraction, selection, and engineering as well as learning about key concepts in machine learning such as supervised versus unsupervised learning. We also had a brief introduction to a few key algorithms in both methods of machine learning and used the `scikit-learn` package to utilize these algorithms to learn and make predictions on data. This chapter was not intended to be a comprehensive treatment of machine learning, but rather to illustrate how pandas can be used to assist users in the machine learning space.

Index

Symbols

installing, on Windows 26
installing, URL 26
installing, via Anaconda 27
installing, Wakari 27
installing, with virtualenv 27
URL 24
interpolate() function
reference link 143
IPython
IPython Notebook
URL 24
isin() function, pandas 272

J

joining 114
join operation
reference link 125

K

Kaggle
URL 287
Kaggle Titanic competition application
about 287, 288
problem of overfitting 288, 289
K-means clustering 321, 322
K-means clustering, scikit-learn
reference link 322

L

label-oriented indexing
about 75, 77, 78
selection, Boolean array used 78, 79
lagging 147
lambda functions
reference link 104
law of large numbers (LLN)
reference link 165
levels
reordering 89
swapping 89
linear regression
about 190-192
example 192-195

Linux
Anaconda installation 17
IPython installation 26
panda installation 20
Python installation 12
logical operators, NumPy array
np.all() 45
np.any() 46
logical subsetting
about 272
in pandas 273
in R 272
logistic regression
about 309, 310
reference link 310

M

machine learning
about 285
reference link 285
machine learning application
Kaggle Titanic competition 287
machine learning systems
working 287
Mac OS X
Anaconda installation 18
IPython installation 26
panda installation 21
Python, installing 12-15
Python, installing from compressed
tarball 13, 14
**Markov Chain Monte Carlo Maximum
Likelihood**
reference link 224
Markov Chain Monte Carlo (MCMC) 224
matching operators
comparing, in R and pandas 271
**mathematical framework, Bayesian
statistics 199-202**
matplotlib
reference link 161
using, for plotting 158-161
maximum likelihood estimator (MLE) 221
mean 164, 165

P

pad method
  reference link 142
paired samples t-test 184
pandas
  about 5, 6
  benefits 7-9
  column name, specifying in 264
  cut() function 279
  data, examining 289, 290
  data structures 50
  data structures, URL 50
  downloading 19, 20
  features 6, 7
  file hierarchy 239, 240
  indexing 69
  installing 19, 20
  installing, from third-party vendor 16
  installing, on Linux 20
  installing, on Mac 21
  installing, on Windows 22
  isin() function 272
  logical subsetting 273
  melt() function 277
  missing values, handling 290-299
  multiple columns, selecting in 265
  split-apply-combine,
      implementing in 275
  URL 7
  used, for data analysis 289
  used, for preprocessing 289
pandas/compat
  about 250
  submodules 250
pandas/computation
  about 250
  submodules 250
pandas/core
  about 240
  submodules 240-243
pandas.DataFrame.any
  URL 95
pandas DataFrames
  versus R DataFrames 265
pandas.get_dummies() function 132

pandas installation, on Linux
  for Fedora 21
  for OpenSuse 21
  for Red Hat 21
  for Ubuntu/Debian 21
pandas installation, on Mac
  binary installation 22
  source installation 22
pandas installation, on Windows
  binary installation 22
  binary installation, URL 22
  Interactive Python (IPython) tool 24
  IPython Notebook 24, 25
  source installation 23
pandas/io
  about 240
  submodules 243, 245
pandas/rpy
  about 240
  reference link 249
  submodules 249
pandas series
  versus R lists 262, 263
pandas/sparse
  about 240
  reference link 247
  submodules 247
pandas/src 240
pandas/stats
  about 240
  submodules 247, 248
pandas/tools
  about 240
  submodules 246, 247
pandas/tseries
  submodules 251-253
pandas/util
  about 240
  submodules 248, 249
panel
  3D NumPy array, using with axis labels 65
  about 65
  items 65
  major_axis 65
  minor_axis 65

Thank you for buying
Mastering pandas

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
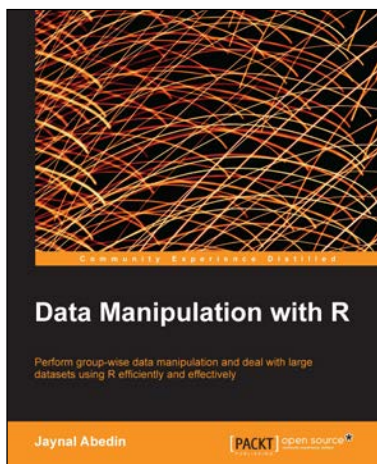
Data Manipulation with R

ISBN: 978-1-78328-109-1   Paperback: 102 pages

Perform group-wise data manipulation and deal with large datasets using R efficiently and effectively

1. Perform factor manipulation and string processing.

2. Learn group-wise data manipulation using plyr.

3. Handle large datasets, interact with database software, and manipulate data using sqldf.
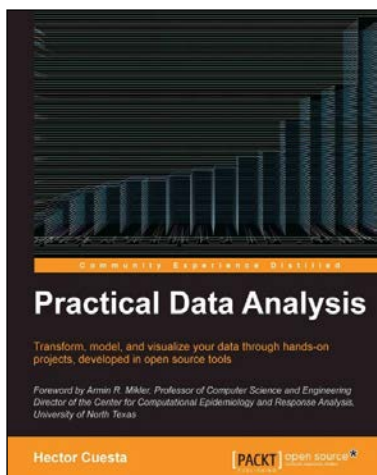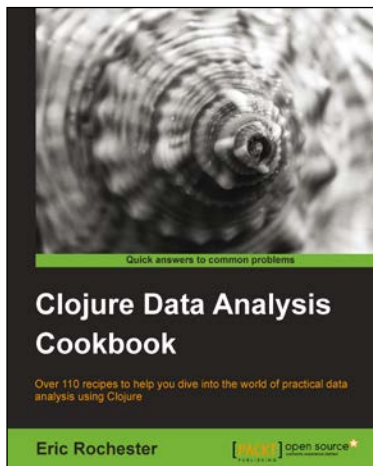
Practical Data Analysis

ISBN: 978-1-78328-099-5   Paperback: 360 pages

Transform, model, and visualize your data through hands-on projects, developed in open source tools

1. Explore how to analyze your data in various innovative ways and turn them into insight.

2. Learn to use the D3.js visualization tool for exploratory data analysis.

3. Understand how to work with graphs and social data analysis.

4. Discover how to perform advanced query techniques and run MapReduce on MongoDB.

Please check **www.PacktPub.com** for information on our titles
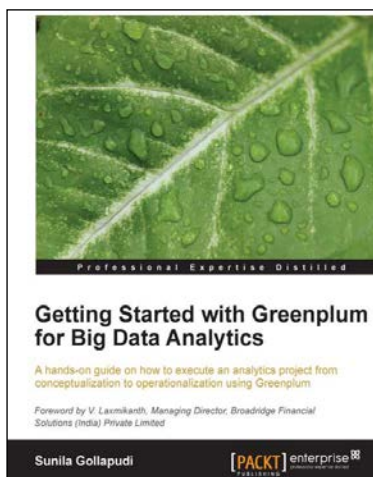
Clojure Data Analysis Cookbook

ISBN: 978-1-78216-264-3   Paperback: 342 pages

Over 110 recipes to help you dive into the world of practical data analysis using Clojure

1. Get a handle on the torrent of data the modern Internet has created.

2. Recipes for every stage from collection to analysis.

3. A practical approach to analyzing data to help you make informed decisions.

Getting Started with Greenplum for Big Data Analytics

ISBN: 978-1-78217-704-3   Paperback: 172 pages

A hands-on guide on how to execute an analytics project from conceptualization to operationalization using Greenplum

1. Explore the software components and appliance modules available in Greenplum.

2. Learn core Big Data architecture concepts and master data loading and processing patterns.

3. Understand Big Data problems and the data science lifecycle.

Please check **www.PacktPub.com** for information on our titles