

Michael Heydt

Learning pandas

Second Edition

High-performance data manipulation and analysis
in Python



Packt

<html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/REC-html40/loose.dtd">

Learning pandas

Second Edition

High-performance data manipulation and analysis in Python

Michael Heydt

Packt

BIRMINGHAM - MUMBAI

<html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/REC-html40/loose.dtd">

Learning pandas

Second Edition

Copyright © 2017 Packt Publishing All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2015

Second edition: June 2017

Production reference: 1300617

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78712-313-7

www.packtpub.com

Credits

Authors Michael Heydt	Copy Editors Safis Editing
Reviewers Sonali Dayal Nicola Rainiero	Project CoordinatorÂ Nidhi Joshi
Commissioning EditorÂ Â Amey Varangaonkar	ProofreaderÂ Safis Editing
Acquisition EditorÂ Tushar GuptaÂ	IndexerÂ Aishwarya GangawaneÂ

Content Development EditorÂ

Aishwarya Pandere

GraphicsÂ

Tania Dutta

Technical EditorÂ

Prasad Ramesh

Production CoordinatorÂ

Melwyn Dsa

Â ¢f

About the Author

Michael Heydt is a technologist, entrepreneur, and educator with decades of professional software development and financial and commodities trading experience. He has worked extensively on Wall Street specializing in the development of distributed, actor-based, high-performance, and high-availability trading systems. He is currently founder of Micro Trading Services, a company that focuses on creating cloud and micro service-based software solutions for finance and commodities trading. He holds a master's in science in mathematics and computer science from Drexel University, and an executive master's of technology management from the University of Pennsylvania School of Applied Science and the Wharton School of Business.

I would really like to thank the team at Packt for continuously pushing me to create and revise this and my other books. I would also like to greatly thank my family for putting up with me disappearing for months on end during my sparse free time to indulge in creating this content. They are my true inspiration.

About the Reviewers

Sonali Dayal is a freelance data scientist in the San Francisco Bay Area. Her work on building analytical models and data pipelines influences major product and financial decisions for clients. Previously, she has worked as a freelance software and data science engineer for early stage startups, where she built supervised and unsupervised machine learning models, as well as interactive data analytics dashboards. She received her BS in biochemistry from Virginia Tech in 2011.

I'd like to thank the team at Packt for the opportunity to review this book and their support throughout the process.

Nicola Rainiero is a civil geotechnical engineer with a background in the construction industry as a self-employed designer engineer. He is also specialized in renewable energy and has collaborated with the Sant Anna University of Pisa for two European projects, REGEOCITIES and PRISCA, using qualitative and quantitative data analysis techniques.

He has the ambition to simplify his work with open software, using and developing new ones. Sometimes obtaining good results, other less good.

A special thanks to Packt Publishing for this opportunity to participate in the review of this book. I thank my family, especially my parents, for their physical and moral support.

For support files and downloads related to your book, please visitÂ 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 atÂ www.PacktPub.comand as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us atÂ service@packtpub.com for more details. AtÂ 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://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1787123138>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

1. pandas and Data Analysis

[Introducing pandas](#)

[Data manipulation, analysis, science, and pandas](#)

[Data manipulation](#)

[Data analysis](#)

[Data science](#)

[Where does pandas fit?](#)

[The process of data analysis](#)

[The process](#)

[Ideation](#)

[Retrieval](#)

[Preparation](#)

[Exploration](#)

[Modeling](#)

[Presentation](#)

[Reproduction](#)

[A note on being iterative and agile](#)

[Relating the book to the process](#)

Concepts of data and analysis in our tour of pandas

Types of data

Structured

Unstructured

Semi-structured

Variables

Categorical

Continuous

Discrete

Time series data

General concepts of analysis and statistics

Quantitative versus qualitative data/analysis

Single and multivariate analysis

Descriptive statistics

Inferential statistics

Stochastic models

Probability and Bayesian statistics

Correlation

Regression

Other Python libraries of value with pandas

Numeric and scientific computing - NumPy and SciPy

Statistical analysis – StatsModels

Machine learning – scikit-learn

PyMC - stochastic Bayesian modeling

Data visualization - matplotlib and seaborn

Matplotlib

Seaborn

Summary

2. Up and Running with pandas

Installation of Anaconda

IPython and Jupyter Notebook

IPython

Jupyter Notebook

Introducing the pandas Series and DataFrame

Importing pandas

The pandas Series

The pandas DataFrame

Loading data from files into a DataFrame

Visualization

Summary

3. Representing Univariate Data with the Series

Configuring pandas

Creating a Series

Creating a Series using Python lists and dictionaries

Creation using NumPy functions

Creation using a scalar value

The .index and .values properties

The size and shape of a Series

Specifying an index at creation

Heads, tails, and takes

Retrieving values in a Series by label or position

Lookup by label using the [] operator and the .ix[] property

Explicit lookup by position with .iloc[]

Explicit lookup by labels with .loc[]

Slicing a Series into subsets

Alignment via index labels

Performing Boolean selection

Re-indexing a Series

Modifying a Series in-place

Summary

4. Representing Tabular and Multivariate Data with the DataFrame

Configuring pandas

Creating DataFrame objects

Creating a DataFrame using NumPy function results

Creating a DataFrame using a Python dictionary and pandas Series objects

[Creating a DataFrame from a CSV file](#)

[Accessing data within a DataFrame](#)

[Selecting the columns of a DataFrame](#)

[Selecting rows of a DataFrame](#)

[Scalar lookup by label or location using .at\[\] and .iat\[\]](#)

[Slicing using the \[\] operator](#)

[Selecting rows using Boolean selection](#)

[Selecting across both rows and columns](#)

[Summary](#)

5. Manipulating DataFrame Structure

[Configuring pandas](#)

[Renaming columns](#)

[Adding new columns with \[\] and .insert\(\)](#)

[Adding columns through enlargement](#)

[Adding columns using concatenation](#)

[Reordering columns](#)

[Replacing the contents of a column](#)

[Deleting columns](#)

[Appending new rows](#)

[Concatenating rows](#)

[Adding and replacing rows via enlargement](#)

[Removing rows using .drop\(\)](#)

[Removing rows using Boolean selection](#)

[Removing rows using a slice](#)

[Summary](#)

6. Indexing Data

[Configuring pandas](#)

[The importance of indexes](#)

[The pandas index types](#)

[The fundamental type - Index](#)

[Integer index labels using Int64Index and RangeIndex](#)

[Floating-point labels using Float64Index](#)

[Representing discrete intervals using IntervalIndex](#)

[Categorical values as an index - CategoricalIndex](#)

[Indexing by date and time using DatetimeIndex](#)

[Indexing periods of time using PeriodIndex](#)

[Working with Indexes](#)

[Creating and using an index with a Series or DataFrame](#)

[Selecting values using an index](#)

[Moving data to and from the index](#)

[Reindexing a pandas object](#)

[Hierarchical indexing](#)

[Summary](#)

[7. Categorical Data](#)

[Configuring pandas](#)

[Creating Categoricals](#)

[Renaming categories](#)

[Appending new categories](#)

[Removing categories](#)

[Removing unused categories](#)

[Setting categories](#)

[Descriptive information of a Categorical](#)

[Munging school grades](#)

[Summary](#)

[8. Numerical and Statistical Methods](#)

[Configuring pandas](#)

[Performing numerical methods on pandas objects](#)

[Performing arithmetic on a DataFrame or Series](#)

[Getting the counts of values](#)

[Determining unique values \(and their counts\)](#)

Finding minimum and maximum values

Locating the n-smallest and n-largest values

Calculating accumulated values

Performing statistical processes on pandas objects

Retrieving summary descriptive statistics

Measuring central tendency: mean, median, and mode

Calculating the mean

Finding the median

Determining the mode

Calculating variance and standard deviation

Measuring variance

Finding the standard deviation

Determining covariance and correlation

Calculating covariance

Determining correlation

Performing discretization and quantiling of data

Calculating the rank of values

Calculating the percent change at each sample of a series

Performing moving-window operations

Executing random sampling of data

Summary

9. Accessing Data

Configuring pandas

Working with CSV and text/tabular format data

Examining the sample CSV data set

Reading a CSV file into a DataFrame

Specifying the index column when reading a CSV file

Data type inference and specification

Specifying column names

Specifying specific columns to load

Saving DataFrame to a CSV file

Working with general field-delimited data

Handling variants of formats in field-delimited data

Reading and writing data in Excel format

Reading and writing JSON files

Reading HTML data from the web

Reading and writing HDF5 format files

Accessing CSV data on the web

Reading and writing from/to SQL databases

Reading data from remote data services

Reading stock data from Yahoo! and Google Finance

Retrieving options data from Google Finance

Reading economic data from the Federal Reserve Bank of St. Louis

Accessing Kenneth French's data

Reading from the World Bank

Summary

10. Tidying Up Your Data

Configuring pandas

What is tidying your data?

How to work with missing data

Determining NaN values in pandas objects

Selecting out or dropping missing data

Handling of NaN values in mathematical operations

Filling in missing data

Forward and backward filling of missing values

Filling using index labels

Performing interpolation of missing values

Handling duplicate data

Transforming data

Mapping data into different values

Replacing values

Applying functions to transform data

Summary

11. Combining, Relating, and Reshaping Data

Configuring pandas

Concatenating data in multiple objects

Understanding the default semantics of concatenation

Switching axes of alignment

Specifying join type

Appending versus concatenation

Ignoring the index labels

Merging and joining data

Merging data from multiple pandas objects

Specifying the join semantics of a merge operation

Pivoting data to and from value and indexes

Stacking and unstacking

Stacking using non-hierarchical indexes

Unstacking using hierarchical indexes

Melting data to and from long and wide format

Performance benefits of stacked data

Summary

12. Data Aggregation

Configuring pandas

The split, apply, and combine (SAC) pattern

Data for the examples

Splitting data

Grouping by a single column's values

Accessing the results of a grouping

Grouping using multiple columns

Grouping using index levels

Applying aggregate functions, transforms, and filters

Applying aggregation functions to groups

Transforming groups of data

The general process of transformation

Filling missing values with the mean of the group

Calculating normalized z-scores with a transformation

Filtering groups from aggregation

Summary

13. Time-Series Modelling

Setting up the IPython notebook

Representation of dates, time, and intervals

The datetime, day, and time objects

Representing a point in time with a Timestamp

Using a Timedelta to represent a time interval

Introducing time-series data

Indexing using DatetimeIndex

Creating time-series with specific frequencies

Calculating new dates using offsets

Representing data intervals with date offsets

Anchored offsets

Representing durations of time using Period

Modelling an interval of time with a Period

Indexing using the PeriodIndex

Handling holidays using calendars

Normalizing timestamps using time zones

Manipulating time-series data

Shifting and lagging

Performing frequency conversion on a time-series

Up and down resampling of a time-series

Time-series moving-window operations

Summary

14. Visualization

Configuring pandas

Plotting basics with pandas

Creating time-series charts

Adorning and styling your time-series plot

Adding a title and changing axes labels

Specifying the legend content and position

Specifying line colors, styles, thickness, and markers

Specifying tick mark locations and tick labels

Formatting axes' tick date labels using formatters

Common plots used in statistical analyses

Showing relative differences with bar plots

Picturing distributions of data with histograms

Depicting distributions of categorical data with box and whisker charts

Demonstrating cumulative totals with area plots

Relationships between two variables with scatter plots

Estimates of distribution with the kernel density plot

Correlations between multiple variables with the scatter plot matrix

Strengths of relationships in multiple variables with heatmaps

Manually rendering multiple plots in a single chart

Summary

15. Historical Stock Price Analysis

Setting up the IPython notebook

Obtaining and organizing stock data from Google

Plotting time-series prices

Plotting volume-series data

Calculating the simple daily percentage change in closing price

Calculating simple daily cumulative returns of a stock

Resampling data from daily to monthly returns

Analyzing distribution of returns

Performing a moving-average calculation

Comparison of average daily returns across stocks

Correlation of stocks based on the daily percentage change of the closing price

Calculating the volatility of stocks

Determining risk relative to expected returns

Summary

Preface

Pandas is a popular Python package used for practical, real-world data analysis. It provides efficient, fast, and high-performance data structures that make data exploration and analysis very easy. This learner's guide will help you through a comprehensive set of features provided by the pandas library to perform efficient data manipulation and analysis.

What this book covers

[Chapter 1](#), *pandas and Data Analysis*, is a hands-on introduction to the key features of pandas. The idea of this chapter is to provide some context for using pandas in the context of statistics and data science. The chapter will get into several concepts in data science and show how they are supported by pandas. This will set a context for each of the subsequent chapters, mentioning each chapter relates to both data science and data science processes.

[Chapter 2](#), *Up and Running with pandas*, instructs the reader on obtain and install pandas, and to get introduce a few of the basic concepts in pandas. We will also look at how the examples are presented using iPython and Jupyter notebook.

[Chapter 3](#), *Representing Univariate Data with the Series*, walks the reader through the use of the pandas Series, which provides 1-dimensional, indexed data representations. The reader will learn about how to create Series objects and how to manipulate data held within. They will also learn about indexes and alignment of data, and about how the Series can be used to slice data.

[Chapter 4](#), *Representing Tabular and Multivariate Data with the DataFrame*, walks the reader through the basic use of the pandas DataFrame, which provides and indexes multivariate data representations. This chapter will instruct the reader to be able to create DataFrame objects using various sets of static data, and how to perform selection of specific columns and rows within. Complex queries, manipulation, and indexing will be now handled in the following chapter.

[Chapter 5](#), *Manipulation and Indexing of DataFrame objects*, expands on the previous chapter and instructs you on how to perform more complex manipulations of a DataFrame. We start by learning how to add, remove, and delete columns and rows; modify data within a DataFrame (or created a modified copy); perform calculations on data within; create hierarchical indexes; and also calculate common statistical results upon DataFrame contents.

[Chapter 6](#), *Indexing Data*, shows how data can be loaded and saved from external sources into both Series and DataFrame objects. The chapter also covers data access from multiple sources such as files, http servers, database systems, and web services. Also covered is the processing of data in CSV, HTML, and JSON formats.

[Chapter 7](#), *Categorical Data*, instructs the reader on how to use the various tools provided by pandas for managing dirty and missing data.

[Chapter 8](#), *Numerical and Statistical Methods*, covers various techniques for combining, splitting, joining, and merging of data located in multiple pandas objects, and then demonstrates on how to reshape data using concepts such as pivots, stacking, and melting.

[Chapter 9](#), *Accessing Data*, talks about grouping and performing aggregate data analysis. In pandas, this is often referred to as the split-apply-combine pattern. The reader will learn about using this pattern to group data in various different configurations and also apply aggregate functions to calculate results upon each group of data.

[Chapter 10](#), *Tidying Up Your Data*, explains how to organize data in a tidy form, that is usable for data analysis.

[Chapter 11](#), *Combining, Relating and Reshaping Data*, tells the readers how they can take data in multiple pandas objects and combine them, through concepts such as joins, merges and concatenation.

[Chapter 12](#), *Data Aggregation*, dives into the integration of pandas with matplotlib to visualize pandas data. The chapter will demonstrate how to present many common statistical and financial data visualizations including bar charts, histograms, scatter plots, area plots, density plots, and heat maps.

[Chapter 13](#), *Time-Series Modeling*, covers representing time series data in pandas. This chapter will cover the extensive capabilities provided by pandas for facilitating analysis of time series data.

[Chapter 14](#), *Visualization*, teaches you how to create data visualizations based upon data stored in pandas data structures. We start with the basics learning, how to create a simple chart from data and control several of the attributes of the chart (such as legends, labels, and colors). We examine the creation of several common types of plot used to represent different types of data that are use those plot types to convey meaning in the underlying data. We also learn how to integrate pandas with D3.js so that we can create rich web-based visualizations.

[Chapter 15](#), *Historical Stock Price Analysis*, shows you how to apply pandas to basic financial problems. It will focus on data obtained from Yahoo! Finance, and will demonstrate a number of financial concepts in financial data such as calculating returns, moving averages, volatility, and several other concepts. The student will also learns how to apply data visualization to these financial concepts.

What you need for this book

This book assumes some familiarity with programming concepts, but those without programming experience, or specifically Python programming experience, will be comfortable with the examples as they focus on pandas constructs more than Python or programming. The examples are based on Anaconda Python 2.7 and pandas 0.15.1. If you do not have either installed, guidance will be given in [Chapter 2, Up and Running with pandas](#), regarding installing pandas on installing both on Windows, OSX, and Ubuntu systems. For those not interested in installing any software, instruction is also given on using the Warkari.io online Python data analysis service.

Who this book is for

This book is ideal for data scientists, data analysts, and Python programmers who want to plunge into data analysis using pandas, and anyone curious about analyzing data. Some knowledge of statistics and programming will help you to get the most out of this book but that's not strictly required. Prior exposure to pandas is also not required.

Conventions

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

Code words in text are shown as follows: "This information can be easily imported into DataFrame using the `pd.read_csv()` function as follows."

A block of code entered in a Python interpreter is set as follows:

```
import pandas as pd
df = pd.DataFrame.from_items([('column1', [1, 2, 3])])
print(df)
```

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

```
| mh@ubuntu:~/Downloads$ chmod +x Anaconda-2.1.0-Linux-x86_64.sh
| mh@ubuntu:~/Downloads$ ./Anaconda-2.1.0-Linux-x86_64.sh
```

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 shortcuts in this book are based on the `Mac OS X 10.5+` scheme."



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 for this book from your account at <http://www.packtpub.com>. 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 download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Learning-Pandas-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

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.

pandas and Data Analysis

Welcome to *Learning pandas*! In this book, we will go on a journey that will see us learning pandas, an open source data analysis library for the Python programming language. The pandas library provides high-performance and easy-to-use data structures and analysis tools built with Python. pandas brings to Python many good things from the statistical programming language R, specifically **data frame** objects and R packages such as `plyr` and `reshape2`, and places them in a single library that you can use from within Python.

In this first chapter, we will take the time to understand pandas and how it fits into the bigger picture of data analysis. This will give the reader who is interested in pandas a feeling for its place in the bigger picture of data analysis instead of having a complete focus on the details of using pandas. The goal is that while learning pandas you also learn why those features exist in support of performing data analysis tasks.

So, let's jump in. In this chapter, we will cover:

- What pandas is, why it was created, and what it gives you
- How pandas relates to data analysis and data science
- The processes involved in data analysis and how it is supported by pandas
- General concepts of data and analytics
- Basic concepts of data analysis and statistical analysis
- Types of data and their applicability to pandas
- Other libraries in the Python ecosystem that you will likely use with pandas

Introducing pandas

pandas is a Python library containing high-level data structures and tools that have been created to help Python programmers to perform powerful data analysis. The ultimate purpose of pandas is to help you quickly discover **information** in data, with information being defined as an underlying meaning.

The development of pandas was begun in 2008 by Wes McKinney; it was open sourced in 2009. pandas is currently supported and actively developed by various organizations and contributors.

pandas was initially designed with finance in mind specifically with its ability around time series data manipulation and processing historical stock information. The processing of financial information has many challenges, the following being a few:

- Representing security data, such as a stock's price, as it changes over time
- Matching the measurement of multiple streams of data at identical times
- Determining the relationship (correlation) of two or more streams of data
- Representing times and dates as first-class entities
- Converting the period of samples of data, either up or down

To do this processing, a tool was needed that allows us to retrieve, index, clean and tidy, reshape, combine, slice, and perform various analyses on both single- and multidimensional data, including heterogeneous-typed data that is automatically aligned along a set of common index labels. This is where pandas comes in, having been created with many useful and powerful features such as the following:

- Fast and efficient `Series` and `DataFrame` objects for data manipulation with integrated indexing
- Intelligent data alignment using indexes and labels
- Integrated handling of missing data
- Facilities for converting messy data into orderly data (tidying)
- Built-in tools for reading and writing data between in-memory data structures and files, databases, and web services
- The ability to process data stored in many common formats such as CSV, Excel, HDF5, and JSON
- Flexible reshaping and pivoting of sets of data
- Smart label-based slicing, fancy indexing, and subsetting of large datasets
- Columns can be inserted and deleted from data structures for size mutability
- Aggregating or transforming data with a powerful data grouping facility to perform split-apply-combine on datasets
- High-performance merging and joining of datasets
- Hierarchical indexing facilitating working with high-dimensional data in a lower-dimensional data structure
- Extensive features for time series data, including date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting, and lagging
- Highly optimized for performance, with critical code paths written in **Cython** or C

The robust feature set, combined with its seamless integration with Python and other tools within the Python ecosystem, has given pandas wide adoption in many domains. It is in use in a wide variety of

academic and commercial domains, including finance, neurosciences, economics, statistics, advertising, and web analytic. It has become one of the most preferred tools for data scientists to represent data for manipulation and analysis.

Python has long been exceptional for data munging and preparation, but less so for data analysis and modeling. pandas helps fill this gap, enabling you to carry out your entire data analysis workflow in Python without having to switch to a more domain -specific language such as R. This is very important, as those familiar with Python, a more generalized programming language than R (more a statistical package), gain many data representation and manipulation features of R while remaining entirely within an incredibly rich Python ecosystem.

Combined with IPython, Jupyter notebooks, and a wide range of other libraries, the environment for performing data analysis in Python excels in performance, productivity, and the ability to collaborate, compared to many other tools. This has led to the widespread adoption of pandas by many users in many industries.

Data manipulation, analysis, science, and pandas

We live in a world in which massive amounts of data are produced and stored every day. This data comes from a plethora of information systems, devices, and sensors. Almost everything you do, and items you use to do it, produces data which can be, or is, captured.

This has been greatly enabled by the ubiquitous nature of services that are connected to networks, and by the great increases in data storage facilities; this, combined with the ever-decreasing cost of storage, has made capturing and storing even the most trivial of data effective.

This has led to massive amounts of data being piled up and ready for access. But this data is spread out all over cyber-space, and is cannot actually be referred to as **information**. It tends to be a collected collection of the recording of events, whether financial, of your interactions with social networks, or of your personal health monitor tracking your heartbeat throughout the day. This data is stored in all kinds of formats, is located in scattered places, and beyond its raw nature does give much insight.

Logically, the overall process can be broken into three major areas of discipline:

- Data manipulation
- Data analysis
- Data science

These three disciplines can and do have a lot of overlap. Where each ends and the others begin is open to interpretation. For the purposes of this book we will define each as in the following sections.

Data manipulation

Data is distributed all over the planet. It is stored in different formats. It has widely varied levels of quality. Because of this there is a need for tools and processes for pulling data together and into a form that can be used for decision making. This requires many different tasks and capabilities from a tool that manipulates data in preparation for analysis. The features needed from such a tool include:

- Programmability for reuse and sharing
- Access to data from external sources
- Storing data locally
- Indexing data for efficient retrieval
- Alignment of data in different sets based upon attributes
- Combining data in different sets
- Transformation of data into other representations
- Cleaning data from cruft
- Effective handling of bad data
- Grouping data into common baskets
- Aggregation of data of like characteristics
- Application of functions to calculate meaning or perform transformations
- Query and slicing to explore pieces of the whole
- Restructuring into other forms
- Modeling distinct categories of data such as categorical, continuous, discrete, and time series
- Resampling data to different frequencies

There are many data manipulation tools in existence. Each differs in support for the items on this list, how they are deployed, and how they are utilized by their users. These tools include relational databases (SQL Server, Oracle), spreadsheets (Excel), event processing systems (such as Spark), and more generic tools such as R and pandas.

Data analysis

Data analysis is the process of creating meaning from data. Data with quantified meaning is often called **information**. Data analysis is the process of creating information from data through the creation of data models and mathematics to find patterns. It often overlaps data manipulation and the distinction between the two is not always clear. Many data manipulation tools also contain analyses functions, and data analysis tools often provide data manipulation capabilities.

Data science

Data science is the process of using statistics and data analysis processes to create an understanding of **phenomena** within data. Data science usually starts with information and applies a more complex domain-based analysis to the information. These domains span many fields such as mathematics, statistics, information science, computer science, machine learning, classification, cluster analysis, data mining, databases, and visualization. Data science is multidisciplinary. Its methods of domain analysis are often very different and specific to a specific domain.

Where does pandas fit?

pandas first and foremost excels in data manipulation. All of the needs itemized earlier will be covered in this book using pandas. This is the core of pandas and is most of what we will focus on in this book.

It is worth noting that pandas has a specific design goal: emphasizing data

But pandas does provide several features for performing data analysis. These capabilities typically revolve around descriptive statistics and functions required for finance such as correlations.

Therefore, pandas itself is not a data science toolkit. It is more of a manipulation tool with some analysis capabilities. pandas explicitly leaves complex statistical, financial, and other types of analyses to other Python libraries, such as SciPy, NumPy, scikit-learn, and leans upon graphics libraries such as **matplotlib** and **ggvis** for data visualization.

This focus is actually a strength of pandas over other languages such as R as pandas applications are able to leverage an extensive network of robust Python frameworks already built and tested elsewhere by the Python community.

The process of data analysis

The primary goal of this book is to thoroughly teach you how to use pandas to manipulate data. But there is a secondary, and perhaps no less important, goal of showing how pandas fits into the processes that a data analyst/scientist performs in everyday life.

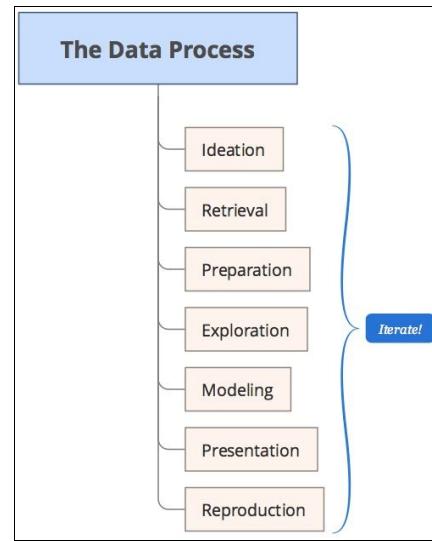
One description of the steps involved in the process of data analysis is given on the pandas web site:

- Munging and cleaning data
- Analyzing/modeling
- Organization into a form suitable for communication

This small list is a good initial definition, but it fails to cover the overall scope of the process and why many features implemented in pandas were created. The following expands upon this process and sets the framework for what is to come throughout this journey.

The process

The proposed process is one that will be referred to as **The Data Process** and is represented in the following diagram:



This process sets up a framework for defining logical steps that are taken in working with data. For now, let's take a quick look at each of these steps in the process and some of the tasks that you as a data analyst using pandas will perform.



It is important to understand that this is not purely a linear process. It is best done in a highly interactive and agile/iterative manner.

Ideation

The first step in any data problem is to identify what it is you want to figure out. This is referred to as **ideation**, of coming up with an idea of what we want to do and prove. Ideation generally relates to hypothesizing about patterns in data that can be used to make intelligent decisions.

These decisions are often within the context of a business, but also within other disciplines such as the sciences and research. The in-vogue thing right now is understanding the operations of businesses, as there are often copious amounts of money to be made in understanding data.

But what kinds of decision are we typically looking to make? The following are several questions for which answers are commonly asked:

- Why did something happen?
- Can we predict the future using historical data?
- How can I optimize operations in the future?

This list is by no means exhaustive, but it does cover a sizable percentage of the reasons why anyone undertakes these endeavors. To get answers to these questions, one must be involved with collecting and understanding data relative to the problem. This involves defining what data is going to be researched, what the benefit is of the research, how the data is going to be obtained, what the success criteria are, and how the information is going to be eventually communicated.

pandas itself does not provide tools to assist in ideation. But once you have gained understanding and skill in using pandas, you will naturally realize how pandas will help you in being able to formulate ideas. This is because you will be armed with a powerful tool you can use to frame many complicated hypotheses.

Retrieval

Once you have an idea you must then find data to try and support your hypothesis. This data can come from within your organization or from external data providers. This data normally is provided as archived data or can be provided in real-time (although pandas is not well known for being a real-time data processing tool).

Data is often very raw, even if obtained from data sources that you have created or from within your organization. Being raw means that the data can be disorganized, may be in various formats, and erroneous; relative to supporting your analysis, it may be incomplete and need manual augmentation.

There is a lot of free data in the world. Much data is not free and actually costs significant amounts of money to obtain. Some is freely available with public APIs, and the others by subscription. Data you pay for is often cleaner, but this is not always the case.

In either case, pandas provides a robust and easy-to-use set of tools for retrieving data from various sources and that may be in many different formats. pandas also gives us the ability to not only retrieve data, but to also provide an initial structuring of the data via pandas data structures without needing to manually create complex coding, which may be required in other tools or programming languages.

Preparation

During preparation, raw data is made ready for exploration. This preparation is often a very interesting process. It is very frequently the case that data from is fraught with all kinds of issues related to quality. You will likely spend a lot of time dealing with these quality issues, and often this is a very non-trivial amount of time.

Why? Well there are a number of reasons:

- The data is simply incorrect
- Parts of the dataset are missing
- Data is not represented using measurements appropriate for your analysis
- The data is in formats not convenient for your analysis
- Data is at a level of detail not appropriate for your analysis
- Not all the fields you need are available from a single source
- The representation of data differs depending upon the provider

The preparation process focuses on solving these issues. pandas provides many great facilities for preparing data, often referred to as **tidying up** data. These facilities include intelligent means of handling missing data, converting data types, using format conversion, changing frequencies of measurements, joining data from multiple sets of data, mapping/converting symbols into shared representations, and grouping data, among many others. We will cover all of these in depth.

Exploration

Exploration involves being able to interactively slice and dice your data to try and make quick discoveries. Exploration can include various tasks such as:

- Examining how variables relate to each other
- Determining how the data is distributed
- Finding and excluding outliers
- Creating quick visualizations
- Quickly creating new data representations or models to feed into more permanent and detailed modeling processes

Exploration is one of the great strengths of pandas. While exploration can be performed in most programming languages, each has its own level of **ceremony**—how much non-exploratory effort must be performed—before actually getting to discoveries.

When used with the **read-eval-print-loop (REPL)** nature of IPython and/or Jupyter notebooks, pandas creates an exploratory environment that is almost free of ceremony. The expressiveness of the syntax of pandas lets you describe complex data manipulation constructs succinctly, and the result of every action you take upon your data is immediately presented for your inspection. This allows you to quickly determine the validity of the action you just took without having to recompile and completely rerun your programs.

Modeling

In the modeling stage you formalize your discoveries found during exploration into an explicit explanation of the steps and data structures required to get to the desired meaning contained within your data. This is the **model**, a combination of both data structures as well as steps in code to get from the raw data to your information and conclusions.

The modeling process is iterative where, through an exploration of the data, you select the variables required to support your analysis, organize the variables for input to analytical processes, execute the model, and determine how well the model supports your original assumptions. It can include a formal modeling of the structure of the data, but can also combine techniques from various analytic domains such as (and not limited to) statistics, machine learning, and operations research.

To facilitate this, pandas provides extensive data modeling facilities. It is in this step that you will move more from exploring your data, to formalizing the data model in `DataFrame` objects, and ensuring the processes to create these models are succinct. Additionally, by being based in Python, you get to use its full power to create programs to automate the process from beginning to end. The models you create are executable.

From an analytic perspective, pandas provides several capabilities, most notably integrated support for descriptive statistics, which can get you to your goal for many types of problems. And because pandas is Python-based, if you need more advanced analytic capabilities, it is very easy to integrate with other parts of the extensive Python scientific environment.

Presentation

The penultimate step of the process is presenting your findings to others, typically in the form of a report or presentation. You will want to create a persuasive and thorough explanation of your solution. This can often be done using various plotting tools in Python and manually creating a presentation.

Jupyter notebooks are a powerful tool in creating presentations for your analyses with pandas. These notebooks provide a means of both executing code and providing rich markdown capabilities to annotate and describe the execution at multiple points in the application. These can be used to create very effective, executable presentations that are visually rich with pieces of code, stylized text, and graphics.

We will explore Jupyter notebooks briefly in Chapter 2, Up and Running with pandas.



Reproduction

An important piece of research is sharing and making your research reproducible. It is often said that if other researchers cannot reproduce your experiment and results, then you didn't prove a thing.

Fortunately, for you, by having used pandas and Python, you will be able to easily make your analysis reproducible. This can be done by sharing the Python code that drives your pandas code, as well as the data.

Jupyter notebooks also provide a convenient means of packaging both the code and application in a means that can be easily shared with anyone else with a Jupyter Notebook installation. And there are many free, and secure, sharing sites on the internet that allow you to either create or deploy your Jupyter notebooks for sharing.

A note on being iterative and agile

Something very important to understand about data manipulation, analysis, and science is that it is an iterative process. Although there is a natural forward flow along the stages previously discussed, you will end up going forwards and backwards in the process. For instance, while in the exploration phase you may identify anomalies in the data that relate to data purity issues from the preparation stage, and need to go back and rectify those issues.

This is part of the fun of the process. You are on an adventure to solve your initial problem, all the while gaining incremental insights about the data you are working with. These insights may lead you to ask new questions, to more exact questions, or to a realization that your initial questions were not the actual questions that needed to be asked. The process is truly a journey and not necessarily the destination.

Relating the book to the process

The following gives a quick mapping of the steps in the process to where you will learn about them in this book. Do not fret if the steps that are earlier in the process are in later chapters. The book will walk you through this in a logical progression for learning pandas, and you can refer back from the chapters to the relevant stage in the process.

Step in process	Place
Ideation	Ideation is the creative process in data science. You need to have the idea. The fact that you are reading this qualifies you as you must be looking to analyze some data, and want to in the future.
Retrieval	Retrieval of data is primarily covered in Chapter 9, Accessing Data .
Preparation	Preparation of data is primarily covered in Chapter 10, Tidying Up your Data , but it is also a common thread running through most of the chapters.
Exploration	Exploration spans Chapter 3, Representing Univariate Data with the Series , through Chapter 15, Historical Stock Price Analysis , so most of the chapters of the book. But the most focused chapters for exploration are Chapter 14, Visualization and Chapter 15, Historical Stock Price Analysis , in both of which we begin to see the results of data analysis.
Modeling	Modeling has its focus in Chapter 3, Representing Univariate Data with the pandas Series , and Chapter 4, Representing Tabular and Multivariate Data with the DataFrame with the pandas <code>DataFrame</code> , and also Chapter 11, Combining, Relating, and Reshaping Data through Chapter 13, Time-Series Modelling , and with a specific focus towards finance in Chapter 15, Historical Stock Price Analysis .
Presentation	Presentation is the primary purpose of Chapter 14, Visualization .
Reproduction	Reproduction flows throughout the book, as the examples are provided as Jupyter notebooks. By working in notebooks, you are by default using a tool for reproduction and have the ability to share notebooks in various ways.

Concepts of data and analysis in our tour of pandas

When learning pandas and data analysis you will come across many concepts in data, modeling and analysis. Let's examine several of these concepts and how they relate to pandas.

Types of data

Working with data in the wild you will come across several broad categories of data that will need to be coerced into pandas data structures. They are important to understand as the tools required to work with each type vary.

pandas is inherently used for manipulating structured data but provides several tools for facilitating the conversion of non-structured data into a means we can manipulate.

Structured

Structured data is any type of data that is organized as fixed fields within a record or file, such as data in relational databases and spreadsheets. Structured data depends upon a data model, which is the defined organization and meaning of the data and often how the data should be processed. This includes specifying the type of the data (integer, float, string, and so on), and any restrictions on the data, such as the number of characters, maximum and minimum values, or a restriction to a certain set of values.

Structured data is the type of data that pandas is designed to utilize. As we will see first with the `Series` and then with the `DataFrame`, pandas organizes structured data into one or more columns of data, each of a single and specific data type, and then a series of zero or more rows of data.

Unstructured

Unstructured data is data that is without any defined organization and which specifically does not break down into stringently defined columns of specific types. This can consist of many types of information such as photos and graphic images, videos, streaming sensor data, web pages, PDF files, PowerPoint presentations, emails, blog entries, wikis, and word processing documents.

While pandas does not manipulate unstructured data directly, it provides a number of facilities to extract structured data from unstructured sources. As a specific example that we will examine, pandas has tools to retrieve web pages and extract specific pieces of content into a `DataFrame`.

Semi-structured

Semi-structured data fits in between unstructured. It can be considered a type of structured data, but lacks the strict data model structure. JSON is a form of semi-structured data. While good JSON will have a defined format, there is no specific schema for data that is always strictly enforced. Much of the time, the data will be in a repeatable pattern that can be easily converted into structured data types like the pandas `DataFrame`, but the process may need some guidance from you to specify or coerce data types.

Variables

When modeling data in pandas, we will be modeling one or more variables and looking to find statistical meaning amongst the values or across multiple variables. This definition of a variable is not in the sense of a variable in a programming language but one of statistical variables.

A variable is any characteristic, number, or quantity that can be measured or counted. A variable is so named because the value may vary between data units in a population and may change in value over time. Stock value, age, sex, business income and expenses, country of birth, capital expenditure, class grades, eye color, and vehicle type are examples of variables.

There are several broad types of statistical variables that we will come across when using pandas:

- Categorical
- Continuous
- Discrete

Categorical

A **categorical** variable is a variable that can take on one of a limited, and usually fixed, number of possible values. Each of the possible values is often referred to as a **level**. Categorical variables in pandas are represented by `Categoricals`, a pandas data type which corresponds to categorical variables in statistics. Examples of categorical variables are gender, social class, blood types, country affiliations, observation time, or ratings such as Likert scales.

Continuous

A **continuous** variable is a variable that can take on infinitely many (an uncountable number of) values. Observations can take any value between a certain set of real numbers. Examples of continuous variables include height, time, and temperature. Continuous variables in pandas are represented by either float or integer types (native to Python), typically in collections that represent multiple samplings of the specific variable.

Discrete

A **discrete** variable is a variable where the values are based on a count from a set of distinct whole values. A discrete variable cannot be a fractional value between any two variables. Examples of discrete variables include the number of registered cars, number of business locations, and number of children in a family, all of which measure whole units (for example 1, 2, or 3 children). Discrete variables are normally represented in pandas by integers (or occasionally floats), again normally in collections of two or more samplings of a variable.

Time series data

Time series data is a first-class entity within pandas. Time adds an important, extra dimension to samples of variables within pandas. Often variables are independent of the time they were sampled at; that is, the time at which they are sampled is not important. But in many cases they are. A time series forms a sample of a discrete variable at specific time intervals, where the observations have a natural temporal ordering.

A stochastic model for a time series will generally reflect the fact that observations close together in time will be more closely related than observations that are further apart. Time series models will often make use of the natural one-way ordering of time so that values for a given period will be expressed as deriving in some way from past values rather than from future values.

A common scenario with pandas is financial data where a variable represents the value of a stock as it changes at regular intervals throughout the day. We often want to determine changes in the rate of change of the price at specific intervals. We may also want to correlate the price of multiple stocks across specific intervals of time.

This is such an important and robust capability in pandas that we will spend an entire chapter examining the concept.

General concepts of analysis and statistics

In this text, we will only approach the periphery of statistics and the technical processes of data analysis. But several analytical concepts of are worth noting, some of which have implementations directly created within pandas. Others will need to rely on other libraries such as SciPy, but you may also come across them while working with pandas so an initial shout-out is valuable.

Quantitative versus qualitative data/analysis

Qualitative analysis is the scientific study of data that can be observed but cannot be measured. It focuses on cataloging the qualities of data. Examples of qualitative data can be:

- The softness of your skin
- How elegantly someone runs

Quantitative analysis is the study of actual values within data, with real measurements of items presented as data. Normally, these are values such as:

- Quantity
- Price
- Height

pandas deals primarily with quantitative data, providing you with extensive tools for representing observations of variables. Pandas does not provide for qualitative analysis, but does let you represent qualitative information.

Single and multivariate analysis

Statistics, from a certain perspective, is the practice of studying variables, and specifically the observation of those variables. Much of statistics is based upon doing this analysis for a single variable, which is referred to as **univariate** analysis. Univariate analysis is the simplest form of analyzing data. It does not deal with causes or relationships and is normally used to describe or summarize data, and to find patterns in it.

Multivariate analysis is a modeling technique where there exist two or more output variables that affect the outcome of an experiment. Multivariate analysis is often related to concepts such as correlation and regression, which help us understand the relationships between multiple variables, as well as how those relationships affect the outcome.

pandas primarily provides fundamental univariate analysis capabilities. And these capabilities are generally descriptive statistics, although there is inherent support for concepts such as correlations (as they are very common in finance and other domains).

Other more complex statistics can be performed with StatsModels. Again, this is not per se a weakness of pandas, but a specific design decision to let those concepts be handled by other dedicated Python libraries.

Descriptive statistics

Descriptive statistics are functions that summarize a given dataset, typically where the dataset represents a population or sample of a single variable (univariate data). They describe the dataset and form measures of a central tendency and measures of variability and dispersion.

For example, the following are descriptive statistics:

- The distribution (for example, normal, Poisson)
- The central tendency (for example, mean, median, and mode)
- The dispersion (for example, variance, standard deviation)

As we will see, the pandas `Series` and `DataFrame` objects have integrated support for a large number of descriptive statistics.

Inferential statistics

Inferential statistics differs from descriptive statistics in that inferential statistics attempts to infer conclusions from data instead of simply summarizing it. Examples of inferential statistics include:

- t-test
- chi square
- ANOVA
- Bootstrapping

These inferential techniques are generally deferred from pandas to other tools such as SciPy and StatsModels.

Stochastic models

Stochastic models are a form of statistical modeling that includes one or more random variables, and typically includes use of time series data. The purpose of a stochastic model is to estimate the chance that an outcome is within a specific forecast to predict conditions for different situations.

An example of stochastic modeling is the Monte Carlo simulation. The Monte Carlo simulation is often used for financial portfolio evaluation by simulating the performance of a portfolio based upon repeated simulation of the portfolio in markets that are influenced by various factors and the inherent probability distributions of the constituent stock returns.

pandas gives us the fundamental data structure for stochastic models in the `DataFrame`, often using time series data, to get up and running for stochastic models. While it is possible to code your own stochastic models and analyses using pandas and Python, in many cases there are domain-specific libraries such as PyMC to facilitate this type of modeling.

Probability and Bayesian statistics

Bayesian statistics is an approach to statistical inference, derived from Bayes' theorem, a mathematical equation built off simple probability axioms. It allows an analyst to calculate any conditional probability of interest. A conditional probability is simply the probability of event A given that event B has occurred.

Therefore, in probability terms, the data events have already occurred and have been collected (since we know the probability). By using Bayes' theorem, we can then calculate the probability of various things of interest, given or conditional upon, this already observed data.

Bayesian modeling is beyond the scope of this book, but again the underlying data models are well handled using pandas and then actually analyzed using libraries such as PyMC.

Correlation

Correlation is one of the most common statistics and is directly built into the pandas `DataFrame`. A correlation is a single number that describes the degree of relationship between two variables, and specifically between two sequences of observations of those variables.

A common example of using a correlation is to determine how closely the prices of two stocks follows each other as time progresses. If the changes move closely, the two stocks have a high correlation, and if there is no discernible pattern they are uncorrelated. This is valuable information that can be used in a number of investment strategies.

The level of correlation of two stocks can also vary slightly with the time frame of the entire dataset, as well as the interval. Fortunately, pandas has powerful capabilities for us to easily change these parameters and rerun correlations. We will look at correlations in several places later in the book.

Regression

Regression is a statistical measure that estimates the strength of relationship between a dependent variable and a series of other variables. It can be used to understand the relationships between variables. An example in finance would be understanding the relationship between commodity prices and the stocks of businesses dealing in those commodities.



There was originally a regression model built directly into pandas, but it has been moved out into the StatsModels library. This shows a pattern common in pandas. Often pandas has concepts built into it, but as they mature they are deemed to fit most effectively into other Python libraries. This is both good and bad. It is initially great to have it directly in pandas, but as you upgrade to new versions of pandas it can break your code!

Other Python libraries of value with pandas

pandas forms one small, but important, part of the data analysis and data science ecosystem within Python. As a reference, here are a few other important Python libraries worth noting. The list is not exhaustive, but outlines several you will likely come across..

Numeric and scientific computing - NumPy and SciPy

NumPy (<http://www.numpy.org>) is the cornerstone toolbox for scientific computing with Python, and is included in most distributions of modern Python. It is actually a foundational toolbox from which pandas was built, and when using pandas you will almost certainly use it frequently. NumPy provides, among other things, support for multidimensional arrays with basic operations on them and useful linear algebra functions.

The use of the array features of NumPy goes hand in hand with pandas, specifically the pandas `series` object. Most of our examples will reference NumPy, but the pandas `series` functionality is such a tight superset of the NumPy array that we will, except for a few brief situations, not delve into details of NumPy.

SciPy (<https://www.scipy.org/>) provides a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics, and much more.

Statistical analysis – StatsModels

StatsModels (<http://statsmodels.sourceforge.net/>) is a Python module that allows users to explore data, estimate statistical models, and perform statistical tests. An extensive list of descriptive statistics, statistical tests, plotting functions, and result statistics are available for different types of data and each estimator. Researchers across fields may find that Stats Models fully meets their needs for statistical computing and data analysis in Python.

Features include:

- Linear regression models
- Generalized linear models
- Discrete choice models
- Robust linear models
- Many models and functions for time series analysis
- Nonparametric estimators
- A collection of datasets as examples
- A wide range of statistical tests
- Input-output tools for producing tables in a number of formats (text, LaTex, HTML) and for reading Stata files into NumPy and pandas
- Plotting functions
- Extensive unit tests to ensure correctness of results

Machine learning – scikit-learn

scikit-learn (<http://scikit-learn.org/>) is a machine learning library built from NumPy, SciPy, and matplotlib. It offers simple and efficient tools for common tasks in data analysis such as classification, regression, clustering, dimensionality reduction, model selection, and preprocessing.

PyMC - stochastic Bayesian modeling

PyMC (<https://github.com/pymc-devs/pymc>) is a Python module that implements Bayesian statistical models and fitting algorithms, including Markov chain Monte Carlo. Its flexibility and extensibility make it applicable to a large number of problems. Along with core sampling functionality, PyMC includes methods for summarizing output, plotting, goodness of fit, and convergence diagnostics.

Data visualization - matplotlib and seaborn

Python has a rich set of frameworks for data visualization. Two of the most popular are **matplotlib** and the newer **seaborn**.

Matplotlib

Matplotlib is a Python 2D plotting library that produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the Jupyter Notebook, web application servers, and four graphical user interface toolkits.

pandas contains very tight integration with matplotlib, including functions as part of `Series` and `DataFrame` objects that automatically call matplotlib. This does not mean that pandas is limited to just matplotlib. As we will see, this can be easily changed to others such as ggplot2 and seaborn.

Seaborn

Seaborn (<http://seaborn.pydata.org/introduction.html>) is a library for making attractive and informative statistical graphics in Python. It is built on top of matplotlib and tightly integrated with the PyData stack, including support for NumPy and pandas data structures and statistical routines from SciPy and StatsModels. It provides additional functionality beyond matplotlib, and also by default demonstrates a richer and more modern visual style than matplotlib.

Summary

In this chapter, we went on a tour of the how and why of pandas, data manipulation/analysis, and science. This started with an overview of why pandas exists, what functionality it contains, and how it relates to concepts of data manipulation, analysis, and data science.

Then we covered a process for data analysis to set a framework for why certain functions exist in pandas. These include retrieving data, organizing and cleaning it up, doing exploration, and then building a formal model, presenting your findings, and being able to share and reproduce the analysis.

Next, we covered several concepts involved in data and statistical modeling. This included covering many common analysis techniques and concepts, so as to introduce you to these and make you more familiar when they are explored in more detail in subsequent chapters.

pandas is also a part of a larger Python ecosystem of libraries that are useful for data analysis and science. While this book will focus only on pandas, there are other libraries that you will come across and that were introduced so you are familiar with them when they crop up.

We are ready to begin using pandas. In the next chapter, we will begin to ease ourselves into pandas, starting with obtaining a Python and pandas environment, an overview of Jupyter notebooks, and then getting a quick introduction to pandas `Series` and `DataFrame` objects before delving into them in more depth in subsequent elements of pandas.

Up and Running with pandas

In this chapter, we will cover how to install pandas and start using its basic functionality. The content of the book is provided as IPython and Jupyter notebooks, and hence we will also take a quick look at using both of those tools.

This book will utilize the Anaconda scientific Python distribution from Continuum. Anaconda is a popular Python distribution with both free and paid components. Anaconda provides cross-platform support, including Windows, Mac, and Linux. The base distribution of Anaconda installs pandas, IPython and Jupyter Notebook, thereby making it almost trivial to get started.

In this chapter will cover the following topics:

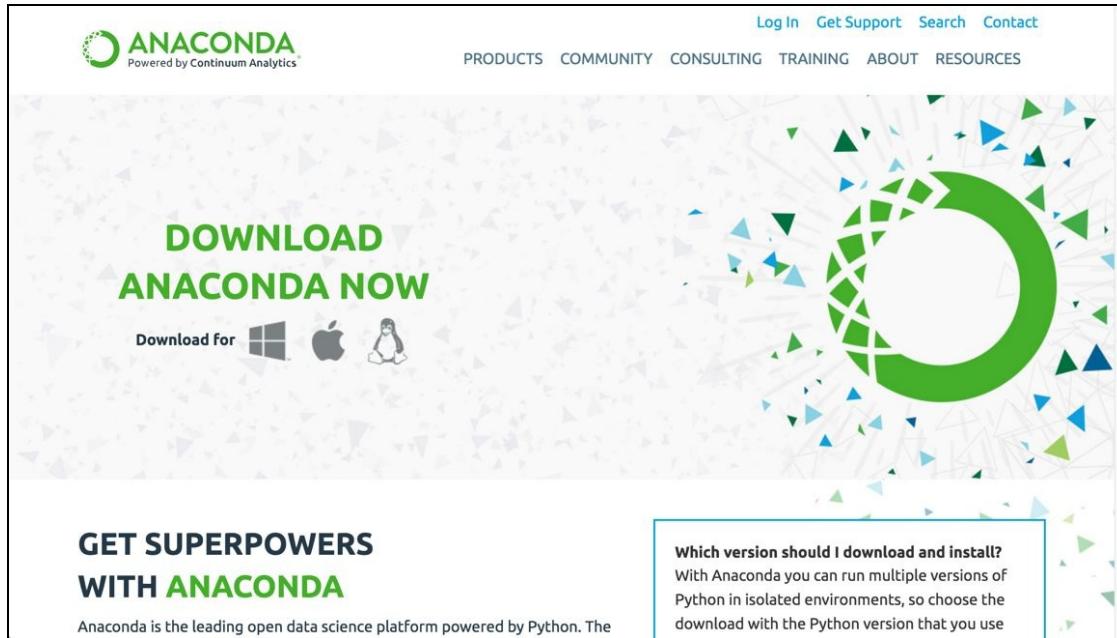
- Installation of Anaconda, pandas, and IPython/Jupyter Notebook
- Using IPython and Jupyter Notebook
- Jupyter and its notebooks
- Setting up your pandas environments
- A quick introduction to the pandas `Series` and `DataFrame`
- Loading data from a CSV file
- Generating a visualization of pandas data

Installation of Anaconda

This book will utilize Anaconda Python version 3, specifically 3.6.1. At the time of writing, pandas is at version 0.20.2. The Anaconda installer by default will install Python, IPython, Jupyter Notebook, and pandas.

Anaconda Python can be downloaded from the Continuum Analytics website at <http://continuum.io/downloads>. The web server will identify your browser's operating system and present you with an appropriate software download file for that platform.

When opening this URL in your browser you will see a page similar to the following:



Click on the link for the installer for your platform. This will present you with a download page similar to

A screenshot of the Anaconda 4.3.1 download page for Windows. It shows three download options: 'Download for Windows', 'Download for macOS', and 'Download for Linux'. The 'Download for Windows' section is highlighted. It includes a brief description of the BSD license, a 'Changelog' link, and instructions for download. It also mentions a zipped Windows installer for firewalls. The 'Python 3.6 version' section is shown in a green box with a '64-BIT INSTALLER (422M)' link. The 'Python 2.7 version' section is shown in a blue box with a '64-BIT INSTALLER (414M)' link. At the bottom right is a 'GET ANACONDA SUPPORT' button.

the following:

Download the 3.x installer. The current version of Anaconda and which will be used in this book is 4.3.1,

```
Last login: Sat Apr  8 21:49:40 on ttys002
[Michaels-iMac-2:~ michaelheydt$ python
Python 3.6.1 |Anaconda custom (x86_64)| (default, Mar 22 2017, 19:25:17)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

with Python 3.6.1:



This changes frequently and by the time you read this it has probably changed.

Execute the installer for your platform, and when it's complete, open a command line or Terminal and execute the `python` command. You should see something similar to the following (this is on a Mac):

```
Last login: Sat Apr  8 21:49:40 on ttys002
[Michaels-iMac-2:~ michaelheydt$ python
Python 3.6.1 |Anaconda custom (x86_64)| (default, Mar 22 2017, 19:25:17)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

You can exit from the Python interpreter by issuing the `exit()` statement:

```
Last login: Sat Apr  8 21:49:40 on ttys002
[Michaels-iMac-2:~ michaelheydt$ python
Python 3.6.1 |Anaconda custom (x86_64)| (default, Mar 22 2017, 19:25:17)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> exit()
Michaels-iMac-2:~ michaelheydt$ 
```

From the Terminal or command line you can verify the installed version of pandas with the `pip show pandas`

```
Python 3.6.1 |Anaconda custom (x86_64)| (default, Mar 22 2017, 19:25:17)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> exit()
Michaels-iMac-2:~ michaelheydt$ pip show pandas
Name: pandas
Version: 0.19.2
Summary: Powerful data structures for data analysis, time series, and statistics
Home-page: http://pandas.pydata.org
Author: The PyData Development Team
Author-email: pydata@googlegroups.com
License: BSD
Location: /Users/michaelheydt/anaconda/lib/python3.6/site-packages
Requires: python-dateutil, pytz, numpy
Michaels-iMac-2:~ michaelheydt$ 
```

command:

The current version that is installed is verified as 0.20.2. Please ensure you are using 0.20.2 or higher, as

there are changes to pandas specific to this version that will be utilized.

Now that we have everything we need installed, let's move on to looking at using IPython and Jupyter Notebook.

IPython and Jupyter Notebook

So far we have executed Python from the command line or Terminal. This is the default **Read-Eval-Print-Loop (REPL)** that comes with Python. This can be used to run all the examples in this book, but the book will use IPython for statements in the text and the code package Jupyter Notebook. Let's take a brief look at both.

IPython

IPython is an alternate shell for interactively working with Python. It provides several enhancements to the default REPL provided by Python.



If you want to learn about IPython in more detail, check out the documentation at <https://ipython.org/ipython-doc/3/interactive/tutorial.html>

To start IPython, simply execute the `ipython` command from the command line/Terminal. When started you

```
Author-email: pydata@googlegroups.com
License: BSD
Location: /Users/michaelheydt/anaconda/lib/python3.6/site-packages
Requires: python-dateutil, pytz, numpy
[Michael's-iMac-2:~ michaelheydt$ ipython
Python 3.6.1 |Anaconda custom (x86_64)| (default, Mar 22 2017, 19:25:17)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: 
```

will see something like the following:



The input prompt shows `In [1]:`. Each time you enter a statement in the IPython REPL, the number in the prompt will increase.

Likewise, output from any particular entry you make will be prefaced with `out [x]:`, where x matches the number of the corresponding `In [x]:`. The following screenshot demonstrates this:

```
Requires: python-dateutil, pytz, numpy
[Michael's-iMac-2:~ michaelheydt$ ipython
Python 3.6.1 |Anaconda custom (x86_64)| (default, Mar 22 2017, 19:25:17)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: 1+1
Out[1]: 2

In [2]: 
```

This numbering of in and out statements will be important to the examples as all examples will be prefaced with `In [x]:` and `out [x]:` so that you can follow along.

Note that these numbers are purely sequential. If you are following through the code in the text and errors occur in the input or you enter additional statements, the numbering may get out of sequence (they can be reset by exiting and restarting IPython). Please use them purely as reference.

Jupyter Notebook

Jupyter Notebook is the evolution of IPython Notebook. It is an open source web application that allows you to create and share documents that contain live code, equations, visualizations, and markdown.

The original IPython Notebook was constrained to Python as the only language. Jupyter Notebook has evolved to allow many programming languages to be used including Python, R, Julia, Scala, and F#.

If you want to take a deeper look at Jupyter Notebook, head to <http://jupyter.org/>, where you will be presented with a page similar to the following:

The screenshot shows the Jupyter website with the following components:

- Header:** jupyter logo, navigation links: Install, About, Resources, Documentation, NBViewer, Widgets, Blog, Donate.
- Left Sidebar:** A screenshot of the Jupyter Notebook interface showing a "Welcome to the Jupyter Notebook Server" message and a "Exploring the Lorenz System" notebook with a 3D plot of the Lorenz attractor.
- Right Content Area:**
 - The Jupyter Notebook:** A section describing it as an open-source web application for creating and sharing documents with live code, equations, and visualizations.
 - Icons and Features:** Four sections with icons and descriptions:
 - Language of choice:** Jupyter Notebook supports over 40 programming languages.
 - Share notebooks:** Notebooks can be shared via email.
 - Interactive widgets:** Code can produce rich output like images.
 - Big data integration:** Leverages big data tools like Apache Spark.

Jupyter Notebook can be downloaded and used independently of Python. Anaconda installs it by default. To start a Jupyter Notebook, issue the following command at the command line or Terminal: **\$ jupyter notebook**

To demonstrate, let's look at how to run the example code that comes with the text. Download the code from the Packt website and unzip the file to a directory of your choosing. In the directory, you will see the following contents similar to the following:

```
[~/Users/michaelheydt/anaconda] bash-3.2$ ls -l
total 7320
-rw-r--r--@ 1 michaelheydt staff 101738 Jun 6 15:34 00_Preface.ipynb
-rw-r--r--@ 1 michaelheydt staff 47667 Apr 8 16:24 01_pandas_and_Data_Analysis.ipynb
-rw-r--r--@ 1 michaelheydt staff 80840 Jun 7 20:33 02_Up_and_Running_with_pandas.ipynb
-rw-r--r--@ 1 michaelheydt staff 92144 Jun 7 20:29 03_Variable_Representation_using_a_Series.ipynb
-rw-r--r--@ 1 michaelheydt staff 47345 May 28 18:49 04_Table_And_MultiVariate_Data_with_The_DataFrame.ipynb
-rw-r--r--@ 1 michaelheydt staff 60752 Apr 30 19:02 05_Manipulating_DataFrame_Structure_and_Contents.ipynb
-rw-r--r--@ 1 michaelheydt staff 40392 May 21 16:46 06_Categorical_Data.ipynb
-rw-r--r--@ 1 michaelheydt staff 47945 May 21 17:55 07_Working_with_Indexes.ipynb
-rw-r--r--@ 1 michaelheydt staff 99970 May 7 21:28 08_Numeric_and_Statistical_Methods.ipynb
-rw-r--r--@ 1 michaelheydt staff 86945 Jun 4 17:44 09_Accessing_Data.ipynb
-rw-r--r--@ 1 michaelheydt staff 69835 May 29 18:04 10_Tidying_Your_Data.ipynb
-rw-r--r--@ 1 michaelheydt staff 64074 May 29 21:25 11_Reorganizing_and_Reshaping_Data.ipynb
-rw-r--r--@ 1 michaelheydt staff 139066 Jun 4 00:17 12_Grouping_and_Aggregating.ipynb
-rw-r--r--@ 1 michaelheydt staff 219724 Jun 4 16:29 13_Time_Series_Data.ipynb
-rw-r--r--@ 1 michaelheydt staff 1440068 Jun 4 21:54 14_Visualization.ipynb
-rw-r--r--@ 1 michaelheydt staff 1084762 Jun 4 23:28 15_Finance.ipynb
drwxr-xr-x@ 25 michaelheydt staff 850 Jun 3 21:17 data
drwxr-xr-x@ 45 michaelheydt staff 1530 May 28 22:58 images
(/~/Users/michaelheydt/anaconda) bash-3.2$
```

Now issue the `jupyter notebook` command. You should see something similar to the following:

```
[Michaels-iMac-2:v2 michaelheydt$ jupyter notebook
[I 18:15:28.371 NotebookApp] Serving notebooks from local directory: /Users/michaelheydt/Dropbox/Packt/LearningPandas/v2
[I 18:15:28.371 NotebookApp] 0 active kernels
[I 18:15:28.371 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/?token=0df85740b89bee4c7504aaedfcacb5b3a8b826e1cf11978c
[I 18:15:28.371 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice
to skip confirmation).
[C 18:15:28.372 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
http://localhost:8888/?token=0df85740b89bee4c7504aaedfcacb5b3a8b826e1cf11978c
[I 18:15:29.100 NotebookApp] Accepting one-time-token-authenticated connection from ::1
```

A browser page will open displaying the Jupyter Notebook homepage, which is `http://localhost:8888/tree`. This will open a web browser window showing this page, which will be a directory listing similar to the

Files

Running

Clusters

Select items to perform actions on them.

Upload

New ▾



□	▼
□	data
□	images
□	00_Preface.ipynb
□	01_pandas and Data Analysis.ipynb
□	02_Up and Running with pandas.ipynb
□	03_Variable Representation using a Series.ipynb
□	04_Table_And_MultiVariate_Data_with_The_DataFrame.ipynb
□	05_Manipulating DataFrame Structure and Contents.ipynb
□	06_Categorical Data.ipynb
□	07_Working with Indexes.ipynb
□	08_Numeric and Statistical Methods.ipynb
□	09_Accessing_Data.ipynb
□	10_Tidying_Your_Data.ipynb
□	11_Reorganizing_and_Reshaping_Data.ipynb
□	12_Grouping_and_Aggregating.ipynb
□	13_Time_Series_Data.ipynb
□	14_Visualization.ipynb
□	15_Finance.ipynb

following:

Clicking on a .ipynb link opens a notebook page. If you open the notebook for this chapter you will see content similar to the following:

The screenshot shows a Jupyter notebook interface. The title bar says "02_Up and Running with pandas". The main area has a toolbar with various icons. Below the toolbar, there's a code cell labeled "In [1]:" containing the following Python code:

```
# import numpy and pandas
import numpy as np
import pandas as pd
from pandas import Series, DataFrame

# used for dates
import datetime

# Set some pandas options that effect the output in the notebooks / ipython
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 15)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 100)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```

The notebook that is displayed is HTML that was generated by Jupyter and IPython. It consists of a number of cells that can be one of four types: code, markdown, raw nbconvert, or heading. All of the examples in this book use either code or markdown cells.

Jupyter runs an IPython kernel for each notebook. Cells that contain Python code are executed within that kernel and the results added to the notebook as HTML.

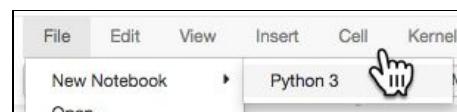
Double-clicking on any of the cells will make the cell editable. When you're done editing the contents of a cell, press *Shift+Enter*, at which point Jupyter/IPython will evaluate the contents and display the results.



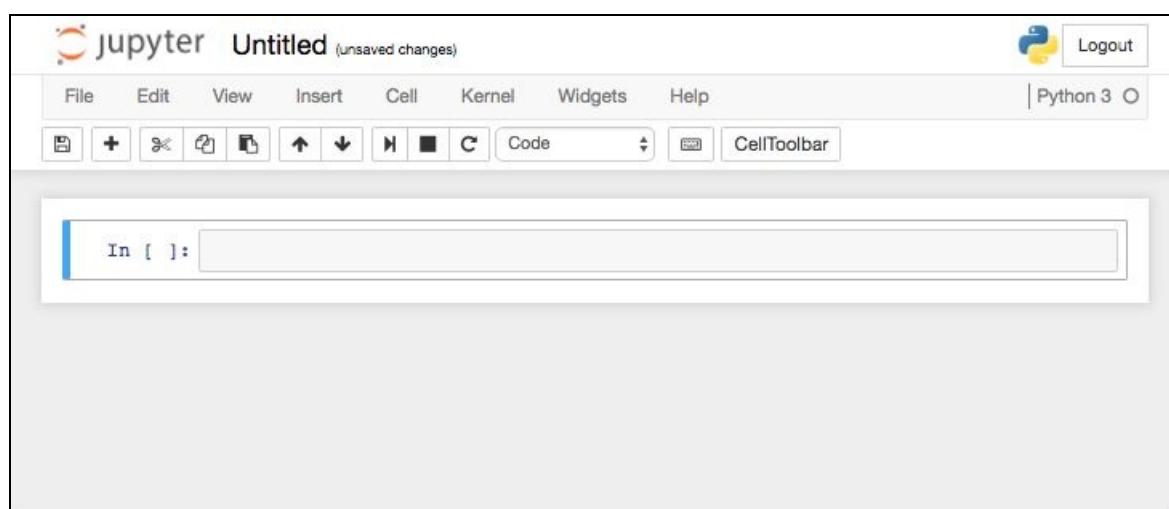
If you want to learn more about the notebook format that underlies the pages, see <https://ipython.org/ipython-doc/3/notebook/nbformat.html>.

The toolbar at the top of a notebook gives you a number of abilities to manipulate the notebook. These include adding, removing, and moving cells up and down in the notebook. Also available are commands to run cells, rerun cells, and restart the underlying IPython kernel.

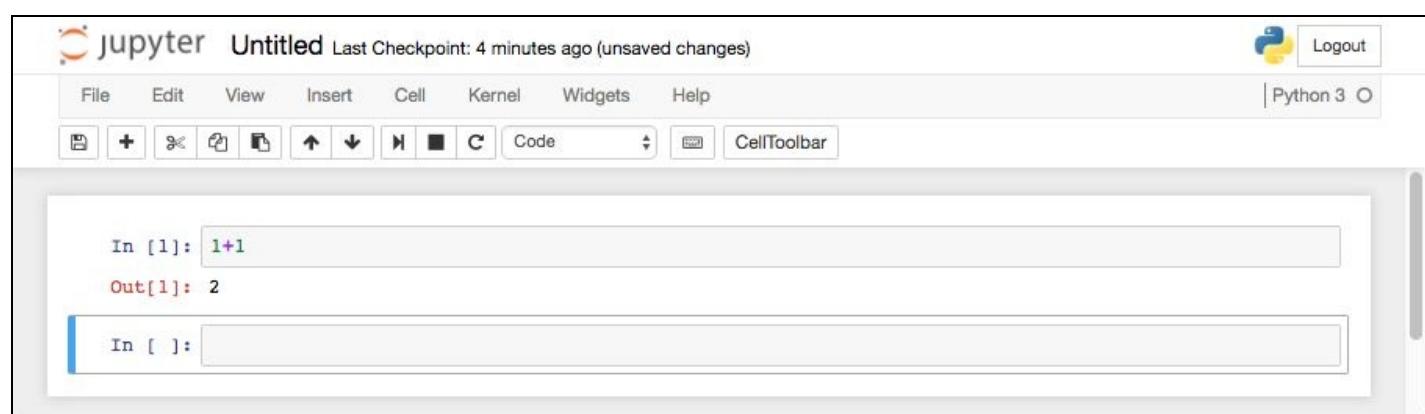
To create a new notebook, go to File | New Notebook | Python 3:



A new notebook page will be created in a new browser tab. Its name will be Untitled:



The notebook consists of a single code cell that is ready to have Python entered. Enter `1+1` in the cell and press *Shift + Enter* to execute.



The cell has been executed and the result shown as `out [1]:`. Jupyter also opened a new cell for you to enter more code or markdown.



Jupyter Notebook automatically saves your changes every minute, but it's still a good thing to save manually every once and a while.

Introducing the pandas Series and DataFrame

Let's jump into using some pandas with a brief introduction to pandas two main data structures, the `Series` and the `DataFrame`. We will examine the following:

- Importing pandas into your application
- Creating and manipulating a pandas `Series`
- Creating and manipulating a pandas `DataFrame`
- Loading data from a file into a `DataFrame`

Importing pandas

Every notebook we will use starts by importing pandas and several other useful Python libraries first. It will also set up several options to control how pandas renders output in a Jupyter Notebook. This code consists of the following:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set some pandas options controlling output format
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```

The first statement imports NumPy and refers to items in the library as `np`.. We won't go into much detail on NumPy in this book, but it is occasionally needed.

The second import makes pandas available to the notebook. We will refer to items in the library with the `pd.` prefix. The `from pandas import Series, DataFrame` statement explicitly imports the `Series` and `DataFrame` objects into the global namespace. This allows us to refer to `Series` and `DataFrame` without the `pd.` prefix. This is convenient as we will use them so frequently that this saves quite a bit of typing.

The `import datetime` statement brings in the `datetime` library, which is commonly used in pandas for time series data. It will be included in the imports for every notebook.

The `pd.set_option()` function calls set up options that inform the notebook how to display output from pandas. The first tells states to render `Series` and `DataFrame` output as text and not HTML. The next two lines specify the maximum number of columns and rows to be output. The final option sets the maximum number of characters of output in each rows.



You can examine more options at the following URL: <http://pandas.pydata.org/pandas-docs/stable/options.html>.

A sharp eye might notice that this cell has no `out [x]`.. Not all cells (or IPython statements) will generate output.

If you desire to use IPython instead of Jupyter Notebook to follow along, you can also execute this code in an IPython shell. For example, you can simply cut and paste the code from the notebook cell. Doing so might look like the following:

```
Michaels-iMac-2:~ michaelheydt$ ipython
Python 3.6.1 |Anaconda custom (x86_64)| (default, Mar 22 2017, 19:25:17)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?           --> Introduction and overview of IPython's features.
%quickref --> Quick reference.
help       --> Python's own help system.
object?    --> Details about 'object', use 'object??' for extra details.

In [1]: # import numpy and pandas
...:     import numpy as np
...:     import pandas as pd
...:     from pandas import Series, DataFrame
...:
...:     # used for dates
...:     import datetime
...:
...:     # Set some pandas options for rendering in notebooks
...:     pd.set_option('display.notebook_repr_html', False)
...:     pd.set_option('display.max_columns', 6)
...:     pd.set_option('display.max_rows', 10)
...:     pd.set_option('display.width', 50)
...:

In [2]: █
```

The IPython shell is smart enough to know you are inserting multiple lines and will indent appropriately. And notice that there is also no `out [x]:` in the IPython shell. `pd.set_option` does not return any content and hence there is no annotation.

The pandas Series

The pandas `Series` is the base data structure of pandas. A series is similar to a NumPy array, but it differs by having an index, which allows for much richer lookup of items instead of just a zero-based array index value.

The following creates a series from a Python list.:

```
In [2]: # create a four item Series
s = pd.Series([1, 2, 3, 4])
s

Out[2]: 0    1
         1    2
         2    3
         3    4
        dtype: int64
```

The output consists of two columns of information. The first is the index and the second is the data in the `series`. Each row of the output represents the index **label** (in the first column) and then the value associated with that label.

Because this `series` was created without specifying an index (something we will do next), pandas automatically creates an integer index with labels starting at 0 and increasing by one for each data item.

The values of a `series` object can then be accessed by using the `[]` operator, passing the label for the value you require. The following gets the value for the label `1`:

```
In [3]: # get value at label 1
s[1]

Out[3]: 2
```

This looks very much like normal array access in many programming languages. But as we will see, the index does not have to start at 0, nor increment by one, and can be many other data types than just an integer. This ability to associate flexible indexes in this manner is one of the great superpowers of pandas.

Multiple items can be retrieved by specifying their labels in a Python list. The following retrieves the values at labels `1` and `3`:

```
In [4]: # return a Series with the row with labels 1 and 3
s[[1, 3]]

Out[4]: 1    2
         3    4
        dtype: int64
```

A `Series` object can be created with a user-defined index by using the `index` parameter and specifying the index labels. The following creates a `Series` with the same values but with an index consisting of string values:

```
In [5]: # create a series using an explicit index
s = pd.Series([1, 2, 3, 4],
              index = ['a', 'b', 'c', 'd'])
s
```

```
Out[5]: a    1
         b    2
         c    3
         d    4
        dtype: int64
```

Data in the `Series` object can now be accessed by those alphanumeric index labels. The following retrieves the values at index labels '`a`' and '`d`':

```
In [6]: # look up items the series having index 'a' and 'd'
s[['a', 'd']]
```

```
Out[6]: a    1
         d    4
        dtype: int64
```

It is still possible to refer to the elements of this `Series` object by their numerical 0-based position. :

```
In [7]: # passing a list of integers to a Series that has
# non-integer index labels will look up based upon
# 0-based index like an array
s[[1, 2]]
```

```
Out[7]: b    2
         c    3
        dtype: int64
```

We can examine the index of a `Series` using the `.index` property:

```
In [8]: # get only the index of the Series
s.index
```

```
Out[8]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

The index is itself actually a pandas object, and this output shows us the values of the index and the data type used for the index. In this case, note that the type of the data in the index (referred to as the `dtype`) is `object` and not `string`. We will examine how to change this later in the book.

A common usage of a `Series` in pandas is to represent a time series that associates date/time index labels with values. The following demonstrates this by creating a date range using the `pd.date_range()` pandas function:

```
In [9]: # create a Series who's index is a series of dates
# between the two specified dates (inclusive)
dates = pd.date_range('2016-04-01', '2016-04-06')
dates
```

```
Out[9]: DatetimeIndex(['2016-04-01', '2016-04-02', '2016-04-03',
                       '2016-04-04', '2016-04-05', '2016-04-06'],
                      dtype='datetime64[ns]', freq='D')
```

This has created a special index in pandas called `DatetimeIndex`, which is a specialized type of pandas index that is optimized to index data with dates and times.

Now let's create a `Series` using this index. The data values represent high temperatures on specific days:

```
In [10]: # create a Series with values (representing temperatures)
# for each date in the index
temp1 = pd.Series([80, 82, 85, 90, 83, 87],
                  index = dates)
temp1

Out[10]: 2016-04-01    80
          2016-04-02    82
          2016-04-03    85
          2016-04-04    90
          2016-04-05    83
          2016-04-06    87
Freq: D, dtype: int64
```

This type of series with a `DateTimeIndex` is referred to as a time series.

We can look up a temperature on a specific date by using the date as a string:

```
In [11]: # what's the temperature for 2016-4-4?
temp1['2016-04-04']

Out[11]: 90
```

Two `Series` objects can be applied to each other with an arithmetic operation. The following code creates a second `Series` and calculates the difference in temperature between the two:

```
In [12]: # create a second series of values using the same index
temp2 = pd.Series([70, 75, 69, 83, 79, 77],
                  index = dates)
# the following aligns the two by their index values
# and calculates the difference at those matching labels
temp_diffs = temp1 - temp2
temp_diffs

Out[12]: 2016-04-01    10
          2016-04-02     7
          2016-04-03    16
          2016-04-04     7
          2016-04-05     4
          2016-04-06    10
Freq: D, dtype: int64
```

 *The result of an arithmetic operation (+, -, /, *, ...) on two `Series` objects that are non-scalar values returns another `Series` object.*

Since the index is not integer, we can also look up values by 0-based value:

```
In [13]: # and also possible by integer position as if the
# series was an array
temp_diffs[2]

Out[13]: 16
```

Finally, pandas provides many descriptive statistical methods. As an example, the following returns the mean of the temperature differences:

```
In [14]: # calculate the mean of the values in the Series
temp_diffs.mean()

Out[14]: 9.0
```


The pandas DataFrame

A pandas `Series` can only have a single value associated with each index label. To have multiple values per index label we can use a data frame. A data frame represents one or more `Series` objects aligned by index label. Each series will be a column in the data frame, and each column can have an associated name.



In a way, a data frame is analogous to a relational database table in that it contains one or more columns of data of heterogeneous types (but a single type for all items in each respective column).

The following creates a `DataFrame` object with two columns and uses the temperature `Series` objects:

```
In [15]: # create a DataFrame from the two series objects temps1 and temps2
# and give them column names
temps_df = pd.DataFrame(
    {'Missoula': temps1,
     'Philadelphia': temps2})
temps_df
```

```
Out[15]:
```

	Missoula	Philadelphia
2016-04-01	80	70
2016-04-02	82	75
2016-04-03	85	69
2016-04-04	90	83
2016-04-05	83	79
2016-04-06	87	77

The resulting data frame has two columns named `Missoula` and `Philadelphia`. These columns are new `Series` objects contained within the data frame with the values copied from the original `Series` objects.

Columns in a `DataFrame` object can be accessed using an array indexer `[]` with the name of the column or a list of column names. The following code retrieves the `Missoula` column:

```
In [16]: # get the column with the name Missoula
temps_df['Missoula']
```

```
Out[16]:
```

2016-04-01	80
2016-04-02	82
2016-04-03	85
2016-04-04	90
2016-04-05	83
2016-04-06	87

Freq: D, Name: Missoula, dtype: int64

And the following code retrieves the `Philadelphia` column:

```
In [17]: # likewise we can get just the Philadelphia column
temps_df['Philadelphia']
```

```
Out[17]:
```

2016-04-01	70
2016-04-02	75
2016-04-03	69
2016-04-04	83
2016-04-05	79
2016-04-06	77

Freq: D, Name: Philadelphia, dtype: int64

A Python list of column names can also be used to return multiple columns:

```
In [18]: # return both columns in a different order
temp_df[['Philadelphia', 'Missoula']]
```

	Philadelphia	Missoula
2016-04-01	70	80
2016-04-02	75	82
2016-04-03	69	85
2016-04-04	83	90
2016-04-05	79	83
2016-04-06	77	87



There is a subtle difference in a `DataFrame` object as compared to a `Series` object. Passing a list to the `[]` operator of `DataFrame` retrieves the specified columns whereas a `Series` would return rows.

If the name of a column does not have spaces, it can be accessed using property-style:

```
In [19]: # retrieve the Missoula column through property syntax
temp_df.Missoula
```

	Missoula
2016-04-01	80
2016-04-02	82
2016-04-03	85
2016-04-04	90
2016-04-05	83
2016-04-06	87

Freq: D, Name: Missoula, dtype: int64

Arithmetic operations between columns within a data frame are identical in operation to those on multiple `Series`. To demonstrate, the following code calculates the difference between temperatures using property

```
In [20]: # calculate the temperature difference between the two cities
temp_df.Missoula - temp_df.Philadelphia
```

	Missoula - Philadelphia
2016-04-01	10
2016-04-02	7
2016-04-03	16
2016-04-04	7
2016-04-05	4
2016-04-06	10

Freq: D, dtype: int64

notation:

A new column can be added to `DataFrame` simply by assigning another `Series` to a column using the array indexer `[]` notation. The following adds a new column in the `DataFrame` with the temperature differences:

```
In [21]: # add a column to temp_df which contains the difference in temps
temp_df['Difference'] = temp_diffs
temp_df
```

	Missoula	Philadelphia	Difference
2016-04-01	80	70	10
2016-04-02	82	75	7
2016-04-03	85	69	16
2016-04-04	90	83	7
2016-04-05	83	79	4
2016-04-06	87	77	10

The names of the columns in a `DataFrame` are accessible via the `.columns` property:

```
In [22]: # get the columns, which is also an Index object
temp_df.columns
```

	Missoula	Philadelphia	Difference
--	----------	--------------	------------

Out[22]: Index(['Missoula', 'Philadelphia', 'Difference'], dtype='object')

The `DataFrame` and `Series` objects can be sliced to retrieve specific rows. The following slices the second through fourth rows of temperature difference values:

```
In [23]: # slice the temp differences column for the rows at  
# location 1 through 4 (as though it is an array)  
temps_df.Difference[1:4]
```

```
Out[23]: 2016-04-02    7  
2016-04-03    16  
2016-04-04    7  
Freq: D, Name: Difference, dtype: int64
```

Entire rows from a data frame can be retrieved using the `.loc` and `.iloc` properties. `.loc` ensures that the lookup is by index label, where `.iloc` uses the 0-based position. -

The following retrieves the second row of the data frame:

```
In [24]: # get the row at array position 1  
temps_df.iloc[1]
```



```
Out[24]: Missoula      82  
Philadelphia    75  
Difference      7  
Name: 2016-04-02 00:00:00, dtype: int64
```

Notice that this result has converted the row into a series with the column names of the data frame pivoted into the index labels of the resulting series. The following shows the resulting index of the result:

```
In [25]: # the names of the columns have become the index  
# they have been 'pivoted'  
temps_df.iloc[1].index
```

```
Out[25]: Index(['Missoula', 'Philadelphia', 'Difference'], dtype='object')
```

Rows can be explicitly accessed via index label using the `.loc` property. The following code retrieves a

```
In [26]: # retrieve row by index label using .loc  
temps_df.loc['2016-04-05']
```



```
Out[26]: Missoula      83  
Philadelphia    79  
Difference      4  
Name: 2016-04-05 00:00:00, dtype: int64
```

row by the index label:

Specific rows in a `DataFrame` object can be selected using a list of integer positions. The following selects the values from the `Difference` column in rows at integer locations 1, 3, and 5:

```
In [27]: # get the values in the Differences column in rows 1, 3 and 5  
# using 0-based location  
temps_df.iloc[[1, 3, 5]].Difference
```

```
Out[27]: 2016-04-02    7  
2016-04-04    7  
2016-04-06   10  
Freq: D, Name: Difference, dtype: int64
```

Rows of a data frame can be selected based upon a logical expression that is applied to the data in each row. The following shows values in the `Missoula` column that are greater than 82 degrees:

```
In [28]: # which values in the Missoula column are > 82?  
temps_df.Missoula > 82
```

```
Out[28]: 2016-04-01    False  
2016-04-02    False  
2016-04-03     True  
2016-04-04     True  
2016-04-05     True  
2016-04-06     True  
Freq: D, Name: Missoula, dtype: bool
```

The results from an expression can then be applied to the `[]` operator of a data frame (and a series) which results in only the rows where the expression evaluated to `True` being returned:

```
In [29]: # return the rows where the temps for Missoula > 82  
temp_df[temps_df.Missoula > 82]
```

```
Out[29]:
```

	Missoula	Philadelphia	Difference
2016-04-03	85	69	16
2016-04-04	90	83	7
2016-04-05	83	79	4
2016-04-06	87	77	10

This technique is referred to as **Boolean Selection** in pandas terminology and will form the basis of selecting rows based upon values in specific columns (like a query in SQL using a `WHERE` clause - but as we will see it is much more powerful).

Loading data from files into a DataFrame

The pandas library provides facilities for easy retrieval of data from a variety of data sources as pandas objects. As a quick example, let's examine the ability of pandas to load data in CSV format.

This example will use a file provided with the code from this book, `data/goog.csv`, and the contents of the file represent time series financial information for the Google stock.

The following statement uses the operating system (from within Jupyter Notebook or IPython) to display the content of this file. Which command you will need to use depends on your operating system:

```
In [30]: # display the contents of test1.csv  
# which command to use depends on your OS  
!head data/goog.csv # on non-windows systems  
#!type data/test1.csv # on windows systems, all lines
```

```
Date,Open,High,Low,Close,Volume  
12/19/2016,790.219971,797.659973,786.27002,794.200012,1225900  
12/20/2016,796.76001,798.650024,793.27002,796.419983,925100  
12/21/2016,795.840027,796.676025,787.099976,794.559998,1208700  
12/22/2016,792.359985,793.320007,788.580017,791.26001,969100  
12/23/2016,790.900024,792.73999,787.280029,789.909973,623400  
12/27/2016,790.679993,797.859985,787.656982,791.549988,789100  
12/28/2016,793.700012,794.22998,783.200012,785.049988,1132700  
12/29/2016,783.330017,785.929993,778.919983,782.789978,742200  
12/30/2016,782.75,782.780029,770.409973,771.820007,1760200
```

This information can be easily imported into a `DataFrame` using the `pd.read_csv()` function:

```
In [31]: # read the contents of the file into a DataFrame  
df = pd.read_csv('data/goog.csv')  
df
```

	Date	Open	High	Low	Close	Volume
0	12/19/2016	790.219971	797.659973	786.270020	794.200012	1225900
1	12/20/2016	796.760010	798.650024	793.270020	796.419983	925100
2	12/21/2016	795.840027	796.676025	787.099976	794.559998	1208700
3	12/22/2016	792.359985	793.320007	788.580017	791.260010	969100
4	12/23/2016	790.900024	792.739990	787.280029	789.909973	623400
..
56	3/13/2017	844.000000	848.684998	843.250000	845.539978	1149500
57	3/14/2017	843.640015	847.239990	840.799988	845.619995	779900
58	3/15/2017	847.590027	848.630005	840.770020	847.200012	1379600
59	3/16/2017	849.030029	850.849976	846.130005	848.780029	970400
60	3/17/2017	851.609985	853.400024	847.109985	852.119995	1712300

[61 rows x 6 columns]

pandas has no idea that the first column in the file is a date and has treated the contents of the date field as a string. This can be verified using the following pandas statement, which shows the type of the `Date`

```
In [32]: # the contents of the date column  
df.Date
```

	Date
0	12/19/2016
1	12/20/2016
2	12/21/2016
3	12/22/2016
4	12/23/2016
..	..
56	3/13/2017
57	3/14/2017
58	3/15/2017
59	3/16/2017
60	3/17/2017

Name: Date, Length: 61, dtype: object

column as a string:

```
In [33]: # we can get the first value in the date column  
df.Date[0]
```

```
Out[33]: '12/19/2016'
```

```
In [34]: # it is a string  
type(df.Date[0])
```

```
Out[34]: str
```

The `parse_dates` parameter of the `pd.read_csv()` function to guide pandas on how to convert data directly into a pandas date object. The following informs pandas to convert the content of the `Date` column into actual

```
In [35]: # read the data and tell pandas the date column should be  
# a date in the resulting DataFrame  
df = pd.read_csv('data/goog.csv', parse_dates=['Date'])  
df
```

	Date	Open	High	Low	Close	Volume
0	2016-12-19	790.219971	797.659973	786.270020	794.200012	1225900
1	2016-12-20	796.760010	798.650024	793.270020	796.419983	925100
2	2016-12-21	795.840027	796.676025	787.099976	794.559998	1208700
3	2016-12-22	792.359985	793.320007	788.580017	791.260010	969100
4	2016-12-23	790.900024	792.739990	787.280029	789.909973	623400
..
56	2017-03-13	844.000000	848.684998	843.250000	845.539978	1149500
57	2017-03-14	843.640015	847.239990	840.799988	845.619995	779900
58	2017-03-15	847.590027	848.630005	840.770020	847.200012	1379600
59	2017-03-16	849.030029	850.849976	846.130005	848.780029	970400
60	2017-03-17	851.609985	853.400024	847.109985	852.119995	1712300

TimeStamp objects:

```
[61 rows x 6 columns]
```

If we check whether it worked, we see that the date is a `Timestamp`:

```
In [36]: # verify the type now is date  
# in pandas, this is actually a Timestamp  
type(df.Date[0])
```



```
Out[36]: pandas._libs.tslib.Timestamp
```

Unfortunately, this has not used the date field as the index for the data frame. Instead, it uses the default

```
In [37]: # unfortunately the index is numeric which makes  
# accessing data by date more complicated  
df.index
```

```
Out[37]: RangeIndex(start=0, stop=61, step=1)
```



Note that this is now a `RangeIndex`, where in previous versions of pandas it would have been an integer index. We'll examine this difference later in the book.

This can be fixed using the `index_col` parameter of the `pd.read_csv()` function to specify which column in the

```
In [38]: # read in again, now specify the data column as being the  
# index of the resulting DataFrame  
df = pd.read_csv('data/goog.csv',  
                 parse_dates=['Date'],  
                 index_col='Date')  
df
```

Date	Open	High	Low	Close	Volume
2016-12-19	790.219971	797.659973	786.270020	794.200012	1225900
2016-12-20	796.760010	798.650024	793.270020	796.419983	925100
2016-12-21	795.840027	796.676025	787.099976	794.559998	1208700
2016-12-22	792.359985	793.320007	788.580017	791.260010	969100
2016-12-23	790.900024	792.739990	787.280029	789.909973	623400
..
2017-03-13	844.000000	848.684998	843.250000	845.539978	1149500
2017-03-14	843.640015	847.239990	840.799988	845.619995	779900
2017-03-15	847.590027	848.630005	840.770020	847.200012	1379600
2017-03-16	849.030029	850.849976	846.130005	848.780029	970400
2017-03-17	851.609985	853.400024	847.109985	852.119995	1712300

file should be used as the index:

```
[61 rows x 5 columns]
```

And the index now is a `DatetimeIndex`, which lets us look up rows using dates.

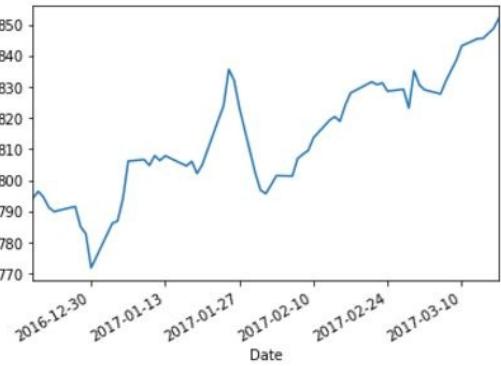
```
In [39]: # and the index is now a DatetimeIndex  
df.index
```

```
Out[39]: DatetimeIndex(['2016-12-19', '2016-12-20', '2016-12-21', '2016-12-22',  
'2016-12-23', '2016-12-27', '2016-12-28', '2016-12-29',  
'2016-12-30', '2017-01-03', '2017-01-04', '2017-01-05',  
'2017-01-06', '2017-01-09', '2017-01-10', '2017-01-11',  
'2017-01-12', '2017-01-13', '2017-01-17', '2017-01-18',  
'2017-01-19', '2017-01-20', '2017-01-23', '2017-01-24',  
'2017-01-25', '2017-01-26', '2017-01-27', '2017-01-30',  
'2017-01-31', '2017-02-01', '2017-02-02', '2017-02-03',  
'2017-02-06', '2017-02-07', '2017-02-08', '2017-02-09',  
'2017-02-10', '2017-02-13', '2017-02-14', '2017-02-15',  
'2017-02-16', '2017-02-17', '2017-02-21', '2017-02-22',  
'2017-02-23', '2017-02-24', '2017-02-27', '2017-02-28',  
'2017-03-01', '2017-03-02', '2017-03-03', '2017-03-06',  
'2017-03-07', '2017-03-08', '2017-03-09', '2017-03-10',  
'2017-03-13', '2017-03-14', '2017-03-15', '2017-03-16',  
'2017-03-17'],  
dtype='datetime64[ns]', name='Date', freq=None)
```


Visualization

We will dive into visualization in quite some depth in Chapter 14, *Visualization*, but prior to then we will occasionally perform a quick visualization of data in pandas. Creating a visualization of data is quite simple with pandas. All that needs to be done is to call the `.plot()` method. The following demonstrates by plotting the Close value of the stock data:

```
In [40]: # plots the values in the Close column  
df.Close.plot();
```



Summary

In this chapter, we installed the Anaconda Scientific version of Python. This also installs pandas and Jupyter Notebook, setting you right up with an environment for performing data manipulation and analysis, along with creating notebooks to visualize, present, and share your analyses.

We also took an introductory look at the pandas `Series` and `DataFrame` objects, demonstrating some of the fundamental capabilities. This exposition showed you how to perform a few basic operations that you can use to get up and running with pandas prior to diving in and learning all the details.

In the next several chapters, we will dive deep into the operations of the `Series` and `DataFrame`, with the next chapter focusing specifically on the `Series`.

Representing Univariate Data with the Series

The `Series` is the primary building block of pandas. It represents a one-dimensional array-like set of values of a single data type. It is often used to model zero or more measurements of a single variable. While it can appear like an array, a `Series` has an associated index that can be used to perform very efficient retrievals of values based upon labels.

A `Series` also performs automatic alignment of data between itself and other pandas objects. Alignment is a core feature of pandas where data is multiple pandas objects that are matched by label value before any operation is performed. This allows the simple application of operations without needing to explicitly code joins.

In this chapter, we will examine how to model measurements of a variable using a `Series`, including using an index to retrieve samples. This examination will include overviews of several patterns involved in index labeling, slicing and querying data, alignment, and re-indexing data.

Specifically, in this chapter we will cover the following topics:

- Creating a series using Python lists, dictionaries, NumPy functions, and scalar values
- Accessing the index and values properties of the `Series`
- Determining the size and shape of a `Series` object
- Specifying an index at the time of `Series` creation
- Using `head`, `tail`, and `take` to access values
- Value lookup by index label and position
- Slicing and common slicing patterns
- Alignment via index labels
- Performing Boolean selection
- Re-indexing a `Series`
- In-place modification of values

Configuring pandas

We start the examples in the chapter using the following imports and configuration statements:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set some pandas options controlling output format
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 80)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```


Creating a Series

A `Series` can be created using several techniques. We will examine the following three:

- Using a Python list or dictionary
- From NumPy arrays
- Using a scalar value

Creating a Series using Python lists and dictionaries

A series can be created from a Python list:

```
In [2]: # create a series of multiple values from a list
s = pd.Series([10, 11, 12, 13, 14])
s

Out[2]: 0    10
         1    11
         2    12
         3    13
         4    14
        dtype: int64
```

The first column of numbers represents the label in the index of the series. The second column contains the values. `dtype: int64` denotes that the data type of the values in the series is `int64`.

By default, pandas will create an index consisting of consecutive integers starting at 0. This makes the series look like an array in many other programming languages. As an example, we can look up the value at label 3:

```
In [3]: # value stored at index label 3
s[3]

Out[3]: 13
```



This lookup was by label value, not 0-based position. We will examine this in detail later in the chapter.

Data types other than integers can be used. The following creates a series of string values:

```
In [4]: # create a Series of alphas
pd.Series(['Mike', 'Marcia', 'Mikael', 'Bleu'])

Out[4]: 0      Mike
         1     Marcia
         2     Mikael
         3      Bleu
        dtype: object
```

To create a series consisting of a sequence of n identical values v , use the Python shorthand for list creation, $[v]^n$. The following creates five values of 2:

```
In [5]: # a sequence of 5 values, all 2
pd.Series([2]*5)

Out[5]: 0    2
         1    2
         2    2
         3    2
         4    2
        dtype: int64
```

A similar type of shorthand is the following, which uses the Python shorthand to use each character as a list item:

```
In [6]: # use each character as a value
pd.Series(list('abcde'))
```

```
Out[6]: 0    a
1    b
2    c
3    d
4    e
dtype: object
```

A `Series` can be directly initialized from a Python dictionary. When using a dictionary, the keys of the dictionary are used as the index labels:

```
In [7]: # create Series from dict
pd.Series({'Mike': 'Dad',
           'Marcia': 'Mom',
           'Mikael': 'Son',
           'Bleu': 'Best doggie ever'})
```

```
Out[7]: Bleu      Best doggie ever
        Marcia      Mom
        Mikael      Son
        Mike       Dad
dtype: object
```


Creation using NumPy functions

It is a common practice to initialize the `Series` object's various NumPy functions. As an example, the following uses the NumPy `np.arange` function to create a sequence of integer values between 4 and 8:

```
In [8]: # 4 through 8
pd.Series(np.arange(4, 9))

Out[8]: 0    4
         1    5
         2    6
         3    7
         4    8
dtype: int64
```

The `np.linspace()` method is similar in functionality but allows us to specify the number of values to be created between (and including) the two specified values, and with a specified number of steps:

```
In [9]: # 0 through 9
pd.Series(np.linspace(0, 9, 5))

Out[9]: 0    0.00
         1    2.25
         2    4.50
         3    6.75
         4    9.00
dtype: float64
```

It is also common to generate a set of random numbers using `np.random.normal()`. The following generates five random numbers from a normal distribution:

```
In [10]: # random numbers
np.random.seed(12345) # always generate the same values
# 5 normally random numbers
pd.Series(np.random.normal(size=5))

Out[10]: 0    -0.204708
         1     0.478943
         2    -0.519439
         3    -0.555730
         4     1.965781
dtype: float64
```


Creation using a scalar value

A `Series` can also be created using a scalar value:

```
In [11]: # create a one item Series
s = pd.Series(2)
s

Out[11]: 0    2
dtype: int64
```

This seems like a degenerate case with the `Series` only having a single value. There are, however, scenarios where this is important, such as when a series is multiplied by a scalar value, like so:

```
In [12]: # create the Series
s = pd.Series(np.arange(0, 5))
# multiple all values by 2
s * 2

Out[12]: 0    0
1    2
2    4
3    6
4    8
dtype: int64
```

Underneath the covers, pandas took the value `2` and created a `Series` from that scalar value with a matching index to that in `s`, and then performed the multiplication through alignment of the two `Series`. We will look at this example again in more detail later in the chapter.

The .index and .values properties

Each series object consists of a series of values and an index. The values can be accessed through the

```
In [13]: # get the values in the Series
s = pd.Series([1, 2, 3])
s.values
```

.values property:

```
Out[13]: array([1, 2, 3])
```

The result from is a NumPy array object, as the following verifies:

```
In [14]: # show that this is a numpy array
type(s.values)
```

```
Out[14]: numpy.ndarray
```

 *This is called out for informational purposes. We will not examine NumPy arrays in this book. Historically, pandas did use NumPy arrays under the covers, so NumPy arrays were more important in the past, but this dependency has been removed in recent versions. But as a convenience, .values returns a NumPy array even if the underlying representation is not a NumPy array.*

In addition, the index for the series can be retrieved using .index:

```
In [15]: # get the index of the Series
s.index
```

```
Out[15]: RangeIndex(start=0, stop=3, step=1)
```

The type of index created by pandas is `RangeIndex`. This is a change in pandas from the previous version of this book, when this type of index did not exist. The `RangeIndex` object represents a range of values from the `start` to the `stop` value with the specified `step`. This is efficient for pandas, compared to the previously utilized `Int64Index`.

 *The `RangeIndex` is simply one type of the indexes that we will explore (much of the detail in is Chapter 6, Indexing Data).*

The size and shape of a Series

The number of items in a `Series` object can be determined by several techniques, the first of which is by

```
In [16]: # example series
s = pd.Series([0, 1, 2, 3])
len(s)
```

```
Out[16]: 4
```

using the Python `len()` function:

The same result can be obtained by using the `.size` property:

```
In [17]: # .size is also the # of items in the Series
s.size
```

```
Out[17]: 4
```

An alternate form for getting the size of a `Series` is to use the `.shape` property. This returns a two-value tuple, but with only the first value specified and representing the size:

```
In [18]: # .shape is a tuple with one value
s.shape
```

```
Out[18]: (4,)
```


Specifying an index at creation

The labels in an index can be specified at the creation of the `Series` by using the `index` parameter of the constructor. The following creates a `Series` and assigns strings to each label of the index:

```
In [19]: # explicitly create an index
labels = ['Mike', 'Marcia', 'Mikael', 'Bleu']
role = ['Dad', 'Mom', 'Son', 'Dog']
s = pd.Series(labels, index=role)
s
```

```
Out[19]: Dad      Mike
          Mom     Marcia
          Son     Mikael
          Dog      Bleu
          dtype: object
```

Examining the `.index` property, we find the following index was created:

```
In [20]: # examine the index
s.index
```

```
Out[20]: Index(['Dad', 'Mom', 'Son', 'Dog'], dtype='object')
```

Using this index, we can ask a question such as who is the Dad?:

Heads, tails, and takes

pandas provides the `.head()` and `.tail()` methods to examine the first (head) or last (tail) few rows in a series. By default, these return the first or last five rows, but this can be changed using the `n` parameter.

Let's examine the usage given the following Series:

```
In [22]: # a ten item Series  
s = pd.Series(np.arange(1, 10),  
              index=list('abcdefghi'))
```

The following retrieves the first five rows:

```
In [23]: # show the first five  
s.head()  
  
Out[23]: a    1  
         b    2  
         c    3  
         d    4  
         e    5  
        dtype: int64
```

The number of items can be changed using the `n` parameter (or just by specifying the number):

```
In [24]: # the first three  
s.head(n = 3) # s.head(3) is equivalent  
  
Out[24]: a    1  
         b    2  
         c    3  
        dtype: int64
```

`.tail()` returns the last five rows:

```
In [25]: # the last five  
s.tail()  
  
Out[25]: e    5  
         f    6  
         g    7  
         h    8  
         i    9  
        dtype: int64
```

It works similarly when specifying a number other than 5:

```
In [26]: # the last 3  
s.tail(n = 3) # equivalent to s.tail(3)  
  
Out[26]: g    7  
         h    8  
         i    9  
        dtype: int64
```

The `.take()` method returns the rows in a series at the specified integer position:

```
In [27]: # only take specific items by position  
s.take([1, 5, 8])  
  
Out[27]: b    2  
         f    6  
         i    9  
        dtype: int64
```


Retrieving values in a Series by label or position

Values in a `Series` can be retrieved in two general ways: by index label or by 0-based position. Pandas provides you with a number of ways to perform either of these lookups. Let's examine a few of the common techniques.

Lookup by label using the [] operator and the .ix[] property

An implicit label lookup is performed using the [] operator. This operator normally looks up values based upon the given index labels.

Let's start by using the following `Series`:

```
In [28]: # we will use this series to examine lookups
s1 = pd.Series(np.arange(10, 15), index=list('abcde'))
s1

Out[28]: a    10
          b    11
          c    12
          d    13
          e    14
         dtype: int64
```

A single value can be looked up using just the index label of the desired item:

```
In [29]: # get the value with label 'a'
s1['a']

Out[29]: 10
```

Multiple items can be retrieved at once using a list of index labels:

```
In [30]: # get multiple items
s1[['d', 'b']]

Out[30]: d    13
          b    11
         dtype: int64
```

We can also look up values using integers that represent positions:

```
In [31]: # gets values based upon position
s1[[3, 1]]

Out[31]: d    13
          b    11
         dtype: int64
```

This works purely because the index is not using integer labels. If integers are passed to [], and the index has integer values, then the lookup is performed by matching the values passed in to the values of the integer labels.

This can be demonstrated using the following `Series`:

```
In [32]: # to demo lookup by matching labels as integer values
s2 = pd.Series([1, 2, 3, 4], index=[10, 11, 12, 13])
s2

Out[32]: 10    1
          11    2
          12    3
          13    4
         dtype: int64
```

The following looks up values at labels 13 and 10, not positions 13 and 10:

```
In [33]: # this is by label not position  
s2[[13, 10]]
```

```
Out[33]: 13    4  
10    1  
dtype: int64
```

Lookup using the `[]` operator is identical to using the `.ix[]` property. However, `.ix[]` has been deprecated since pandas version 0.20.1. The reason for the deprecation is the confusion caused by integers being passed to the operators and the difference in operation depending on the type of the label in the index.

The ramifications of this is that neither `[]` or `.ix[]` should be used for lookup. Instead, use the `.loc[]` and `.iloc[]` properties, which explicitly look up by label or position only.

Explicit lookup by position with .iloc[]

Value lookup by position can be performed using `.iloc[]`. The following demonstrates using integers as

```
In [34]: # explicitly by position  
s1.iloc[[0, 2]]  
  
Out[34]: a    10  
          c    12  
          dtype: int64
```

parameters:

The following looks up by position, even though the index has integer labels:

```
In [35]: # explicitly by position  
s2.iloc[[3, 2]]  
  
Out[35]: 13    4  
          12    3  
          dtype: int64
```

Note that if you specify a location that does not exist (below zero, or greater than the number of items— one), then an exception will be thrown.

Explicit lookup by labels with .loc[]

Lookup by label can also be achieved by using the `.loc[]` property:

```
In [36]: # explicit via labels  
s1.loc[['a', 'd']]
```

```
Out[36]: a    10  
          d    13  
          dtype: int64
```

There are no problems using integer labels:

```
In [37]: # get items at position 11 and 12  
s2.loc[[11, 12]]
```

```
Out[37]: 11    2  
          12    3  
          dtype: int64
```

Note that `.loc[]` has a different behavior than `.iloc[]` when passing an index label that is not in the index. In that case, pandas will return a `NaN` value instead of throwing an exception:

```
In [38]: # -1 and 15 will be NaN  
s1.loc[['a', 'f']]
```

```
Out[38]: a    10.0  
          f    NaN  
          dtype: float64
```

 *What is `NaN`? We will see this in more detail later in the chapter, but pandas uses it for the representation of missing data or numbers that can't be found through index lookups. It also has ramifications in various statistical methods that we will also examine later in this chapter.*

Slicing a Series into subsets

pandas `Series` support a feature called **slicing**. Slicing is a powerful way to retrieve subsets of data from a pandas object. Through slicing, we can select data based upon position or index labels and have greater control over the sequencing of the items that result (forwards or reverse) and the interval (every item, every other).

Slicing overloads the normal array `[]` operator (and also `.loc[]`, `.iloc[]`, and `.ix[]`) to accept a **slice object**. A slice object is created using a syntax of `start:end:step`, the components representing the first item, last item, and the increment between each item that you would like as the `step`.

Each component of the slice is optional and provides a convenient means to select entire rows by omitting a component of the slice specifier.

To start demonstrating slicing, we will use the following `Series`:

```
In [39]: # a Series to use for slicing
# using index labels not starting at 0 to demonstrate
# position based slicing
s = pd.Series(np.arange(100, 110), index=np.arange(10, 20))
s
```

```
Out[39]: 10    100
11    101
12    102
13    103
14    104
15    105
16    106
17    107
18    108
19    109
dtype: int64
```

We can select consecutive items using `start:end` for the slice. The following selects the five items in positions 1 through 5 in the `Series`. Since we did not specify a `step` component, it defaults to 1. Also note that the `end` label is not included in the result:

```
In [40]: # slice showing items at position 1 thorough 5
s[1:6]
```

```
Out[40]: 11    101
12    102
13    103
14    104
15    105
dtype: int64
```

This result is roughly equivalent to the following:

```
In [41]: # lookup via list of positions
s.iloc[[1, 2, 3, 4, 5]]
```

```
Out[41]: 11    101
12    102
13    103
14    104
15    105
dtype: int64
```

It is roughly equivalent as this use of `.iloc[]` returns a copy of the data in the source. A slice is a reference to the data in the source. Modification of contents of the resulting slice will affect the source `Series`. We

will examine this process further in a later section on modifying `Series` data in place.

A slice can return every other item by specifying a step of 2:

```
In [42]: # items at position 1, 3, 5  
s[1:6:2]  
  
Out[42]: 11    101  
13    103  
15    105  
dtype: int64
```

As stated earlier, each component of the slice is optional. If the `start` component is omitted, the results will start at the first item. As an example, the following is a shorthand for `.head()`:

```
In [43]: # first five by slicing, same as .head(5)  
s[:5]  
  
Out[43]: 10    100  
11    101  
12    102  
13    103  
14    104  
dtype: int64
```

All items at and after a specific position can be selected by specifying the `start` component and omitting the `end`. The following selects all items, starting with the 4th:

```
In [44]: # fourth position to the end  
s[4:]  
  
Out[44]: 14    104  
15    105  
16    106  
17    107  
18    108  
19    109  
dtype: int64
```

A `step` can also be used in both of the two previous scenarios to skip over items:

```
In [45]: # every other item in the first five positions  
s[:5:2]  
  
Out[45]: 10    100  
12    102  
14    104  
dtype: int64
```

```
In [46]: # every other item starting at the fourth position  
s[4::2]  
  
Out[46]: 14    104  
16    106  
18    108  
dtype: int64
```

Use of a negative `step` value will reverse the result. The following demonstrates how to reverse the `Series`:

```
In [47]: # reverse the Series
s[::-1]

Out[47]: 19    109
18    108
17    107
16    106
15    105
14    104
13    103
12    102
11    101
10    100
dtype: int64
```

A value of `-2` will return every other item from the start position, working towards the beginning of the series in reverse order. The following example returns every other item before and including the row at position 4:

```
In [48]: # every other starting at position 4, in reverse
s[4::-2]

Out[48]: 14    104
12    102
10    100
dtype: int64
```

Negative values for the `start` and `end` of a slice have special meaning. A negative `start` value of `-n` means the last `n` rows:

```
In [49]: # -4:, which means the last 4 rows
s[-4:]

Out[49]: 16    106
17    107
18    108
19    109
dtype: int64
```

A negative `end` value of `-n` will return all but the last `n` rows:

```
In [50]: # :-4, all but the last 4
s[:-4]

Out[50]: 10    100
11    101
12    102
13    103
14    104
15    105
dtype: int64
```

Negative `start` and `end` components can be combined. The following first retrieves the last four rows, and then, from those, all but the last one (so the first three):

```
In [51]: # equivalent to s.tail(4).head(3)
s[-4:-1]

Out[51]: 16    106
17    107
18    108
dtype: int64
```

It is also possible to slice a series with a non-integer index. To demonstrate, let's use the following series:

```
In [52]: # used to demonstrate the next two slices
s = pd.Series(np.arange(0, 5),
              index=['a', 'b', 'c', 'd', 'e'])
s

Out[52]: a    0
          b    1
          c    2
          d    3
          e    4
         dtype: int64
```

Using this Series, slicing with integer values will extract items based on position (as before):

```
In [53]: # slices by position as the index is characters
s[1:3]

Out[53]: b    1
          c    2
         dtype: int64
```

But, when using non-integer values as components for the slice, pandas will attempt to understand the data type and pick the appropriate items from the series. As an example, the following slices from 'b' through 'd':

```
In [54]: # this slices by the strings in the index
s['b':'d']

Out[54]: b    1
          c    2
          d    3
         dtype: int64
```


Alignment via index labels

Alignment of `Series` data by index labels is a fundamental concept in pandas, as well as being one of its most powerful concepts. Alignment provides automatic correlation of related values in multiple Series objects based upon index labels. This saves a lot of error-prone effort matching data in multiple sets using standard procedural techniques.

To demonstrate alignment, let's perform an example of adding values in two `Series` objects. Let's start with the following two `Series` objects representing two different samples of a set of variables (`a` and `b`):

```
In [55]: # First series for alignment
s1 = pd.Series([1, 2], index=['a', 'b'])
s1
```



```
Out[55]: a    1
          b    2
          dtype: int64
```

```
In [56]: # Second series for alignment
s2 = pd.Series([4, 3], index=['b', 'a'])
s2
```



```
Out[56]: b    4
          a    3
          dtype: int64
```

Now suppose we would like to total the values for each variable. We can express this simply as `s1 + s2`:

```
In [57]: # add them
s1 + s2
```



```
Out[57]: a    4
          b    6
          dtype: int64
```

pandas has matched the measurement for each variable in each series, added those values, and returned us the sum for each in one succinct statement.

It is also possible to apply a scalar value to a `Series`. The result will be that the scalar will be applied to each value in the `Series` using the specified operation:

```
In [58]: # multiply all values in s3 by 2
s1 * 2
```



```
Out[58]: a    2
          b    4
          dtype: int64
```

Remember earlier when it was stated that we would come back to creating a `Series` with a scalar value? When performing this type of operation, pandas actually performs the following actions:

```
In [59]: # scalar series using s3's index
t = pd.Series(2, s1.index)
t
```



```
Out[59]: a    2
          b    2
          dtype: int64
```

```
In [60]: # multiply s1 by t  
s1 * t
```



```
Out[60]: a    2  
         b    4  
        dtype: int64
```

The first step is the creation of a `Series` from the scalar value, but with the index of the target `Series`. The multiplication is then applied to the aligned values of the two `Series` objects, which perfectly align because the index is identical.

The labels in the indexes are not required to align. Where alignment does not occur, pandas will return `NaN` as the result:

```
In [61]: # we will add this to s1  
s3 = pd.Series([5, 6], index=['b', 'c'])  
s3
```



```
Out[61]: b    5  
         c    6  
        dtype: int64
```

```
In [62]: # s1 and s3 have different sets of index labels  
# NaN will result for a and c  
s1 + s3
```



```
Out[62]: a    NaN  
         b    7.0  
         c    NaN  
        dtype: float64
```

The `NaN` value is, by default, the result of any pandas alignment where an index label does not align with the other `Series`. This is an important characteristic of pandas, when compared to NumPy. If labels do not align, there should not be an exception thrown. This helps when some data is missing but it is acceptable for this to happen. Processing continues, but pandas lets you know there's an issue (but not necessarily a problem) by returning `NaN`.

Labels in a pandas index do not need to be unique. The alignment operation actually forms a Cartesian product of the labels in the two `Series`. If there are n 'a' labels in series 1, and m labels in series 2, then the result will have $n*m$ total rows in the result.

To demonstrate this let's use the following two `Series` objects:

```
In [63]: # 2 'a' labels  
s1 = pd.Series([1.0, 2.0, 3.0], index=['a', 'a', 'b'])  
s1
```



```
Out[63]: a    1.0  
         a    2.0  
         b    3.0  
        dtype: float64
```

```
In [64]: # 3 a labels  
s2 = pd.Series([4.0, 5.0, 6.0, 7.0], index=['a', 'a', 'c', 'a'])  
s2
```



```
Out[64]: a    4.0  
         a    5.0  
         c    6.0  
         a    7.0  
        dtype: float64
```

This will result in 6 'a' index labels and `NaN` for 'b' and 'c':

```
In [65]: # will result in 6 'a' index labels, and NaN for b and c
s1 + s2

Out[65]: a    5.0
          a    6.0
          a    8.0
          a    6.0
          a    7.0
          a    9.0
          b    NaN
          c    NaN
dtype: float64
```


Performing Boolean selection

Indexes give us a very powerful and efficient means of looking up values in a `Series` based upon their labels. But what if you want to look up entries in a `Series` based upon the values?

To handle this scenario pandas provides us with Boolean selection. A Boolean selection applies a logical expression to the values of the `Series` and returns a new series of Boolean values representing the result of that expression upon each value. This result can then be used to extract only values where `True` was a result.

To demonstrate Boolean selection, let's start with the following `Series` and apply the greater than operator to determine values greater than or equal to 3:

```
In [66]: # which rows have values that are > 5?
s = pd.Series(np.arange(0, 5), index=list('abcde'))
logical_results = s >= 3
logical_results
```



```
Out[66]: a    False
         b    False
         c    False
         d    True
         e    True
        dtype: bool
```

This results in a `Series` with matching index labels and the result of the expression as applied to the value of each label. The `dtype` of the values is `bool`.

This series can then be used to select values from the original series. This selection is performed by passing the Boolean results to the `[]` operator of the source.

```
In [67]: # select where True
s[logical_results]
```



```
Out[67]: d    3
         e    4
        dtype: int64
```

The syntax can be simplified by performing the logical operation within the `[]` operator:

```
In [68]: # a little shorter version
s[s > 5]
```



```
Out[68]: Series([], dtype: int64)
```

Unfortunately, multiple logical operators cannot be used in a normal Python syntax. As an example, the following causes an exception to be thrown:

```
In [69]: # commented as it throws an exception
# s[s >= 2 and s < 5]
```

There are technical reasons for why the preceding code does not work. The solution is to express the equation differently, putting parentheses around each of the logical conditions and using different

```
In [70]: # correct syntax
s[(s >= 2) & (s < 5)]
```



```
Out[70]: c    2
         d    3
         e    4
        dtype: int64
```

operators for and/or (`|` and `&`):

It is possible to determine whether all the values in a series match a given expression using the `.all()` method. The following asks if all elements in the series are greater than or equal to 0:

```
In [71]: # are all items >= 0?  
(s >= 0).all()  
Out[71]: True
```

The `.any()` method returns `True` if any value satisfies the expressions. The following asks if any element is less than 2:

```
In [72]: # any items < 2?  
s[s < 2].any()  
Out[72]: True
```

You can determine how many items satisfied the expression using the `.sum()` method on the resulting selection. This is because the `.sum()` method of a series when given a series of Boolean values will treat

```
In [73]: # how many values < 2?  
(s < 2).sum()  
Out[73]: 2
```

True as 1 and False as 0:

Re-indexing a Series

Re-indexing in pandas is a process that makes the data in a `Series` conform to a set of labels. It is used by pandas to perform much of the alignment process and is hence a fundamental operation.

Re-indexing achieves several things:

- Re-ordering existing data to match a set of labels
- Inserting `NaN` markers where no data exists for a label
- Possibly filling missing data for a label using some type of logic (defaulting to adding `NaN` values)

Re-indexing can be as simple as simply assigning a new index to the `.index` property of a `Series`. The following demonstrates changing the index of a `Series` in this manner:

```
In [74]: # sample series of five items
np.random.seed(123456)
s = pd.Series(np.random.randn(5))
s

Out[74]: 0    0.469112
1   -0.282863
2   -1.509059
3   -1.135632
4    1.212112
dtype: float64
```



```
In [75]: # change the index
s.index = ['a', 'b', 'c', 'd', 'e']
s

Out[75]: a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
e    1.212112
dtype: float64
```



The number of elements in the list being assigned to the `.index` property must match the number of rows or an exception will be thrown. Re-indexing also modified the `Series` in-place.

Flexibility in creating a new index is provided through use of the `.reindex()` method. One case is in assigning a new index where the number of labels does not match the number of values:

```
In [76]: # a series that we will reindex
np.random.seed(123456)
s1 = pd.Series(np.random.randn(4), ['a', 'b', 'c', 'd'])
s1

Out[76]: a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
dtype: float64
```

The following re-indexes the `Series` using a set of labels that has new, missing, and overlapping values:

```
In [77]: # reindex with different number of labels
# results in dropped rows and/or NaN's
s2 = s1.reindex(['a', 'c', 'g'])
s2

Out[77]: a    0.469112
c   -1.509059
g      NaN
dtype: float64
```

There are several things here that are important to point out about `.reindex()`. The first is that the result of a `.reindex()` method is a new `Series` and not an in-place modification. The new `Series` has an index with labels, as specified in the passing to the function. The data is copied for each label that exists in the original `Series`. If a label is not found in the original `Series`, then `NaN` will be assigned as the value. Finally, rows in the `Series` with labels that are not in the new index are dropped.

Re-indexing is also useful when you want to align two `Series` to perform an operation on values in two `Series` but the `Series` objects do not have labels that align for some reason. A common scenario is that one `Series` has labels of integer type and the other is strings, but the underlying meaning of the values is the same (this is common when getting data from remote sources). Take the following `Series` objects as an example:

```
In [78]: # different types for the same values of labels
# causes big trouble
s1 = pd.Series([0, 1, 2], index=[0, 1, 2])
s2 = pd.Series([3, 4, 5], index=['0', '1', '2'])
s1 + s2

Out[78]: 0    NaN
          1    NaN
          2    NaN
          0    NaN
          1    NaN
          2    NaN
          dtype: float64
```

Although the meaning of the labels in the two `Series` is the same, they will align due to their data types being different. This is easily fixed once the problem is identified:

```
In [79]: # reindex by casting the label types
# and we will get the desired result
s2.index = s2.index.values.astype(int)
s1 + s2

Out[79]: 0    3
          1    5
          2    7
          dtype: int64
```

The `.reindex()` method has the default action of inserting `NaN` as a missing value when labels are not found in the source `Series`. This value can be changed by using the `fill_value` parameter. The following example demonstrates using `0` instead of `NaN`:

```
In [80]: # fill with 0 instead of NaN
s2 = s1.copy()
s2.reindex(['a', 'f'], fill_value=0)

Out[80]: a    0.469112
          f    0.000000
          dtype: float64
```

When performing a re-index on ordered data such as a time series, it is possible to perform interpolation, or filling of values. There will be a more elaborate discussion on interpolation and filling in Chapter 10, *Time Series Data*, but the following examples introduce the concept. Let's start with the following `Series`:

```
In [81]: # create example to demonstrate fills
s3 = pd.Series(['red', 'green', 'blue'], index=[0, 3, 5])
s3

Out[81]: 0    red
          3    green
          5    blue
          dtype: object
```

The following example demonstrates the concept of forward filling, often referred to as **last known value**. The `Series` is re-indexed to create a contiguous integer index, and by using the `method='ffill'` parameter, any new index labels are assigned the previously known **non-NaN** value:

```
In [82]: # forward fill example
s3.reindex(np.arange(0,7), method='ffill')

Out[82]: 0    red
          1    red
          2    red
          3  green
          4  green
          5  blue
          6  blue
dtype: object
```

Index labels 1 and 2 are matched to red at label 0, 4 and 5 to green from label 3, and 6 to blue from label 5.

The following example fills backward using `method='bfill'`:

```
In [83]: # backwards fill example
s3.reindex(np.arange(0,7), method='bfill')

Out[83]: 0    red
          1  green
          2  green
          3  green
          4  blue
          5  blue
          6    NaN
dtype: object
```

Label 6 did not have a previous value, so it is set to `NaN`; 4 is set to the value of 5 (`blue`); and 2 and 1 to the value of label 3 (`green`).

Modifying a Series in-place

In-place modification of a `Series` is a slightly controversial topic. When possible, it is preferred to perform operations that return a new `Series` with the modifications represented in the new `Series`. But, if needed, it is possible to change values and add/remove rows in-place.

An additional row can be added in place to a series by assigning a value to an `index` label that does not already exist. The following code creates a `Series` object and adds an additional item to the series:

```
In [84]: # generate a Series to play with
np.random.seed(123456)
s = pd.Series(np.random.randn(3), index=['a', 'b', 'c'])
s

Out[84]: a    0.469112
          b   -0.282863
          c   -1.509059
          dtype: float64
```

```
In [85]: # change a value in the Series
# this is done in-place
# a new Series is not returned that has a modified value
s['d'] = 100
s

Out[85]: a    0.469112
          b   -0.282863
          c   -1.509059
          d   100.000000
          dtype: float64
```

The value at a specific index label can be changed in place by assignment:

```
In [86]: # modify the value at 'd' in-place
s['d'] = -100
s

Out[86]: a    0.469112
          b   -0.282863
          c   -1.509059
          d  -100.000000
          dtype: float64
```

Rows can be removed from a `Series` by passing their `index` labels to the `del()` function. The following demonstrates removal of the row with the index label '`a`':

```
In [87]: # remove a row / item
del(s['a'])
s

Out[87]: b   -0.282863
          c   -1.509059
          d  -100.000000
          dtype: float64
```

 To add and remove items out of place, you use `pd.concat()` to add and remove using a Boolean selection.

An important thing to keep in mind when using slicing is that the result of the slice is a view into the original `Series`. Modification of values through the result of the slice operation will modify the original `Series`.

Consider the following example, which selects the first two elements in the `series` and stores them in a new variable:

```
In [88]: copy = s.copy() # preserve s
          slice = copy[:2] # slice with first two rows
          slice

Out[88]: b    -0.282863
          c    -1.509059
          dtype: float64
```

The following assignment of a value to an element of a slice will change the value in the original `series`:

```
In [89]: # change item with label 10 to 1000
          slice['b'] = 0
          # and see it in the source
          copy

Out[89]: b      0.000000
          c     -1.509059
          d    -100.000000
          dtype: float64
```


Summary

In this chapter, you learned about the pandas `Series` object and how it can be used to represent an indexed representation of variable measurements. We started with how to create and initialize a `Series` and its associated index, and then examined how to manipulate the data in one or more `Series` objects. We examined how to align `Series` objects by index label and apply mathematical operations across the aligned values. We then examined how to look up data by index, as well as how to perform queries based upon the data (Boolean expressions). We then closed with an examination of how to use re-indexing to change indexes and align data.

In the next chapter, you will learn how the `DataFrame` is used to represent multiple `Series` of data in a uniform tabular structure.

Representing Tabular and Multivariate Data with the DataFrame

The pandas `DataFrame` object extends the capabilities of the `Series` object into two-dimensions. Instead of a single series of values, each row of a data frame can have multiple values, each of which is represented as a column. Each row of a data frame can then model multiple related properties of a subject under observation, and with each column being able to represent different types of data.

Each column of a data frame is a pandas `Series`, and a data frame can be considered a form of data like a spreadsheet or a database table. But these comparisons do not do the `DataFrame` justice, as a data frame has very distinct qualities specific to pandas, such as automatic data alignment of the `Series` objects that represent the columns.

This automatic alignment makes a data frame much more capable of exploratory data analysis than spreadsheets or databases. Combined with the ability to slice data simultaneously across both rows and columns, this ability to interact with and explore data in a data frame is incredibly effective for finding the required information.

In this chapter, we will dive into the pandas `DataFrame`. Many of the concepts will be familiar to the `Series`, but with the addition of data and tools to support its manipulation. Specifically, in this chapter we will cover the following topics:

- Creating a `DataFrame` from Python objects, NumPy functions, Python dictionaries, pandas `Series` objects, and CSV files
- Determining the size of the dimensions of a data frame
- Specifying and manipulating the names of the columns in a data frame
- Alignment of rows during the creation of a data frame
- Selecting specific columns and rows of a data frame
- Applying slicing to a data frame
- Selecting rows and columns of a data frame by location and label
- Scalar value lookup
- Boolean selection as applied to a data frame

Configuring pandas

We start the examples in the chapter using the following imports and configuration statements:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set some pandas options controlling output format
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 80)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```


Creating DataFrame objects

There are a number of ways to create a data frame. A data frame can be created from either a single or multi-dimensional set of data. The techniques that we will examine are as follows:

- Using the results of NumPy functions
- Using data from a Python dictionary consisting of lists or pandas `Series` objects
- Using data from a CSV file

While examining each of these we will also examine how to specify column names, demonstrate how alignment is performed during initialization, and see how to determine the dimensions of a data frame.

Creating a DataFrame using NumPy function results

A data frame can be created from a one-dimensional NumPy array of integers ranging from 1 to 5:

```
In [2]: # From a 1-d array
pd.DataFrame(np.arange(1, 6))

Out[2]: 0
         0 1
         1 2
         2 3
         3 4
         4 5
```

The first column of the output shows the labels of the index that was created. Since an index was not specified at the time of creation, pandas created a based `RangeIndex` with labels starting at 0.

The data is in the second column and consists of the values 1 through 5. The 0 above the column of data is the name of the column. When the column names are not specified at the time of the creation of the data frame, pandas names the columns with incremental integers starting at 0.

A multi-dimensional NumPy array can also be used and results in the creation of multiple columns:

```
In [3]: # create a DataFrame from a 2-d ndarray
df = pd.DataFrame(np.array([[10, 11], [20, 21]]))
df

Out[3]:   0   1
          0  10  11
          1  20  21
```

The columns of a `DataFrame` can be accessed using the `.columns` property:

```
In [4]: # retrieve the columns index
df.columns

Out[4]: RangeIndex(start=0, stop=2, step=1)
```

This shows that, when column names are not specified, pandas will create a `RangeIndex` to represent the columns.

Column names can be specified using the `columns` parameter. The following creates a two-column `DataFrame` that represents two samples of temperatures for two cities:

```
In [5]: # specify column names
df = pd.DataFrame(np.array([[70, 71], [90, 91]]),
                  columns=['Missoula', 'Philadelphia'])
df

Out[5]:    Missoula  Philadelphia
          0        70            71
          1        90            91
```

The number of rows in a `DataFrame` can be found using the `len()` function:

```
In [6]: # how many rows?
len(df)

Out[6]: 2
```

The dimensions of the `DataFrame` can be found using the `.shape` property:

```
In [7]: # what is the dimensionality  
df.shape
```

```
Out[7]: (2, 2)
```


Creating a DataFrame using a Python dictionary and pandas Series objects

A Python dictionary can be used to initialize a `DataFrame`. When using a Python dictionary, pandas will use the keys as the column names and the values for each key as the data in the column:

```
In [8]: # initialization using a python dictionary
temp_missoula = [70, 71]
temp_philly = [90, 91]
temperatures = {'Missoula': temp_missoula,
                 'Philadelphia': temp_philly}
pd.DataFrame(temperatures)

Out[8]:    Missoula  Philadelphia
          0         70            90
          1         71            91
```

A common technique of creating a `DataFrame` is by using a list of pandas `Series` objects that will be used as the rows:

```
In [9]: # create a DataFrame for a list of Series objects
temp_at_time0 = pd.Series([70, 90])
temp_at_time1 = pd.Series([71, 91])
df = pd.DataFrame([temp_at_time0, temp_at_time1])
df

Out[9]:    0   1
          0  70  90
          1  71  91
```

In this scenario, each `Series` represents a single measurement for each city at a specific measurement interval.

To name the columns, we could attempt to use the `columns` parameter:

```
In [10]: # try to specify column names
df = pd.DataFrame([temp_at_time0, temp_at_time1],
                  columns=['Missoula', 'Philadelphia'])
df

Out[10]:    Missoula  Philadelphia
          0       NaN        NaN
          1       NaN        NaN
```

This result is different from what we perhaps expected, as the values have been filled with `NaN`. This can be rectified in two ways. The first is to assign the column names to the `.columns` property:

```
In [11]: # specify names of columns after creation
df = pd.DataFrame([temp_at_time0, temp_at_time1])
df.columns = ['Missoula', 'Philadelphia']
df

Out[11]:    Missoula  Philadelphia
          0         70            90
          1         71            91
```

Another technique is to use a Python dictionary where the keys are the column names and the value for each key is a `Series` representing the measurements in that specific column:

```
In [12]: # construct using a dict of Series objects
temp_mso_series = pd.Series(temp_missoula)
temp_phl_series = pd.Series(temp_philly)
df = pd.DataFrame({'Missoula': temp_mso_series,
                    'Philadelphia': temp_phl_series})
df

Out[12]:    Missoula  Philadelphia
0            70             90
1            71             91
```

Be aware that during construction of a `DataFrame`, the `Series` that are supplied will be aligned. The following demonstrates this by adding a third city whose index values are different:

```
In [13]: # alignment occurs during creation
temp_nyc_series = pd.Series([85, 87], index=[1, 2])
df = pd.DataFrame({'Missoula': temp_mso_series,
                    'Philadelphia': temp_phl_series,
                    'New York': temp_nyc_series})
df

Out[13]:    Missoula  New York  Philadelphia
0            70.0      NaN        90.0
1            71.0      85.0        91.0
2            NaN       87.0      NaN
```


Creating a DataFrame from a CSV file

A data frame can be created by reading data from a CSV file using the `pd.read_csv()` function.



`pd.read_csv()` will be more extensively examined in Chapter 9, Accessing Data.

To demonstrate this process, we will load data from a file that contains a snapshot of the S&P 500. This file is named `sp500.csv` and is located in the code bundle's `data` directory.

The first line of the file has the names of each variable/column, and the remaining 500 lines represent the values for the 500 different stocks.

The following code loads the data, while specifying which column in the file to use for the index and also that we only want four specific columns (0, 2, 3, and 7):

```
In [14]: # read in the data and print the first five rows
# use the Symbol column as the index, and
# only read in columns in positions 0, 2, 3, 7
sp500 = pd.read_csv("data/sp500.csv",
                     index_col='Symbol',
                     usecols=[0, 2, 3, 7])
```

Examining the first five rows using `.head()` shows us the following structure and contents of the resulting

```
In [15]: # peek at the first 5 rows of the data using .head()
sp500.head()

Out[15]:
          Symbol      Sector   Price  Book Value
        MMM    Industrials  141.14  26.668
        ABT    Health Care  39.60  15.573
        ABBV   Health Care  53.95  2.954
        ACN    Information Technology  79.79  8.326
        ACE    Financials  102.91  86.897
```

data frame:

Let's examine a few properties of this data frame. It should have 500 rows of data. This can be checked by examining the length of the data frame:

```
In [16]: # how many rows of data? Should be 500
len(sp500)

Out[16]: 500
```

We expect that it has a shape of 500 rows and three columns:

```
In [17]: # what is the shape?
sp500.shape

Out[17]: (500, 3)
```

The size of the data frame can be found using the `.size` property. This property returns the number of data values in the data frame. We would expect $500 \times 3 = 1,500$:

```
In [18]: # what is the size?
sp500.size

Out[18]: 1500
```

The index of the data frame consists of the symbols for the 500 stocks:

```
In [19]: # examine the index  
sp500.index
```



```
Out[19]: Index(['MMM', 'ABT', 'ABBV', 'ACN', 'ACE', 'ACT', 'ADBE',  
               'AES', 'AET', 'AFL',  
               ...  
               'XEL', 'XRX', 'XLNX', 'XL', 'XYL', 'YHOO', 'YUM',  
               'ZMH', 'ZION', 'ZTS'],  
              dtype='object', name='Symbol', length=500)
```

The columns consist of the following three names:

```
In [20]: # get the columns  
sp500.columns
```



```
Out[20]: Index(['Sector', 'Price', 'Book Value'], dtype='object')
```

Note that, although we specified four columns when loading, the result only contains three columns because one of the four columns in the source file was used for the index.

Accessing data within a DataFrame

A data frame consists of both rows and columns, and has constructs to select data from specific rows and columns. These selections use the same operators as a `Series`, including `[]`, `.loc[]`, and `.iloc[]`.

Because of the multiple dimensions, the process by which these are applied differs slightly. We will examine these by first learning to select columns, then rows, a combination of rows and columns in a single statement, and also by using Boolean selections.

Additionally, pandas provides a construct for selecting a single scalar value at a specific row and column that we will investigate. This technique is important and exists because it is a very high-performance means of accessing these values.

Selecting the columns of a DataFrame

Selecting the data in specific columns of a `DataFrame` is performed by using the `[]` operator. This differs from a `Series`, where `[]` specified rows. The `[]` operator can be passed either a single object or a list of objects representing the columns to retrieve.

The following retrieves the column with the name `'Sector'`:

```
In [21]: # retrieve the Sector column
sp500['Sector'].head()

Out[21]: Symbol
          MMM      Industrials
          ABT      Health Care
          ABBV     Health Care
          ACN  Information Technology
          ACE   Financials
Name: Sector, dtype: object
```

When a single column is retrieved from a `DataFrame`, the result is a `Series`:

```
In [22]: type(sp500['Sector'])

Out[22]: pandas.core.series.Series
```

Multiple columns can be retrieved by specifying a list of column names:

```
In [23]: # retrieve the Price and Book Value columns
sp500[['Price', 'Book Value']].head()

Out[23]:    Price  Book Value
Symbol
  MMM      141.14      26.668
  ABT       39.60      15.573
  ABBV      53.95      2.954
  ACN       79.79      8.326
  ACE      102.91      86.897
```

Since this has multiple columns, the result is a `DataFrame` instead of a `Series`:

```
In [24]: # show that this is a DataFrame
type(sp500[['Price', 'Book Value']])

Out[24]: pandas.core.frame.DataFrame
```

Columns can also be retrieved by attribute access. A `DataFrame` will have properties added that represent the names of each column, as long as the name does not contain spaces. The following retrieves the `Price`

```
In [25]: # attribute access of column by name
sp500.Price

Out[25]: Symbol
          MMM      141.14
          ABT       39.60
          ABBV      53.95
          ACN       79.79
          ACE      102.91
          ...
          YHOO      35.02
          YUM        74.77
          ZMH      101.84
          ZION      28.43
          ZTS       30.53
Name: Price, Length: 500, dtype: float64
```

column in this manner:

Note that this will not work for the `Book Value` column, as the name has a space.

Selecting rows of a DataFrame

Rows can be retrieved via an index label value using `.loc[]`:

```
In [26]: # get row with label MMM  
# returned as a Series  
sp500.loc['MMM']
```

```
Out[26]: Sector      Industrials  
          Price       141.14  
          Book Value   26.668  
          Name: MMM, dtype: object
```

Furthermore, multiple rows can be retrieved using a list of labels:

```
In [27]: # rows with label MMM and MSFT  
# this is a DataFrame result  
sp500.loc[['MMM', 'MSFT']]
```

```
Out[27]:           Sector  Price  Book Value  
Symbol  
MMM            Industrials  141.14    26.668  
MSFT          Information Technology  40.12    10.584
```

Rows can be retrieved by location using `.iloc[]`:

```
In [28]: # get rows in location 0 and 2  
sp500.iloc[[0, 2]]
```

```
Out[28]:           Sector  Price  Book Value  
Symbol  
MMM            Industrials  141.14    26.668  
ABBV           Health Care  53.95     2.954
```

It is possible to look up the location in the index of a specific label value and then use that value to

```
In [29]: # get the location of MMM and A in the index  
i1 = sp500.index.get_loc('MMM')  
i2 = sp500.index.get_loc('A')  
(i1, i2)
```

retrieve the row by position:

```
Out[29]: (0, 10)
```

```
In [30]: # and get the rows  
sp500.iloc[[i1, i2]]
```

```
Out[30]:           Sector  Price  Book Value  
Symbol  
MMM            Industrials  141.14    26.668  
A              Health Care  56.18    16.928
```

As a final note in this section, these operations are also possible using `.ix[]`. However, this has been deprecated. For more details, see <http://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing>.

Scalar lookup by label or location using .at[] and .iat[]

Individual scalar values can be looked up by label using `.at[]`, passing it both the row label and the column name:

```
In [31]: # by label in both the index and column  
sp500.at['MMM', 'Price']  
Out[31]: 141.13999999999999
```

Scalar values can also be looked up by location using `.iat[]`, passing both the row location and then the column location. This is the preferred method of accessing single values and gives the highest

```
In [32]: # by location. Row 0, column 1  
sp500.iat[0, 1]  
Out[32]: 141.13999999999999
```

performance:

Slicing using the [] operator

Slicing a `DataFrame` across its index is syntactically identical to performing the same operation with a `Series`. Because of this, we will not go into the details of the various permutations of slices in this section and only look at a few representative examples applied to a `DataFrame`.

When slicing using the `[]` operator, the slice is performed across the index and not the columns. The following retrieves the first five rows:

```
In [33]: # first five rows
sp500[:5]
```

Symbol	Sector	Price	Book Value
MMM	Industrials	141.14	26.668
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954
ACN	Information Technology	79.79	8.326
ACE	Financials	102.91	86.897

And the following returns rows starting with the `ABT` label through the `ACN` labels:

```
In [34]: # ABT through ACN labels
sp500['ABT':'ACN']
```

Symbol	Sector	Price	Book Value
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954
ACN	Information Technology	79.79	8.326

Slicing a `DataFrame` also applies to the `.iloc[]` and `.loc[]` properties. Using these properties is considered best practice.

Selecting rows using Boolean selection

Rows can be selected by using Boolean selection. When applied to a data frame, a Boolean selection can utilize data from multiple columns. Consider the following query, which identifies all stocks with a price

```
In [35]: # what rows have a price < 100?
sp500.Price < 100

Out[35]: Symbol
      MMM    False
      ABT     True
      ABBV    True
      ACN     True
      ACE    False
      ...
      YHOO   True
      YUM     True
      ZMH    False
      ZION   True
      ZTS     True
Name: Price, Length: 500, dtype: bool
```

less than 100:

This result can then be applied to the `DataFrame` using the `[]` operator to return only the rows where the

```
In [36]: # now get the rows with Price < 100
sp500[sp500.Price < 100]

Out[36]:      Sector  Price  Book Value
Symbol
ABT        Health Care  39.60  15.573
ABBV       Health Care  53.95  2.954
ACN  Information Technology  79.79  8.326
ADBE  Information Technology  64.30  13.262
AES         Utilities  13.61  5.781
...
...          ...
XYL        Industrials  38.42  12.127
YHOO  Information Technology  35.02  12.768
YUM  Consumer Discretionary  74.77  5.147
ZION        Financials  28.43  30.191
ZTS         Health Care  30.53  2.150
[407 rows x 3 columns]
```

result was True:

Multiple conditions can be put together using parentheses. The following retrieves the symbols and price for all stocks with a price between 6 and 10:

```
In [37]: # get only the Price where Price is < 10 and > 6
r = sp500[(sp500.Price < 10) &
           (sp500.Price > 6)] ['Price']
r

Out[37]: Symbol
      HCBK    9.80
      HBAN    9.10
      SLM     8.82
      WIN     9.38
Name: Price, dtype: float64
```

It is common to perform selection using multiple variables. The following demonstrates this by finding all rows where the `Sector` is `Health Care` and the `Price` is greater than or equal to `100.00`:

```
In [38]: # price > 100 and in the Health Care Sector
r = sp500[(sp500.Sector == 'Health Care') &
           (sp500.Price > 100.00)][['Price', 'Sector']]
r
```

```
Out[38]:      Price      Sector
Symbol
ACT     213.77  Health Care
ALXN    162.30  Health Care
AGN     166.92  Health Care
AMGN    114.33  Health Care
BCR     146.62  Health Care
...
REGN    297.77  Health Care
TMO     115.74  Health Care
WAT     100.54  Health Care
WLP     108.82  Health Care
ZMH     101.84  Health Care
```

[19 rows x 2 columns]

Selecting across both rows and columns

It is a common practice to select a subset of data that consists of a set of rows and columns. The following demonstrates this by first selecting a slice of rows and then the desired columns:

```
In [39]: # select the price and sector columns for ABT and ZTS  
sp500.loc[['ABT', 'ZTS']][['Sector', 'Price']]
```

```
Out[39]:      Sector  Price  
Symbol  
ABT      Health Care  39.60  
ZTS      Health Care  30.53
```


Summary

In this chapter, you learned how to create pandas `DataFrame` objects and various means of selecting data based upon the indexes and values in various columns. These examples paralleled those of the `Series`, but demonstrated that, because the `DataFrame` has columns and an associated columns index, the syntax has variations from the `Series`.

In the next chapter, we will dive further into data manipulation, using the `DataFrame` and focusing on making modifications to `DataFrame` structure and contents.

Manipulating DataFrame Structure

pandas provides a powerful manipulation engine for you to use to explore your data. This exploration often involves making modifications to the structure of `DataFrame` objects to remove unnecessary data, change the format of existing data, or to create derived data from data in other rows or columns. These chapter will demonstrate how to perform these powerful and important operations.

Specifically, in this chapter we will cover:

- Renaming columns
- Adding new columns with `[]` and `.insert()`
- Adding columns through enlargement
- Adding columns using concatenation
- Reordering columns
- Replacing the contents of a column
- Deleting columns
- Adding new rows
- Concatenating rows
- Adding and replacing rows via enlargement
- Removing rows using `.drop()`
- Removing rows using Boolean selection
- Removing rows using a slice

Configuring pandas

The following code will configure the pandas environment for the examples that follow. This also loads the S&P 500 data set so that it can be used in the examples:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# Set some pandas options controlling output format
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# read in the data and print the first five rows
# use the Symbol column as the index, and
# only read in columns in positions 0, 2, 3, 7
sp500 = pd.read_csv("data/sp500.csv",
                    index_col='Symbol',
                    usecols=[0, 2, 3, 7])
```


Renaming columns

Columns can be renamed using the appropriately named `.rename()` method. This method can be passed a dictionary object where the keys represent the labels of the columns that are to be renamed, and the value for each key is the new name.

The following will change the name of the 'Book Value' column to 'BookValue', removing the space and allowing access to that column's data using property notation.

```
In [2]: # rename the Book Value column to not have a space
# this returns a copy with the column renamed
newSP500 = sp500.rename(columns=
                        {'Book Value': 'BookValue'})
# print first 2 rows
newSP500[:2]
```



```
Out[2]:      Sector  Price  BookValue
Symbol
MMM      Industrials  141.14    26.668
ABT      Health Care   39.60    15.573
```

Using `.rename()` in this manner returns a new data frame with the columns renamed and the data copied from the original. The following verifies that the original was not modified.

```
In [3]: # verify the columns in the original did not change
sp500.columns
```



```
Out[3]: Index(['Sector', 'Price', 'Book Value'], dtype='object')
```

To modify the data frame in-place without making a copy you can use the `inplace=True` parameter.

```
In [4]: # this changes the column in-place
sp500.rename(columns=
              {'Book Value': 'BookValue'},
              inplace=True)
# we can see the column is changed
sp500.columns
```



```
Out[4]: Index(['Sector', 'Price', 'BookValue'], dtype='object')
```

It is now possible to use the `.BookValue` property to access the data.

```
In [5]: # and now we can use .BookValue
sp500.BookValue[:5]
```



```
Out[5]: Symbol
MMM      26.668
ABT      15.573
ABBV     2.954
ACN      8.326
ACE      86.897
Name: BookValue, dtype: float64
```


Adding new columns with [] and .insert()

New columns can be added to a data frame using the `[]` operator. Let's add a new column named `RoundedPrice` which will represent the rounding of values in the `Price` column.

```
In [6]: # make a copy so that we keep the original data unchanged
sp500_copy = sp500.copy()
# add the new column
sp500_copy['RoundedPrice'] = sp500.Price.round()
sp500_copy[:2]
```

```
Out[6]:      Sector  Price  BookValue  RoundedPrice
Symbol
MMM    Industrials  141.14    26.668        141.0
ABT    Health Care   39.60    15.573        40.0
```

The way that pandas performs this is by first selecting the `Price` column's data from `sp500` and then all of the values in that `Series` are rounded. Then pandas aligns this new `Series` with the `DataFrame` and adds it as a new column named `RoundedPrice`. The new column is added to the end of the columns index.

The `.insert()` method can be used to add a new column at a specific location. The following inserts the `RoundedPrice` column between `Sector` and `Price`:

```
In [7]: # make a copy so that we keep the original data unchanged
copy = sp500.copy()
# insert sp500.Price * 2 as the
# second column in the DataFrame
copy.insert(1, 'RoundedPrice', sp500.Price.round())
copy[:2]
```

```
Out[7]:      Sector  RoundedPrice  Price  BookValue
Symbol
MMM    Industrials        141.0  141.14    26.668
ABT    Health Care         40.0   39.60    15.573
```


Adding columns through enlargement

A column can be added using the `.loc[]` property and a slice. The following demonstrates this by adding a new column to a subset of `sp500` named `PER` with all values initialized to 0.

```
In [8]: # copy of subset / slice
ss = sp500[:3].copy()
# add the new column initialized to 0
ss.loc[:, 'PER'] = 0
# take a look at the results
ss
```

```
Out[8]:      Sector  Price  BookValue  PER
Symbol
MMM    Industrials  141.14    26.668    0
ABT    Health Care   39.60    15.573    0
ABBV   Health Care   53.95     2.954    0
```

A series with existing data can also be added in this manner. The following adds the `PER` column with random data from a series. Since this uses alignment, it is necessary to use the same index as the target data frame.

```
In [9]: # copy of subset / slice
ss = sp500[:3].copy()
# add the new column initialized with random numbers
np.random.seed(123456)
ss.loc[:, 'PER'] = pd.Series(np.random.normal(size=3), index=ss.index)
# take a look at the results
ss
```

```
Out[9]:      Sector  Price  BookValue      PER
Symbol
MMM    Industrials  141.14    26.668  0.469112
ABT    Health Care   39.60    15.573 -0.282863
ABBV   Health Care   53.95     2.954 -1.509059
```


Adding columns using concatenation

Both the `[]` operator and `.insert()` method modify the target data frame in-place. If a new data frame with the additional columns is desired (leaving the original unchanged) then we can use the `pd.concat()` function. This function creates a new data frame with all of the specified `DataFrame` objects concatenated in the order of specification.

This following creates a new `DataFrame` with a single column containing the rounded price. It then uses `pd.concat()` with `axis=1` to signify that the given `DataFrame` objects should be concatenated along the columns axis (as compared to rows which would use `axis=0`) .

```
In [10]: # create a DataFrame with only the RoundedPrice column
rounded_price = pd.DataFrame({'RoundedPrice':
                                sp500.Price.round()})
# concatenate along the columns axis
concatenated = pd.concat([sp500, rounded_price], axis=1)
concatenated[:5]
```

```
Out[10]:
```

	Sector	Price	BookValue	\
Symbol				
MMM	Industrials	141.14	26.668	
ABT	Health Care	39.60	15.573	
ABBV	Health Care	53.95	2.954	
ACN	Information Technology	79.79	8.326	
ACE	Financials	102.91	86.897	

	RoundedPrice
Symbol	
MMM	141.0
ABT	40.0
ABBV	54.0
ACN	80.0
ACE	103.0

 Concatenation will be covered in more detail in Chapter 11, *Combining, Relating, and Reshaping Data*.

It is possible to have duplicate column names as a result of a concatenation. To demonstrate this happening, let's recreate `rounded_price` but name the column `Price`.

```
In [11]: # create a DataFrame with only the RoundedPrice column
rounded_price = pd.DataFrame({'Price': sp500.Price.round()})
rounded_price[:5]
```

```
Out[11]:
```

	Price
Symbol	
MMM	141.0
ABT	40.0
ABBV	54.0
ACN	80.0
ACE	103.0

The concatenation will now result in duplicate columns.

```
In [12]: # this will result in duplicate Price column
dups = pd.concat([sp500, rounded_price], axis=1)
dups[:5]

Out[12]:
```

Symbol	Sector	Price	BookValue	Price
MMM	Industrials	141.14	26.668	141.0
ABT	Health Care	39.60	15.573	40.0
ABBV	Health Care	53.95	2.954	54.0
ACN	Information Technology	79.79	8.326	80.0
ACE	Financials	102.91	86.897	103.0

Interestingly, you can retrieve both columns with the `.Price` property.

```
In [13]: # retrieves both Price columns
dups.Price[:5]

Out[13]:
```

Symbol	Price	Price
MMM	141.14	141.0
ABT	39.60	40.0
ABBV	53.95	54.0
ACN	79.79	80.0
ACE	102.91	103.0

If you want to get a specific `Price` column in the scenario, you will need to retrieve it by position and not name.

Reordering columns

Columns can be reordered out-of-place by selecting the columns in the desired order. The following demonstrates by reversing the columns.

```
In [14]: # return a new DataFrame with the columns reversed
reversed_column_names = sp500.columns[::-1]
sp500[reversed_column_names][:5]
```

```
Out[14]:      BookValue    Price           Sector
Symbol
MMM        26.668   141.14       Industrials
ABT        15.573   39.60       Health Care
ABBV       2.954    53.95       Health Care
ACN        8.326    79.79  Information Technology
ACE        86.897   102.91       Financials
```

 *There is not really a way of changing the order of the columns in-place. See: <http://stackoverflow.com/questions/25878198/change-pandas-dataframe-column-order-in-place>.*

Replacing the contents of a column

The contents of a `DataFrame` can be replaced by assigning a new `Series` to an existing column using the `[]` operator. The following demonstrates replacing the `Price` column with the `Price` column from `rounded_price`.

```
In [15]: # this occurs in-place so let's use a copy
copy = sp500.copy()
# replace the Price column data with the new values
# instead of adding a new column
copy.Price = rounded_price.Price
copy[:5]
```

```
Out[15]:
```

		Sector	Price	BookValue
Symbol				
MMM		Industrials	141.0	26.668
ABT		Health Care	40.0	15.573
ABBV		Health Care	54.0	2.954
ACN	Information	Technology	80.0	8.326
ACE		Financials	103.0	86.897

Data for a column can also be replaced (in-place) using a slice.

```
In [16]: # this occurs in-place so let's use a copy
copy = sp500.copy()
# replace the Price column data with rounded values
copy.loc[:, 'Price'] = rounded_price.Price
copy[:5]
```

```
Out[16]:
```

		Sector	Price	BookValue
Symbol				
MMM		Industrials	141.0	26.668
ABT		Health Care	40.0	15.573
ABBV		Health Care	54.0	2.954
ACN	Information	Technology	80.0	8.326
ACE		Financials	103.0	86.897

Deleting columns

Columns can be deleted from a `DataFrame` by using the `del` keyword or the `.pop()` or `.drop()` method of the data frame. The behavior of each of these differs slightly:

- `del` will simply delete the series from the `DataFrame` (in-place)
- `pop()` will both delete the series and return the series as a result (also in-place)
- `drop(labels, axis=1)` will return a new data frame with the column(s) removed (the original `DataFrame` object is not modified)

The following demonstrates using `del` to delete the `BookValue` column from a copy of the `sp500` data:

```
In [17]: # Example of using del to delete a column
# make a copy as this is done in-place
copy = sp500.copy()
del copy['BookValue']
copy[:2]
```

```
Out[17]:      Sector    Price
Symbol
MMM      Industrials  141.14
ABT      Health Care   39.60
```

The following uses the `.pop()` method to remove the `Sector` column:

```
In [18]: # Example of using pop to remove a column from a DataFrame
# first make a copy of a subset of the data frame as
# pop works in place
copy = sp500.copy()
# this will remove Sector and return it as a series
popped = copy.pop('Sector')
# Sector column removed in-place
copy[:2]
```

```
Out[18]:      Price    BookValue
Symbol
MMM      141.14      26.668
ABT      39.60      15.573
```

The `.pop()` method has the benefit that it gives us the popped columns.

```
In [19]: # and we have the Sector column as the result of the pop
popped[:5]
```

```
Out[19]: Symbol
          Industrials
          Health Care
          Health Care
          Information Technology
          Financials
Name: Sector, dtype: object
```

The `.drop()` method can be used to remove both rows and columns. To use it to remove columns, specify `axis=1`:

Appending new rows

Appending of rows is performed using the `.append()` method of the `DataFrame`. The process of appending returns a new `DataFrame` with the data from the original `DataFrame` added first and then rows from the second. Appending does not perform alignment and can result in duplicate index labels.

The following code demonstrates appending two `DataFrame` objects extracted from the `sp500` data. The first `DataFrame` consists of rows (by position) 0, 1 and 2, and the second consists of rows (also by position) 10, 11 and 2. The row at position 2 (with label `ABBV`) is included in both to demonstrate the creation of duplicate index labels.

```
In [21]: # copy the first three rows of sp500
df1 = sp500.iloc[0:3].copy()
# copy 10th and 11th rows
df2 = sp500.iloc[[10, 11, 2]]
# append df1 and df2
appended = df1.append(df2)
# the result is the rows of the first followed by
# those of the second
appended
```

```
Out[21]:
```

Symbol	Sector	Price	BookValue
MMM	Industrials	141.14	26.668
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954
A	Health Care	56.18	16.928
GAS	Utilities	52.98	32.462
ABBV	Health Care	53.95	2.954

The set of columns of the `DataFrame` objects used in an append do not need to be the same. The resulting data frame will consist of the union of the columns in both, with missing column data filled with `NaN`. The following demonstrates this by creating a third data frame using the same index as `df1` but having a single column with a name not in `df1`.

```
In [22]: # data frame using df1.index and just a PER column
# also a good example of using a scalar value
# to initialize multiple rows
df3 = pd.DataFrame(0.0,
                   index=df1.index,
                   columns=['PER'])
df3
```

```
Out[22]:
```

Symbol	PER
MMM	0.0
ABT	0.0
ABBV	0.0

Now let's append `df3` and `df1`.

```
In [23]: # append df1 and df3  
# each has three rows, so 6 rows is the result  
# df1 had no PER column, so NaN from for those rows  
# df3 had no BookValue, Price or Sector, so NaN's  
df1.append(df3)
```

```
Out[23]:
```

Symbol	BookValue	PER	Price	Sector
MMM	26.668	NaN	141.14	Industrials
ABT	15.573	NaN	39.60	Health Care
ABBV	2.954	NaN	53.95	Health Care
MMM	NaN	0.0	NaN	NaN
ABT	NaN	0.0	NaN	NaN
ABBV	NaN	0.0	NaN	NaN

The `ignore_index=True` parameter can be used to append without forcing the index to be retained from either `DataFrame`. This is useful when the index values are not of significant meaning and you just want concatenated data with sequentially increasing integers as indexes:

```
In [24]: # ignore index labels, create default index  
df1.append(df3, ignore_index=True)
```

```
Out[24]:
```

	BookValue	PER	Price	Sector
0	26.668	NaN	141.14	Industrials
1	15.573	NaN	39.60	Health Care
2	2.954	NaN	53.95	Health Care
3	NaN	0.0	NaN	NaN
4	NaN	0.0	NaN	NaN
5	NaN	0.0	NaN	NaN

Notice that the resulting `DataFrame` has a default `RangeIndex` and that the data from the index (`symbol`) is completely excluded from the result.

Concatenating rows

The rows from multiple `DataFrame` objects can be concatenated to each other using the `pd.concat()` function and by specifying `axis=0`. The default operation of `pd.concat()` on two `DataFrame` objects along the row axis operates in the same way as the `.append()` method.

This is demonstrated by reconstructing the two datasets from the earlier append example and concatenating them instead.

```
In [25]: # copy the first three rows of sp500
df1 = sp500.iloc[0:3].copy()
# copy 10th and 11th rows
df2 = sp500.iloc[[10, 11, 2]]
# pass them as a list
pd.concat([df1, df2])
```

```
Out[25]:      Sector  Price  BookValue
Symbol
MMM    Industrials  141.14    26.668
ABT    Health Care   39.60    15.573
ABBV   Health Care   53.95    2.954
A      Health Care   56.18    16.928
GAS    Utilities     52.98    32.462
ABBV   Health Care   53.95    2.954
```

If the set of columns in all `DataFrame` objects is not identical, pandas will fill those values with `NaN`.

```
In [26]: # copy df2
df2_2 = df2.copy()
# add a column to df2_2 that is not in df1
df2_2.insert(3, 'Foo', pd.Series(0, index=df2.index))
# see what it looks like
df2_2
```

```
Out[26]:      Sector  Price  BookValue  Foo
Symbol
A      Health Care   56.18    16.928    0
GAS    Utilities     52.98    32.462    0
ABBV   Health Care   53.95    2.954    0
```

```
In [27]: # now concatenate
pd.concat([df1, df2_2])
```

```
Out[27]:      BookValue  Foo  Price  Sector
Symbol
MMM        26.668  NaN  141.14  Industrials
ABT        15.573  NaN  39.60  Health Care
ABBV       2.954  NaN  53.95  Health Care
A          16.928  0.0  56.18  Health Care
GAS        32.462  0.0  52.98  Utilities
ABBV       2.954  0.0  53.95  Health Care
```

Duplicate index labels can result as the rows are copied verbatim from the source objects. The keys

parameter can be used to help differentiate which data frame a set of rows originated from. The following demonstrates by using `keys` to add a level to the index representing the source object:

```
In [28]: # specify keys
r = pd.concat([df1, df2_2], keys=['df1', 'df2'])
r
```

```
Out[28]:
```

	Symbol	BookValue	Foo	Price	Sector
df1	MMM	26.668	NaN	141.14	Industrials
	ABT	15.573	NaN	39.60	Health Care
	ABBV	2.954	NaN	53.95	Health Care
df2	A	16.928	0.0	56.18	Health Care
	GAS	32.462	0.0	52.98	Utilities
	ABBV	2.954	0.0	53.95	Health Care

We will examine hierarchical indexes in more detail in Chapter 6, Working with Indexes.



Adding and replacing rows via enlargement

Rows can also be added to a `DataFrame` using the `.loc` property. The parameter for `.loc` specifies the index label where the row is to be placed. If the label does not exist, the values are appended to the data frame using the given index label. If the label does exist, the values in the specified row are replaced.

The following example takes a subset of `sp500` and adds a row with the label `FOO`:

```
In [29]: # get a small subset of the sp500
# make sure to copy the slice to make a copy
ss = sp500[:3].copy()
# create a new row with index label FOO
# and assign some values to the columns via a list
ss.loc['FOO'] = ['the sector', 100, 110]
ss
```

	Sector	Price	BookValue
Symbol			
MMM	Industrials	141.14	26.668
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954
FOO	the sector	100.00	110.000

Note that this change is made in place whether it adds or replaces a row.

Removing rows using .drop()

The `.drop()` method of the `DataFrame` can be used to remove rows. The `.drop()` method takes a list of index labels to drop and returns a copy of the `DataFrame` with the specified rows removed.

```
In [30]: # get a copy of the first 5 rows of sp500
ss = sp500[:5]
ss
```

```
Out[30]:
```

		Sector	Price	BookValue
Symbol				
MMM	Industrials	141.14	26.668	
ABT	Health Care	39.60	15.573	
ABBV	Health Care	53.95	2.954	
ACN	Information Technology	79.79	8.326	
ACE	Financials	102.91	86.897	

```
In [31]: # drop rows with labels ABT and ACN
afterdrop = ss.drop(['ABT', 'ACN'])
afterdrop[:5]
```

```
Out[31]:
```

		Sector	Price	BookValue
Symbol				
MMM	Industrials	141.14	26.668	
ABBV	Health Care	53.95	2.954	
ACE	Financials	102.91	86.897	

Removing rows using Boolean selection

Boolean selection can also be used to remove rows from a `DataFrame`. The result of a Boolean selection will return a copy of rows where the expression is True. To drop rows, simply construct an expression that returns `False` for the rows that are to be dropped and then apply the expression to the data frame.

The following demonstrates dropping rows where `Price` is greater than 300. First, construct the expression.

```
In [32]: # determine the rows where Price > 300
selection = sp500.Price > 300
# report number of rows and number that will be dropped
(len(selection), selection.sum())

Out[32]: (500, 10)
```

From this result, we now know that there are 10 rows where the price is greater than 300. To get a data frame with those rows removed, select the complement of the selection.

Removing rows using a slice

Slicing can be used to remove records from a data frame. It is a process similar to Boolean selection where we select out all of the rows except for the ones you want removed.

Suppose we want to remove all but the first three records from `sp500`. The slice to perform this task is `[:3]` which returns the first three rows.

```
In [34]: # get only the first three rows
only_first_three = sp500[:3]
only_first_three
```



```
Out[34]:
```

Symbol	Sector	Price	BookValue
MMM	Industrials	141.14	26.668
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954

Remember, that since this is a slice, the result is a view into the original data frame. The rows have not been removed from the `sp500` data, and changes to these three rows will change the data in `sp500`. The proper action to prevent this is to make a copy of the slice which results in a new data frame with the data for the specified rows copied.

Summary

In this chapter, you learned how to do several common data manipulations using the pandas `DataFrame` object, specifically ones that change the structure of the `DataFrame` by adding or removing rows and columns. Additionally, we saw how to replace the data in specific rows and columns.

In the next chapter, we will examine in more detail the use of indexes to be able to efficiently retrieve data from within pandas objects.

Indexing Data

An index is a tool for optimized look up of values from a series or DataFrame. They are a lot like a key in a relational database, but more powerful. They provide the means of alignment for multiple sets of data and also carry semantics for how to handle various tasks with data such as resampling to different frequencies.

Much of the modeling that you will perform with pandas depends critically on how you set up your indexes. A properly implemented index will optimize performance and be an invaluable tool in driving your analysis.

We have previously used indexes briefly, and in this chapter, we will dive quite a bit deeper. During this deep dive, we will learn more about:

- The importance of indexes
- The types of pandas indexes, including `RangeIndex`, `Int64Index`, `CategoricalIndex`, `Float64Index`, `DatetimeIndex`, and `PeriodIndex`
- Setting and resetting an index
- Creating hierarchical indexes
- Selection of data using hierarchical indexes

Configuring pandas

We start with the standard configuration for pandas but we also load the S&P 500 data for use in several examples.

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set some pandas options controlling output format
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline

# read in the data and print the first five rows
# use the Symbol column as the index, and
# only read in columns in positions 0, 2, 3, 7
sp500 = pd.read_csv("data/sp500.csv",
                    index_col='Symbol',
                    usecols=[0, 2, 3, 7])
```


The importance of indexes

Pandas indexes allow efficient lookup of values. If indexes did not exist, a linear search across all of our data would be required. Indexes create optimized shortcuts to specific data items using a direct lookup instead of a search process.

To begin examining the value of indexes we will use the following `DataFrame` of 10000 random numbers:

```
In [2]: # create DataFrame of random numbers and a key column
np.random.seed(123456)
df = pd.DataFrame({'foo':np.random.random(10000), 'key':range(100,
df[:5]
```



```
Out[2]:      foo    key
0  0.126970  100
1  0.966718  101
2  0.260476  102
3  0.897237  103
4  0.376750  104
```

Suppose we want to look up the value of the random number where `key==10099` (I explicitly picked this value as it is the last row in the `DataFrame`). We can do this using a Boolean selection.

```
In [3]: # boolean select where key is 10099
df[df.key==10099]
```



```
Out[3]:      foo    key
9999  0.272283  10099
```

Conceptually, this is simple. But what if we want to do this repeatedly? This can be simulated in Python using the `%timeit` statement. The following code performs the lookup repeatedly and reports on the performance.

```
In [4]: # time the select
%timeit df[df.key==10099]
```



```
1000 loops, best of 3: 535 µs per loop
```

This result states that there are 1,000 executions performed three times, and the fastest of those three took lookup 0.00535 seconds per loop on average (a total of 5.35 seconds for that one set of 1,000 loops).

Now let's try this using an index to help us look up the values. The following code sets the index of this `DataFrame` to match the values of the `key` column.

```
In [5]: # move key to the index
df_with_index = df.set_index(['key'])
df_with_index[:5]
```



```
Out[5]:      foo
key
100  0.126970
101  0.966718
102  0.260476
103  0.897237
104  0.376750
```

And now it is possible to look up this value using `.loc[]`.

```
In [6]: # now can lookup with the index  
df_with_index.loc[10099]  
  
Out[6]: foo    0.272283  
Name: 10099, dtype: float64
```

That was just one lookup. Let's time it with %timeit.

```
In [7]: # and this is a lot faster  
%timeit df_with_index.loc[10099]  
10000 loops, best of 3: 112 µs per loop
```

The lookup using the index is roughly five times faster. Because of this greater performance, it is normally a best practice to perform lookup by index whenever possible. The downside of using an index is that it can take time to construct and also consumes more memory.

Many times, you will inherently know what your indexes should be and you can just create them upfront and get going with exploration. Other times, it will take some exploration first to determine the best index. And often it is possible that you do not have enough data or the proper fields to create a proper index. In these cases, you may need to use a partial index that returns multiple semi-ambiguous results and still perform Boolean selection on that set to get to the desired result.



It is a best practice when performing exploratory data analysis to first load the data and explore it using queries / Boolean selection. Then create an index if your data naturally supports one, or if you do require the increased speed.

The pandas index types

The pandas provides many built-in indexes. Each of the index types is designed for optimized lookups based upon a specific data type or pattern of data. Let's look at several of them that are commonly used.

The fundamental type - Index

This type of index is the most generic and represents an ordered and sliceable set of values. The values that it contains must be hashable Python objects. This is because the index will use this hash to form an efficient lookup of values associated with the value of that object. While hash lookup is preferred over linear lookup, there are other types of indexes that can be further optimized.

It is common that the columns index will be of this generic type. The following code demonstrates the use of this index type as the columns of the `DataFrame`.

```
In [8]: # show that the columns are actually an index
temp = pd.DataFrame({ "City": [ "Missoula", "Philadelphia" ],
                      "Temperature": [ 70, 80 ] })
temp
```

	City	Temperature
0	Missoula	70
1	Philadelphia	80

```
In [9]: # we can see columns is an index
temp.columns
```

```
Out[9]: Index(['City', 'Temperature'], dtype='object')
```

While this type of index generally works well for alphanumeric column names, it is possible to use other types of indexes as the column index if desired.

Integer index labels using Int64Index and RangeIndex

The `Int64Index` represents an immutable array of 64-bit integers that map to values. Until more recent versions of pandas, this was the default index type when an index was not specified or integers are used as seen in the following code snippet:

```
In [10]: # explicitly create an Int64Index
df_i64 = pd.DataFrame(np.arange(10, 20), index=np.arange(0, 10))
df_i64[:5]
```

```
Out[10]: 0
0 10
1 11
2 12
3 13
4 14
```

```
In [11]: # view the index
df_i64.index
```

```
Out[11]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')
```

Using this index, lookups of rows in the `DataFrame` are highly efficient as they are performed using a contiguous in-memory array.

Recent versions of pandas added the `RangeIndex` as an optimization of `Int64Index`. It has the ability to represent an integer-based index that starts at a specific integer value, has an end integer value, and can also have a step specified.

This use of start, stop, and step was such a common pattern that it warranted adding its own subclass to pandas. By using these three values, memory can be conserved and the execution time is in the same order as in an `Int64Index`.

`RangeIndex` has become the default index for pandas objects. This is demonstrated by the following, which creates a range of integer values defaulting to a `RangeIndex`.

```
In [12]: # by default we are given a RangeIndex
df_range = pd.DataFrame(np.arange(10, 15))
df_range[:5]
```

```
Out[12]: 0
0 10
1 11
2 12
3 13
4 14
```

```
In [13]: df_range.index
```

```
Out[13]: RangeIndex(start=0, stop=5, step=1)
```


Floating-point labels using Float64Index

Floating-point numbers can be used as index labels by using a `Float64Index`.

```
In [14]: # indexes using a Float64Index
df_f64 = pd.DataFrame(np.arange(0, 1000, 5),
                      np.arange(0.0, 100.0, 0.5))
df_f64.iloc[:5] # need iloc to slice first five

Out[14]:      0
0.0    0
0.5    5
1.0   10
1.5   15
2.0   20
```



Note that this slice returned 11 rows, instead of the first five. This is because of the `Float64Index`, and pandas taking our statement to mean from the beginning to all values up to 5.0.

Representing discrete intervals using IntervalIndex

Distinct buckets of labels can be represented using an `IntervalIndex`. The interval is closed at one end, either the left or right, meaning that the value at that end of the interval is included in that interval. The following code shows the creation of a `DataFrame` using intervals as an index.

```
In [15]: df_f64.index  
Out[15]: Float64Index([ 0.0,  0.5,  1.0,  1.5,  2.0,  2.5,  3.0,  
                      3.5,  4.0,  4.5,  
                      ...  
                     95.0, 95.5, 96.0, 96.5, 97.0, 97.5, 98.0,  
                     98.5, 99.0, 99.5],  
                     dtype='float64', length=200)
```

```
In [16]: # a DataFrame with an IntervalIndex  
df_interval = pd.DataFrame({ "A": [1, 2, 3, 4]},  
                           index = pd.IntervalIndex.from_breaks(  
                                [0, 0.5, 1.0, 1.5, 2.0]))  
df_interval  
  
Out[16]:          A  
(0.0, 0.5]  1  
(0.5, 1.0]  2  
(1.0, 1.5]  3  
(1.5, 2.0]  4
```


Categorical values as an index - CategoricalIndex

A `CategoricalIndex` is used to represent a sparsely populated index for an underlying `Categorical`. The following creates a `DataFrame` with one column being a `Categorical`.

```
In [17]: df_interval.index  
Out[17]: IntervalIndex([(0.0, 0.5], (0.5, 1.0], (1.0, 1.5], (1.5, 2.0])  
          closed='right',  
          dtype='interval[float64]')
```

Shifting categorical column (`B`) into the index of the `DataFrame` shows that there is now a `CategoricalIndex`.

```
In [18]: # create a DataFrame with a Categorical column  
df_categorical = pd.DataFrame({'A': np.arange(6),  
                               'B': list('aabbc'a)})  
df_categorical['B'] = df_categorical['B'].astype('category',  
                                              categories=list('cab'))  
df_categorical  
Out[18]:   A   B  
0   0   a  
1   1   a  
2   2   b  
3   3   b  
4   4   c  
5   5   a
```

And lookups can then be performed using existing values in the underlying categorical.

```
In [19]: # shift the categorical column to the index  
df_categorical = df_categorical.set_index('B')  
df_categorical.index  
Out[19]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=[  
                           'c', 'a', 'b'], ordered=False, name='B', dtype='category')
```

Categoricals will be examined in detail in Chapter 7, Categorical Data.



Indexing by date and time using DatetimeIndex

A `DatetimeIndex` is used to represent a set of dates and times. These are extensively used in time series data where samples are taken at specific intervals of time. To briefly demonstrate this, the following code creates a range of 5-hourly time periods and uses them as the index for this series.

```
In [20]: # lookup values in category 'a'  
df_categorical.loc['a']  
  
Out[20]: A  
B  
a 0  
a 1  
a 5
```

The type of the index can be seen to be a `DatetimeIndex`.

```
In [21]: # create a DatetimeIndex from a date range  
rng = pd.date_range('5/1/2017', periods=5, freq='H')  
ts = pd.Series(np.random.randn(len(rng)), index=rng)  
ts  
  
Out[21]: 2017-05-01 00:00:00    1.239792  
2017-05-01 01:00:00    -0.400611  
2017-05-01 02:00:00     0.718247  
2017-05-01 03:00:00     0.430499  
2017-05-01 04:00:00     1.155432  
Freq: H, dtype: float64
```

The underlying representation of the date/time is a 64-bit integer, making lookups by date and time very efficient.

Indexing periods of time using PeriodIndex

It is also important to be able to represent periods such as a day, month, or year. This is much like an interval but for a period of time. These scenarios can be modeled by using the `PeriodIndex` and specifying a specific frequency for the periods in the index.

The following demonstrates by modeling three 1-month periods starting from `2017-01`.

```
In [22]: ts.index
Out[22]: DatetimeIndex(['2017-05-01 00:00:00',
                       '2017-05-01 01:00:00',
                       '2017-05-01 02:00:00',
                       '2017-05-01 03:00:00',
                       '2017-05-01 04:00:00'],
                      dtype='datetime64[ns]', freq='H')
```

And this index can then be used within a `Series` OR `DataFrame`.

```
In [23]: # explicitly create a PeriodIndex
periods = pd.PeriodIndex(['2017-1', '2017-2', '2017-3'], freq='M')
periods
Out[23]: PeriodIndex(['2017-01', '2017-02', '2017-03'], dtype='period
[M]', freq='M')
```


Working with Indexes

Having covered creating various types of pandas indexes, let's now examine several common patterns of use for these indexes. Specifically, we will examine:

- Creating and using an index with a Series or DataFrame
- Means of selecting values with an index
- Moving data to and from the index
- Reindexing a pandas object

Creating and using an index with a Series or DataFrame

Indexes can either be created explicitly or you can let pandas create them implicitly. Explicit creation takes place when you assign the index using the `index` parameter of a constructor.

The following code demonstrates by first creating a `DatetimeIndex` independently.

```
In [24]: # use the index in a Series
period_series = pd.Series(np.random.randn(len(periods)),
                           index=periods)
period_series

Out[24]: 2017-01    -0.449276
2017-02     2.472977
2017-03    -0.716023
Freq: M, dtype: float64
```

You can use this index on its own or associate it with a `Series` OR `DataFrame`. This code creates a `DataFrame` utilizing the index.

```
In [25]: # create a DatetimeIndex
date_times = pd.DatetimeIndex(pd.date_range('5/1/2017',
                                             periods=5,
                                             freq='H'))
date_times

Out[25]: DatetimeIndex(['2017-05-01 00:00:00',
                       '2017-05-01 01:00:00',
                       '2017-05-01 02:00:00',
                       '2017-05-01 03:00:00',
                       '2017-05-01 04:00:00'],
                      dtype='datetime64[ns]', freq='H')
```

An index can also be directly assigned to an existing `DataFrame` or `Series` by setting the `.index` property.

```
In [26]: # create a DataFrame using the index
df_date_times = pd.DataFrame(np.arange(0, len(date_times)),
                             index=date_times)
df_date_times

Out[26]:          0
2017-05-01 00:00:00  0
2017-05-01 01:00:00  1
2017-05-01 02:00:00  2
2017-05-01 03:00:00  3
2017-05-01 04:00:00  4
```


Selecting values using an index

Values can be looked up using an index using the `[]` operator or by using the following property indexers of a `Series` OR `DataFrame`:

<code>.loc[]</code>	Looks up by the label, not the location. But be careful; if the labels are integers, then integers will be treated as labels!
<code>.at[]</code>	Like <code>.loc[]</code> , but this can only retrieve a single value.
<code>.iloc[]</code>	Lookup is based on the 0-based position and not the index label.
<code>.ix[]</code>	Hybrid, which when given an integer will attempt 0-based lookup; other types are label-based. This is going to be deprecated, so stick with the other three properties.

Values can be looked up in a `Series` by using the `[]` operator as in the following `DataFrame`, which has its `b` value retrieved.

```
In [27]: # set the index of a DataFrame
df_date_times.index = pd.DatetimeIndex(pd.date_range('6/1/2017',
                                                       periods=5,
                                                       freq='H'))
df_date_times
Out[27]: 0
2017-06-01 00:00:00 0
2017-06-01 01:00:00 1
2017-06-01 02:00:00 2
2017-06-01 03:00:00 3
2017-06-01 04:00:00 4
```

```
In [28]: # create a series
s = pd.Series(np.arange(0, 5), index=list('abcde'))
s
Out[28]: a    0
         b    1
         c    2
         d    3
         e    4
dtype: int64
```

Lookup using `[]` on a `Series` is equivalent to using the `.loc[]` property.

```
In [29]: # lookup by index label
s['b']
Out[29]: 1
```

The `[]` operator retrieves columns instead of the rows when applied to `DataFrame`.

```
In [30]: # explicit lookup by label  
s.loc['b']
```

```
Out[30]: 1
```

```
In [31]: # create a DataFrame with two columns  
df = pd.DataFrame([ np.arange(10, 12),  
                     np.arange(12, 14)],  
                   columns=list('ab'),  
                   index=list('vw'))  
df
```

```
Out[31]:   a   b  
          v  10  11  
          w  12  13
```

To look up via the row index with a DataFrame, one of the property indexers must be utilized.

```
In [32]: # this returns the column 'a'  
df['a']
```

```
Out[32]: v    10  
         w    12  
Name: a, dtype: int64
```

The property indexer forms are also able to use slicing.

```
In [33]: # return the row 'w' by label  
df.loc['w']
```

```
Out[33]: a    12  
         b    13  
Name: w, dtype: int64
```

```
In [34]: # slices the Series from index label b to d  
s['b':'d']
```

```
Out[34]: b    1  
         c    2  
         d    3  
dtype: int64
```

And it is also possible to pass a list of values.

```
In [35]: # this explicitly slices from label b to d  
s.loc['b':'d']
```

```
Out[35]: b    1  
         c    2  
         d    3  
dtype: int64
```


Moving data to and from the index

A `DataFrame` object can have its index reset by using the `.reset_index()`. A common use of this is for moving the contents of a `DataFrame` object's index into one or more columns.

The following code moves the symbols in the index of `sp500` into a column and replaces the index with a default integer index.

```
In [36]: # and this looks up rows by label
s.loc[['a', 'c', 'e']]
```



```
Out[36]: a    0
          c    2
          e    4
         dtype: int64
```

```
In [37]: # examine some of the sp500 data
sp500[:5]
```


	Symbol	Sector	Price	Book Value
0	MMM	Industrials	141.14	26.668
1	ABT	Health Care	39.60	15.573
2	ABBV	Health Care	53.95	2.954
3	ACN	Information Technology	79.79	8.326
4	ACE	Financials	102.91	86.897

A column of data can be moved to the index of a `DataFrame` object using the `.set_index()` method and by specifying the column to be moved. The following code moves the `Sector` column to the index.

```
In [38]: # reset the index which moves the values in the index to a column
index_moved_to_col = sp500.reset_index()
index_moved_to_col[:5]
```


	Symbol	Sector	Price	Book Value
0	MMM	Industrials	141.14	26.668
1	ABT	Health Care	39.60	15.573
2	ABBV	Health Care	53.95	2.954
3	ACN	Information Technology	79.79	8.326
4	ACE	Financials	102.91	86.897

It is possible to move multiple columns to the index, forming a hierarchical/multi-index. Hierarchical indexes will be covered later in the chapter.

Reindexing a pandas object

A `DataFrame` can be reindexed using the `.reindex()` method. Reindexing makes the `DataFrame` conform to the new index, aligning data from the old index with the new and filling in `NaN` where alignment fails.

This code demonstrates re-indexing the `sp500` to the three specified index labels.

```
In [39]: # and now set the Sector column to be the index
index_moved_to_col.set_index('Sector')[:5]
```

Sector	Symbol	Price	Book Value
Industrials	MMM	141.14	26.668
Health Care	ABT	39.60	15.573
Health Care	ABBV	53.95	2.954
Information Technology	ACN	79.79	8.326
Financials	ACE	102.91	86.897

This process creates a new `DataFrame` with specified rows. If a row is not found for a specific value, then `NaN` values are inserted as is seen with the '`FOO`' label. This method is actually a good technique of filtering out data based on the index labels.

Reindexing can also be done upon columns, as demonstrated by the following code for the specified three column names:

```
In [40]: # reindex to have MMM, ABBV, and FOO index labels
reindexed = sp500.reindex(index=['MMM', 'ABBV', 'FOO'])
# note that ABT and ACN are dropped and FOO has NaN values
reindexed
```

Symbol	Sector	Price	Book Value
MMM	Industrials	141.14	26.668
ABBV	Health Care	53.95	2.954
FOO	NaN	NaN	NaN

`NaN` values are inserted for `NewCol` as it was not present in the original data.

Hierarchical indexing

Hierarchical indexing is a feature of pandas that allows the combined use of two or more indexes per row. Each of the indexes in a hierarchical index is referred to as a level. The specification of multiple levels in an index allows for efficient selection of different subsets of data using different combinations of the values at each level. Technically, a pandas index that has multiple levels of hierarchy is referred to as a `MultiIndex`.

The following code demonstrates creating and accessing data via a `MultiIndex` using the `sp500` data. Suppose we want to organize this data by both the values of `Sector` and `Symbol` so that we can efficiently look up data based on a combination of values from both variables. We can accomplish this with the following code, which moves the `Sector` and `Symbol` values into a `MultiIndex`:

```
In [41]: # reindex columns
sp500.reindex(columns=['Price',
                      'Book Value',
                      'NewCol'])[:5]

Out[41]:      Price  Book Value  NewCol
Symbol
MMM      141.14     26.668    NaN
ABT      39.60      15.573    NaN
ABBV     53.95      2.954     NaN
ACN      79.79      8.326     NaN
ACE      102.91     86.897    NaN
```

The `.index` property now shows that the index is a `MultiIndex` object:

```
In [42]: # first, push symbol into a column
reindexed = sp500.reset_index()
# and now index sp500 by sector and symbol
multi_fi = reindexed.set_index(['Sector', 'Symbol'])
multi_fi[:5]

Out[42]:           Price  Book Value
Sector      Symbol
Industrials MMM      141.14     26.668
Health Care ABT      39.60      15.573
               ABBV     53.95      2.954
Information Technology ACN      79.79      8.326
Financials   ACE      102.91     86.897
```

As stated, a `MultiIndex` object contains two or more levels, two in this case:

```
In [43]: # the index is a MultiIndex
type(multi_fi.index)

Out[43]: pandas.core.indexes.multi.MultiIndex
```

Each level is a distinct `Index` object:

```
In [44]: # this has two levels
len(multi_fi.index.levels)

Out[44]: 2
```

```
In [45]: # each index level is an index
multi_fi.index.levels[0]
```

```
Out[45]: Index(['Consumer Discretionary', 'Consumer Discretionary',
   'Consumer Staples', 'Consumer Staples ', 'Energy',
   'Financials', 'Health Care', 'Industrials',
   'Industries', 'Information Technology', 'Materials',
   'Telecommunications Services', 'Utilities'],
  dtype='object', name='Sector')
```

The values of the index itself, at a specific level for every row, can be retrieved by the `.get_level_values()` method:

```
In [46]: # each index level is an index
multi_fi.index.levels[1]
```

```
Out[46]: Index(['A', 'AA', 'AAPL', 'ABBV', 'ABC', 'ABT', 'ACE',
   'ACN', 'ACT', 'ADBE',
   ...
   'XLNX', 'XOM', 'XRAY', 'XRX', 'XYL', 'YHOO', 'YUM',
   'ZION', 'ZMH', 'ZTS'],
  dtype='object', name='Symbol', length=500)
```

Access to values in the `DataFrame` object via a hierarchical index is performed using the `.xs()` method. This method works similar to the `.ix` attribute but provides parameters to specify the multidimensionality of the index.

The following code selects all items with a level 0 (`Sector`) index label of `Industrials`:

```
In [47]: # values of index level 0
multi_fi.index.get_level_values(0)
```

```
Out[47]: Index(['Industrials', 'Health Care', 'Health Care',
   'Information Technology', 'Financials',
   'Health Care', 'Information Technology',
   'Utilities', 'Health Care', 'Financials',
   ...
   'Utilities', 'Information Technology',
   'Information Technology', 'Financials',
   'Industrials', 'Information Technology',
   'Consumer Discretionary', 'Health Care',
   'Financials', 'Health Care'],
  dtype='object', name='Sector', length=500)
```

The index of the resulting `DataFrame` consists of the levels that were not specified, in this case, `Symbol`. The levels for which values were specified are dropped from the resulting index.

The `level` parameter can be used to select the rows with a specific value of the index at the specified level. The following code selects rows where the `symbol` component of the index is `ALLE`.

```
In [48]: # get all stocks that are Industrials
# note the result drops level 0 of the index
multi_fi.xs('Industrials')[:5]
```

```
Out[48]:      Price  Book Value
Symbol
MMM      141.14    26.668
ALLE     52.46     0.000
APH      95.71    18.315
AVY      48.20    15.616
BA       132.41   19.870
```

To prevent levels from being dropped (if you don't specify each level), you can use the `drop_levels=False`

option.

```
In [49]: # select rows where level 1 (Symbol) is ALLE
# note that the Sector level is dropped from the result
multi_fi.xs('ALLE', level=1)

Out[49]:      Price  Book Value
Sector
Industrials  52.46      0.0
```

To select from a hierarchy of indexes, you can chain .xs() calls with different levels together. The following selects the row with `Industrials` at level 0 and `UPS` at level 1.

```
In [50]: # Industrials, without dropping the level
multi_fi.xs('Industrials', drop_level=False)[:5]

Out[50]:      Price  Book Value
Sector      Symbol
Industrials MMM    141.14    26.668
              ALLE   52.46     0.000
              APH    95.71    18.315
              AVY    48.20    15.616
              BA     132.41   19.870
```

An alternate syntax is to pass the values of each level of the hierarchical index as a tuple.

```
In [51]: # drill through the levels
multi_fi.xs('Industrials').xs('UPS')

Out[51]: Price      102.73
Book Value  6.79
Name: UPS, dtype: float64
```

Note that .xs() can only be used for getting and not for setting values.

Summary

In this chapter, we took a deeper dive into using indexes in pandas to organize and retrieve data. We examined many of the useful types of indexes and how they can be used with different types of data to efficiently access values without needing to query data in the rows. And we concluded with an examination of using hierarchical indexes to be able to efficiently retrieve data that match to labels in multiple indexes, giving us a powerful means of being able to select subsets of data.

At this point, we have covered much of the basic modeling portion of pandas. In the next chapter, we will examine to represent categorical variable with pandas.

Categorical Data

A categorical variable is a type of variable in statistics that represents a limited and often fixed set of values. This is in contrast to continuous variables, which can represent an infinite number of values. Common types of categorical variables include gender (where there are two values, male and female) or blood types (which can be one of the small sets of types of blood, such as A, B, and O).

pandas has the ability to represent Categorical variables using a type of pandas object known as `categorical`. These pandas objects are designed to efficiently represent data that is grouped into a set of buckets, each represented by an integer code that represents one of the categories. The use of these underlying codes gives pandas the ability to efficiently represent sets of categories and to perform ordering and comparisons of data across multiple categorical variables.

During this chapter, we will learn the following about Categoricals:

- Creating Categoricals
- Renaming categories
- Appending new categories
- Removing categories
- Removing unused categories
- Setting categories
- Descriptive statistics
- Value counts
- Minimum, maximum, and mode
- How to use a Categorical to assign letter grades to students based on their numeric grade

Configuring pandas

We start the examples in this chapter using the following imports and configuration statements:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set some pandas options controlling output format
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 80)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```


Creating Categoricals

A pandas Categorical is used to represent a categorical variable. A categorical variable consists of a finite set of values and is often used to map values into a set of categories and track how many values are present in each category. Another purpose is to map sections of continuous values into a discrete set of named labels, an example of which is mapping a numeric grade to a letter grade. We will examine how to perform this mapping at the end of the chapter.

There are several ways to create a pandas Categorical. The following screenshot demonstrates creating a Categorical directly from a list:

```
In [2]: # create a categorical directly from a list.  
lmh_values = ["low", "high", "medium", "medium", "high"]  
lmh_cat = pd.Categorical(lmh_values)  
lmh_cat  
  
Out[2]: [low, high, medium, medium, high]  
Categories (3, object): [high, low, medium]
```

This Categorical is created from a list consisting of five strings and three distinct values: low, medium, and high. When creating the Categorical, pandas determines each unique value in the list and uses those as the categories.

These categories can be examined using the `.categories` property:

```
In [3]: # examine the categories  
lmh_cat.categories  
  
Out[3]: Index(['high', 'low', 'medium'], dtype='object')
```

The Categorical has created an index consisting of the three distinct values in the provided list.

The values of the Categorical can be retrieved using `.get_values()`:

```
In [4]: # retrieve the values  
lmh_cat.get_values()  
  
Out[4]: array(['low', 'high', 'medium', 'medium', 'high'], dtype=object)
```

Each category of the Categorical is assigned an integer value. This value is referred to as a **code**. These assigned codes can be accessed with the `.codes` property.

```
In [5]: # .codes shows the integer mapping for each value of the categorical  
lmh_cat.codes  
  
Out[5]: array([1, 0, 2, 2, 0], dtype=int8)
```

These codes effectively represent the position in the categoricals index for each value. In this case, the mapping is *low* = 1, *high* = 0, and *medium* = 2. This ordering perhaps does not make logical sense and was determined by pandas by serially processing strings in the `lmh_values` array.

This ordering can be better controlled by specifying the categories using the `categories` parameter.

```
In [6]: # create from list but explicitly state the categories
lmh_cat = pd.Categorical(lmh_values,
                         categories=["low", "medium", "high"])
lmh_cat

Out[6]: [low, high, medium, medium, high]
Categories (3, object): [low, medium, high]
```

Note the new order of the categories, as specified in the constructor. Because of this ordering, the codes are now:

```
In [7]: # the codes are...
lmh_cat.codes

Out[7]: array([0, 2, 1, 1, 2], dtype=int8)
```

This now represents a mapping of low = 0, medium = 1, and high = 2. This is more useful, as it can be used to sort values in an order that matches the meaning of each category and their relation to the other categories. When sorting a Categorical, the sorting will be performed using the codes and not the actual value. The following demonstrates sorting of `lmh_cat`:

```
In [8]: # sorting is done using the codes underlying each value
lmh_cat.sort_values()

Out[8]: [low, medium, medium, high, high]
Categories (3, object): [low, medium, high]
```

A categorical variable can also be represented as a series whose `dtype` is specified as "category". The following screenshot demonstrates taking an array of categories and creating a series with a `dtype` category:

```
In [9]: # create a categorical using a Series and dtype
cat_series = pd.Series(lmh_values, dtype="category")
cat_series

Out[9]: 0      low
1      high
2    medium
3    medium
4      high
dtype: category
Categories (3, object): [high, low, medium]
```

Another way of creating a series with a Categorical is to first create the series and then convert one column into a Categorical using the `.astype('category')` method. The following screenshot demonstrates this by creating a data frame with the values in one column, and then a second column with the values converted to a Categorical:

```
In [10]: # create a categorical using .astype()
s = pd.Series(lmh_values)
as_cat = s.astype('category')
cat_series
```

```
Out[10]: 0      low
1     high
2   medium
3   medium
4     high
dtype: category
Categories (3, object): [high, low, medium]
```

A series that is created as a Categorical contains a `.cat` property that allows access to the Categorical used by the Series:

```
In [11]: # a categorical has a .cat property that lets you access various info
cat_series.cat
```

```
Out[11]: <pandas.core.categorical.CategoricalAccessor object at 0x10acale48>
```

The variable is a `CategoricalAccessor` object that allows access to various properties of the underlying Categorical. As an example, the following code gets the categories:

```
In [12]: # get the index for the categorical
cat_series.cat.categories
```

```
Out[12]: Index(['high', 'low', 'medium'], dtype='object')
```

Several pandas functions also return Categoricals. One is `pd.cut()`, which creates bins of objects within specific ranges of values. The following screenshot demonstrates cutting a series of values that represent 5 random numbers between 0 and 100 into 10 Categorical bins, each 10 integers wide:

```
In [13]: # create a DataFrame of 100 values
np.random.seed(123456)
values = np.random.randint(0, 100, 5)
bins = pd.DataFrame({ "Values": values})
bins
```

```
Out[13]:    Values
0      65
1      49
2      56
3      43
4      43
```

```
In [14]: # cut the values into
bins['Group'] = pd.cut(values, range(0, 101, 10))
bins
```

```
Out[14]:    Values      Group
0      65  (60, 70]
1      49  (40, 50]
2      56  (50, 60]
3      43  (40, 50]
4      43  (40, 50]
```

The `Group` column represents a categorical variable, as created by the `cut` function, and the value at each index level represents the specific category the value has been associated with.

```
In [15]: # examine the categorical that was created  
bins.Group
```

```
Out[15]: 0    (60, 70]  
1    (40, 50]  
2    (50, 60]  
3    (40, 50]  
4    (40, 50]  
Name: Group, dtype: category  
Categories (10, interval[int64]): [(0, 10] < (10, 20] < (20, 30] < (30, 40] ... (60, 70] < (70, 80] < (80, 90] < (90, 100]]
```

An explicit ordering of the categories can be specified using `ordered = True`. This information means that the ordering of the categories is significant and allows for the comparison of values in multiple series of categorical variables. To demonstrate this, the following code creates an ordered Categorical representing three metals (and four values from those three categories):

```
In [16]: # create an ordered categorical of precious metals  
# order is important for determining relative value  
metal_values = ["bronze", "gold", "silver", "bronze"]  
metal_categories = ["bronze", "silver", "gold"]  
metals = pd.Categorical(metal_values,  
                        categories=metal_categories,  
                        ordered = True)  
metals
```

```
Out[16]: [bronze, gold, silver, bronze]  
Categories (3, object): [bronze < silver < gold]
```

The Categorical that is created has a strict ordering of `bronze` as less valuable than `silver` and `silver` as less valuable than `gold`.

This ordering can be used to sort or compare one Categorical to another. To demonstrate this, let's create the following Categorical, which reverses the values:

```
In [17]: # reverse the metals  
metals_reversed_values = pd.Categorical(metals.get_values()[:-1],  
                                         categories = metals.categories,  
                                         ordered=True)  
metals_reversed_values
```

```
Out[17]: [bronze, silver, gold, bronze]  
Categories (3, object): [bronze < silver < gold]
```

These two categorical variables can be compared with each other. The following code shows whether each metal is less valuable than the corresponding metal at the same index label in the other Categorical:

```
In [18]: # compare the two categoricals  
metals <= metals_reversed_values
```

```
Out[18]: array([ True, False,  True,  True], dtype=bool)
```

This comparison is performed by pandas by matching the underlying code for each value. The `metals` variable has these codes:

```
In [19]: # codes are the integer value associated with each item  
metals.codes
```

```
Out[19]: array([0, 2, 1, 0], dtype=int8)
```

And `metals_reversed_values` has the following codes.

```
In [20]: # and for metals2  
metals_reversed_values.codes
```

```
Out[20]: array([0, 1, 2, 0], dtype=int8)
```

Application of the logical operator results in the previously shown result.

As a final example of creating a Categorical, the following screenshot demonstrates creating a Categorical that specifies a value (`copper`) that is not one of the specified categories. In this case, pandas will substitute `NaN` for that value.

```
In [21]: # creating a categorical with a value that is not in the cat gives a NaN (copper in this case)  
pd.Categorical(["bronze", "copper"],  
              categories=metal_categories)
```

```
Out[21]: [bronze, NaN]  
Categories (3, object): [bronze, silver, gold]
```

This technique can be used to filter out values that don't fit within the Categorical at the time of creation.

Renaming categories

A Categorical can have values renamed by assigning new values to the `.categories` property or using the `.rename_categories()` method.

```
In [22]: # create a categorical with 3 categories
cat = pd.Categorical(["a","b","c","a"],
                     categories=["a", "b", "c"])
cat

Out[22]: [a, b, c, a]
Categories (3, object): [a, b, c]
```

```
In [23]: # renames the categories (and also the values)
cat.categories = ["bronze", "silver", "gold"]
cat

Out[23]: [bronze, silver, gold, bronze]
Categories (3, object): [bronze, silver, gold]
```

Note that this was an in-place modification. To prevent an in-place modification, we can use the `.rename_categories()` method.

```
In [24]: # this also renames
cat.rename_categories(["x", "y", "z"])

Out[24]: [x, y, z, x]
Categories (3, object): [x, y, z]
```

We can verify that this was not done in-place.

```
In [25]: # the rename is not done in-place
cat

Out[25]: [bronze, silver, gold, bronze]
Categories (3, object): [bronze, silver, gold]
```


Appending new categories

Categories can be appended by using the `.add_categories()` method. The following code adds a `copper` category to our metals. This is not done in-place unless pandas is explicitly told to do so:

```
In [26]: # add a new platinum category
with_platinum = metals.add_categories(["platinum"])
with_platinum

Out[26]: [bronze, gold, silver, bronze]
Categories (4, object): [bronze < silver < gold < platinum]
```


Removing categories

Categories can be removed by using the `.remove_categories()` method. Values that are removed are replaced by `np.NaN`. The following screenshot demonstrates this by removing the `bronze` category:

```
In [27]: # remove bronze category
no_bronze = metals.remove_categories(["bronze"])
no_bronze

Out[27]: [NaN, gold, silver, NaN]
Categories (2, object): [silver < gold]
```


Removing unused categories

Unused categories can be removed by using `.remove_unused_categories()`, as follows:

```
In [28]: # remove any unused categories (in this case, platinum)
with_platinum.remove_unused_categories()
```

```
Out[28]: [bronze, gold, silver, bronze]
Categories (3, object): [bronze < silver < gold]
```


Setting categories

It is also possible to add and remove categories in one step using the `.set_categories()` method. Given the following series:

```
In [29]: # sample Series
s = pd.Series(["one", "two", "four", "five"], dtype="category")
s

Out[29]: 0    one
1    two
2    four
3    five
dtype: category
Categories (4, object): [five, four, one, two]
```

The following code sets the categories to "one" and "four":

```
In [30]: # remove the "two" and "-" category - leaves NaNs
s = s.cat.set_categories(["one", "four"])
s

Out[30]: 0    one
1    NaN
2    four
3    NaN
dtype: category
Categories (2, object): [one, four]
```

The result has `NaN` replacing for the categories that now do not exist.

Descriptive information of a Categorical

The `.describe()` method on a Categorical will produce descriptive statistics in a manner similar to `Series` or `DataFrame`.

```
In [31]: # get descriptive info on the metals categorical  
metals.describe()
```

```
Out[31]:      counts   freqs  
categories  
bronze        2    0.50  
silver        1    0.25  
gold          1    0.25
```

The result gives us the number of instances of each category and the frequency distribution of each.

The count of values for each category can be obtained using `.value_counts()`.

```
In [32]: # count the values in the categorical  
metals.value_counts()
```

```
Out[32]: bronze    2  
silver     1  
gold      1  
dtype: int64
```

And the min, max, and mode values are found using the respective methods.

```
In [33]: # find the min, max and mode of the metals categorical  
(metals.min(), metals.max(), metals.mode())
```

```
Out[33]: ('bronze', 'gold', [bronze]  
Categories (3, object): [bronze < silver < gold])
```


Munging school grades

Now let's look at applying Categoricals to help us organize information based on categories instead of numbers. The problem we will examine is assigning a letter grade to a student based on their numeric grade.

```
In [34]: # 10 students with random grades
np.random.seed(123456)
names = ['Ivana', 'Norris', 'Ruth', 'Lane', 'Skye', 'Sol', 'Dylan', 'Katina', 'Alissa', "Marc"]
grades = np.random.randint(50, 101, len(names))
scores = pd.DataFrame({'Name': names, 'Grade': grades})
scores
```

```
Out[34]:   Grade      Name
0      51    Ivana
1      92   Norris
2     100    Ruth
3      99    Lane
4      93   Skye
5      97     Sol
6      93   Dylan
7      77   Katina
8      82   Alissa
9      73     Marc
```

This data frame represents the raw score for each of the students. Next, we break down numeric grades into letter codes. The following code defines the bins for each grade and the associated letter grade for each bin:

```
In [35]: # bins and their mappings to letter grades
score_bins = [0, 59, 62, 66, 69, 72, 76, 79, 82, 86, 89, 92, 99, 100]
letter_grades = ['F', 'D-', 'D', 'D+', 'C-', 'C', 'C+', 'B-', 'B', 'B+', 'A-', 'A', 'A+']
```

Using these values, we can perform a cut that assigns the letter grade.

```
In [36]: # cut based upon the bins and assign the letter grade
letter_cats = pd.cut(scores.Grade, score_bins, labels=letter_grades)
scores['Letter'] = letter_cats
scores
```

```
Out[36]:   Grade      Name Letter
0      51    Ivana      F
1      92   Norris     A-
2     100    Ruth     A+
3      99    Lane      A
4      93   Skye      A
5      97     Sol      A
6      93   Dylan      A
7      77   Katina     C+
8      82   Alissa     B-
9      73     Marc      C
```

Examining the underlying Categorical shows how the following code was created and how the letter grades are related in value:

```
In [37]: # examine the underlying categorical  
letter_cats
```

```
Out[37]: 0      F  
1      A-  
2      A+  
3      A  
4      A  
5      A  
6      A  
7      C+  
8      B-  
9      C  
Name: Grade, dtype: category  
Categories (13, object): [F < D- < D < D+ ... B+ < A- < A < A+]
```

To determine how many students received each grade, we can use `.cat.value_counts()`:

```
In [38]: # how many of each grade occurred?  
scores.Letter.value_counts()
```

```
Out[38]: A      4  
A+     1  
A-     1  
B-     1  
C+     1  
..  
B      0  
C-     0  
D+     0  
D      0  
D-     0  
Name: Letter, Length: 13, dtype: int64
```

Since the letter grade Categorical has a logical ordering for the letter grades, we can use it to order the students from highest to lowest letter grade by sorting on the Categorical column.

```
In [39]: # and sort by letter grade instead of numeric grade  
scores.sort_values(by=['Letter'], ascending=False)
```

```
Out[39]:   Grade    Name Letter  
2      100     Ruth    A+  
6       93    Dylan     A  
5       97      Sol     A  
4       93    Skye     A  
3       99    Lane     A  
1       92  Norris    A-  
8       82  Alissa    B-  
7       77  Katina    C+  
9       73    Marc     C  
0       51  Ivana     F
```


Summary

In this chapter, we examined how to model categorical variables using pandas Categoricals. We started with a review of the ways to create Categoricals and also looked at several examples of how to utilize underlying integer codes to classify each category. We then examined several means of modifying a categorical once it has been created. The chapter closed with an example of using a categorical to break data into a set of named bins.

In the next chapter, we examine performing numerical and statistical analysis on pandas data.

Numerical and Statistical Methods

pandas is extremely powerful for modeling and manipulating data, but it also provides many powerful tools for numerical and statistical analysis. These capabilities are tightly integrated with pandas data structures and thereby make complex calculations very simple once the data is modeled.

This chapter will examine many of these capabilities. It starts with common numerical methods such as arithmetic with alignment across multiple objects, as well as finding specific values such as minimums and maximums. We then will look at many of the statistical capabilities of pandas, such as working with quantiles, ranking of values, variance, correlation, and many others.

Last but not least, we will examine a very powerful capability provided in pandas known as the rolling window. Rolling windows provide a means of applying various methods such mean calculation across regular subsets of data. These types of operations are essential in various analyses such as determining several characteristics of stock data as it changes across time. The concept will be introduced in this chapter, and we will cover it in much more detail in later chapters.

There's a lot to cover in this chapter, including:

- Performing arithmetical operations on pandas objects
- Getting the counts of values
- Determining unique values (and their counts)
- Finding the minimum and maximum values
- Locating the n-smallest and n-largest values
- Calculating accumulated values
- Retrieving summary descriptive statistics
- Measuring central tendency (the mean, median, and mode)
- Calculating variance, standard deviation, covariance, and correlation
- Performing discretization and quantiling of data
- Calculating the rank of values
- Calculating the percentage change at each sample in a series
- Performing moving window operations
- Executing random sampling of data

Configuring pandas

We will start with the following configuration of the environment, using our standard import and configuration of pandas. In addition to the normal imports and formatting, we will also import the S&P 500 data and the monthly stock data:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set formattign options
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline

# read in the data and print the first five rows
# use the Symbol column as the index, and
# only read in columns in positions 0, 2, 3, 7
sp500 = pd.read_csv("data/sp500.csv",
                    index_col='Symbol',
                    usecols=[0, 2, 3, 7])

# one month of stock history data
omh = pd.read_csv("data/omh.csv")
```


Performing numerical methods on pandas objects

pandas provides a rich set of functions and operations that facilitate the ability to perform arithmetic operations and to calculate various numerical characteristics of data. In this section, we will examine many of these, including:

- Performing arithmetic on a DataFrame or Series
- Getting the counts of values
- Determining unique values (and their counts)
- Finding minimum and maximum values
- Locating the n-smallest and n-largest values
- Calculating accumulated values

Performing arithmetic on a DataFrame or Series

Arithmetic operations can be performed on a DataFrame (and Series) using the `+`, `-`, `/`, and `*` operators. While these may seem trivial in nature, pandas adds a powerful twist by performing alignment of the values on the left and right sides of the equations. Hence indexes play a big part in arithmetic, and how this affects the results needs to be understood by the user of pandas.

Additionally, pandas not only provides the standard operators for arithmetic but also provides several methods, `.add()`, `.sub()`, `.mul()`, and `.div()`, that provide higher performance and greater flexibility in specifying the axes on which to apply the function.

Arithmetic operations using a scalar value will be applied to every element of a DataFrame. To demonstrate, we begin with the following DataFrame of four columns of random numbers:

```
In [2]: # set the seed to allow replicatable results
np.random.seed(123456)
# create the DataFrame
df = pd.DataFrame(np.random.randn(5, 4),
                  columns=['A', 'B', 'C', 'D'])
df

Out[2]:          A         B         C         D
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
```

By default, any arithmetic operation will be applied across all rows and columns of a DataFrame. This will result in a new DataFrame object containing the results (leaving the original unchanged):

```
In [3]: # multiply everything by 2
df * 2

Out[3]:          A         B         C         D
0  0.938225 -0.565727 -3.018117 -2.271265
1  2.424224 -0.346429  0.238417 -2.088472
2 -1.723698 -4.209138 -0.989859  2.143608
3  1.443110 -1.413542 -2.079150  0.543720
4 -0.849945  1.134041  0.552464 -2.174801
```

When performing an operation between a DataFrame and a Series, pandas will align the Series index along the DataFrame columns, performing what is referred to as a row-wise broadcast. This is perhaps a little bit counter-intuitive, but it is quite powerful in applying different values in each column on a row-by-row basis.

To demonstrate, the following example retrieves the first row of the DataFrame and then subtracts this from each row, essentially resulting in the difference in value of each row from the first:

```
In [4]: # get first row
s = df.iloc[0]
# subtract first row from every row of the DataFrame
diff = df - s
diff

Out[4]:          A         B         C         D
0  0.000000  0.000000  0.000000  0.000000
1  0.743000  0.109649  1.628267  0.091396
2 -1.330961 -1.821706  1.014129  2.207436
3  0.252443 -0.423908  0.469484  1.407492
4 -0.894085  0.849884  1.785291  0.048232
```

This process also works when reversing the order by subtracting the `DataFrame` from the `Series` object as pandas is smart enough to figure out the proper application:

```
In [5]: # subtract DataFrame from Series
diff2 = s - df
diff2

Out[5]:      A          B          C          D
0  0.000000  0.000000  0.000000  0.000000
1 -0.743000 -0.109649 -1.628267 -0.091396
2  1.330961  1.821706 -1.014129 -2.207436
3 -0.252443  0.423908 -0.469484 -1.407492
4  0.894085 -0.849884 -1.785291 -0.048232
```

The set of columns resulting from an arithmetic operation will be the union of the labels in the index of both the series and the columns index of the `DataFrame` object (as per alignment rules). If a label representing the result column is not found in either the `Series` or the `DataFrame` object, then the values will be filled with `NaN`. The following code demonstrates this by creating a `Series` with an index that represents a subset of the columns in the `DataFrame`, but with an additional label:

```
In [6]: # B, C
s2 = s[1:3]
# add E
s2['E'] = 0
# see how alignment is applied in math
df + s2

Out[6]:      A          B          C          D          E
0  NaN -0.565727 -3.018117  NaN  NaN
1  NaN -0.456078 -1.389850  NaN  NaN
2  NaN -2.387433 -2.003988  NaN  NaN
3  NaN -0.989634 -2.548633  NaN  NaN
4  NaN  0.284157 -1.232826  NaN  NaN
```

pandas aligns the index labels of `df` with those of `s2`. Since `s2` does not have an `A` or `D` label in the columns, the result contains `NaN` in those columns. And since `df` does not have an `E` label, it is also filled with `NaN` (even though `E` existed in the `Series`).

An arithmetic operation between two `DataFrame` objects will align by both the column and index labels. The following code extracts a small portion of `df` and subtracts it from the full `DataFrame`. The result demonstrates that the aligned values subtract to 0 while the others are set to `NaN`:

```
In [7]: # get rows 1 through three, and only B, C columns
subframe = df[1:4][['B', 'C']]
# we have extracted a little square in the middle of df
subframe

Out[7]:      B          C
1 -0.173215  0.119209
2 -2.104569 -0.494929
3 -0.706771 -1.039575
```

```
In [8]: # demonstrate the alignment of the subtraction
df - subframe

Out[8]:      A          B          C          D
0  NaN  NaN  NaN  NaN
1  NaN  0.0  0.0  NaN
2  NaN  0.0  0.0  NaN
3  NaN  0.0  0.0  NaN
4  NaN  NaN  NaN  NaN
```

Additional control of an arithmetic operation can be gained using the arithmetic methods provided by the

DataFrame object. These methods provide the specification of a specific axis. The following code demonstrates subtraction of the A column values from those in every column:

```
In [9]: # get the A column
a_col = df['A']
df.sub(a_col, axis=0)

Out[9]:      A          B          C          D
0  0.0 -0.751976 -1.978171 -1.604745
1  0.0 -1.385327 -1.092903 -2.256348
2  0.0 -1.242720  0.366920  1.933653
3  0.0 -1.428326 -1.761130 -0.449695
4  0.0  0.991993  0.701204 -0.662428
```


Getting the counts of values

The `.count()` method gives us the count of the non-`NaN` items in a `Series`.

```
In [10]: s = pd.Series(['a', 'a', 'b', 'c', np.NaN])
# number of occurrences of each unique value
s.count()

Out[10]: 4
```


Determining unique values (and their counts)

A list of unique values in a Series can be obtained using `.unique()`:

```
In [11]: # return a list of unique items  
s.unique()
```

```
Out[11]: array(['a', 'b', 'c', nan], dtype=object)
```

The number of unique values (excluding `NaN`) can be obtained using `.nunique()`:

```
In [12]: s.nunique()
```

```
Out[12]: 3
```

To include `NaN` in the result, use `dropna=False` as a parameter.

```
In [13]: s.nunique(dropna=False)
```

```
Out[13]: 4
```

The count of each unique value can be determined using `.value_counts()` (a process also known as

```
In [14]: # get summary stats on non-numeric data  
s.value_counts(dropna=False)
```

```
Out[14]: a      2  
         c      1  
         b      1  
         NaN    1  
         dtype: int64
```

Histogramming):

Finding minimum and maximum values

The minimum and maximum values can be determined using `.min()` and `.max()`.

```
In [15]: # location of min price for both stocks  
omh[['MSFT', 'AAPL']].min()
```

```
Out[15]: MSFT      45.16  
          AAPL     106.75  
          dtype: float64
```

```
In [16]: # and location of the max  
omh[['MSFT', 'AAPL']].max()
```

```
Out[16]: MSFT      48.84  
          AAPL     115.93  
          dtype: float64
```

Some pandas statistical methods are referred to as indirect statistics as they don't return actual values but an indirect, related value. For example, `.idxmin()` and `.idxmax()` return the index position where the minimum and maximum value exist.

```
In [17]: # location of min price for both stocks  
omh[['MSFT', 'AAPL']].idxmin()
```

```
Out[17]: MSFT      11  
          AAPL      11  
          dtype: int64
```

```
In [18]: # and location of the max  
omh[['MSFT', 'AAPL']].idxmax()
```

```
Out[18]: MSFT      3  
          AAPL      2  
          dtype: int64
```


Locating the n-smallest and n-largest values

Sometime we need to know the n-smallest and n-largest values in a dataset. These can be found using `.nsmallest()` and `.nlargest()`. The following code demonstrates this by returning the 4-smallest values for MSFT.

```
In [19]: # get the 4 smallest values
omh.nsmallest(4, ['MSFT'])['MSFT']

Out[19]: 11    45.16
          12    45.74
          21    46.45
          10    46.67
Name: MSFT, dtype: float64
```

And likewise, the 4-largest values.

```
In [20]: # get the 4 largest values
omh.nlargest(4, ['MSFT'])['MSFT']

Out[20]: 3     48.84
          0     48.62
          1     48.46
          16    48.45
Name: MSFT, dtype: float64
```

The form for a series is slightly different since there are no columns to specify.

```
In [21]: # nsmallest on a Series
omh.MSFT.nsmallest(4)

Out[21]: 11    45.16
          12    45.74
          21    46.45
          10    46.67
Name: MSFT, dtype: float64
```


Calculating accumulated values

Accumulations are statistical methods that determine a value by continuously applying the next value in a series to the running result. Good examples are the cumulative product and cumulative sum of a Series. The following code demonstrates the calculation of a cumulative product.

```
In [22]: # calculate a cumulative product
pd.Series([1, 2, 3, 4]).cumprod()

Out[22]: 0      1
          1      2
          2      6
          3     24
         dtype: int64
```

The result is another `Series` that represents the accumulated value at each position. Here is the calculation of the cumulative sum of the same `Series`.

```
In [23]: # calculate a cumulative sum
pd.Series([1, 2, 3, 4]).cumsum()

Out[23]: 0      1
          1      3
          2      6
          3     10
         dtype: int64
```


Performing statistical processes on pandas objects

Descriptive statistics gives us the ability to understand numerous measures of data that describe a specific characteristic of the underlying data. Built into pandas are several classes of these descriptive statistical operations that can be applied to a Series or DataFrame.

Let's examine several facets of statistical analysis / techniques provided by pandas:

- Summary descriptive statistics
- Measuring central tendency: mean, median, and mode
- Variance and standard deviation

Retrieving summary descriptive statistics

pandas objects provide the `.describe()` method, which returns a set of summary statistics of the object's data. When applied to a `DataFrame`, `.describe()` will calculate the summary statistics for each column. The following code calculates these statistics for both stocks in `omh`.

```
In [24]: # get summary statistics for the stock data
omh.describe()

Out[24]:      MSFT        AAPL
count    22.000000  22.000000
mean    47.493182 112.411364
std     0.933077  2.388772
min    45.160000 106.750000
25%    46.967500 111.660000
50%    47.625000 112.530000
75%    48.125000 114.087500
max    48.840000 115.930000
```

With one quick method call, we have calculated the count, mean, standard deviation, minimum, and maximum values, and even the 25, 50, and 75 percentiles for both series of stock data.

`.describe()` can also be applied to a `Series`. The following code calculates summary statistics for just `MSFT`.

```
In [25]: # just the stats for MSFT
omh.MSFT.describe()

Out[25]: count    22.000000
mean    47.493182
std     0.933077
min    45.160000
25%    46.967500
50%    47.625000
75%    48.125000
max    48.840000
Name: MSFT, dtype: float64
```

And only the mean can be obtained as follows.

```
In [26]: # only the mean for MSFT
omh.MSFT.describe()['mean']

Out[26]: 47.493181818181817
```

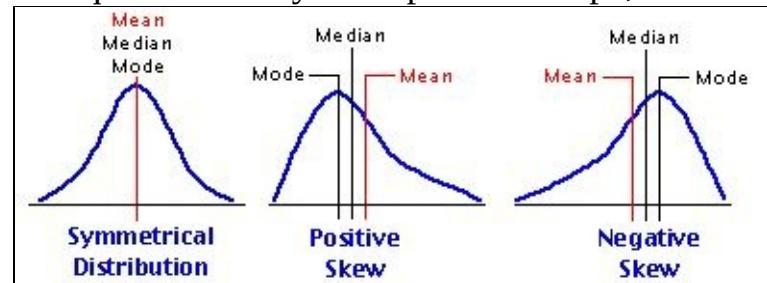
Non-numerical data will result in a slightly different set of summary statistics, returning the total number of items (`count`), the count of unique values (`unique`), most frequently occurring value (`top`), and the number of times it appears (`freq`):

```
In [27]: # get summary stats on non-numeric data
s = pd.Series(['a', 'a', 'b', 'c', np.nan])
s.describe()

Out[27]: count    4
unique   3
top      a
freq     2
dtype: object
```


Measuring central tendency: mean, median, and mode

Mean and median give us several useful measurements of data that help us begin to understand the distribution of values and also the shape of that distribution. The relationship of these three values gives us a quick summary concept of the shape, as shown in the following diagram:



Now let's examine how to find each of these values using pandas.

Calculating the mean

The mean, commonly referred to as the average, gives us a measurement of the **central tendency** of data. It is determined by summing up all the measurements and then dividing by the number of measurements.

The mean can be calculated using `.mean()`. The following code calculates the average of the prices for `AAPL` and `MSFT`:

```
In [28]: # the mean of all the columns in omh
omh.mean()

Out[28]: MSFT      47.493182
          AAPL     112.411364
          dtype: float64
```

Pandas has taken each column and independently calculated the mean for each. It returned the results as values in a Series that is indexed with the column names. The default is to apply the method on `axis=0`, applying the function to each column. This code switches the axis and returns the average price for all stocks on each day:

```
In [29]: # calc the mean of the values in each row
omh.mean(axis=1)[:5]

Out[29]: 0    81.845
          1    81.545
          2    82.005
          3    82.165
          4    81.710
          dtype: float64
```


Finding the median

The median gives us the central position in a series of values. By definition, the median is the value in the data where there are an equal number of other values that are both less and greater than it. The median is important as it is less influenced by outlying values and asymmetric data than the mean.

The median of the values is determined using the `.median()`:

```
In [30]: # calc the median of the values in each column  
omh.median()  
  
Out[30]: MSFT      47.625  
          AAPL     112.530  
          dtype: float64
```


Determining the mode

The mode is the most common value of a Series and is found with `.mode()`. The following code determines the mode of the given Series:

```
In [31]: # find the mode of this Series
s = pd.Series([1, 2, 3, 3, 5])
s.mode()

Out[31]: 0    3
          dtype: int64
```

Note that this has not returned a scalar value representing the mode, but a Series. This is because there can be more than one value for the mode of a Series. This is demonstrated with the following example:

```
In [32]: # there can be more than one mode
s = pd.Series([1, 2, 3, 3, 5, 1])
s.mode()

Out[32]: 0    1
          1    3
          dtype: int64
```


Calculating variance and standard deviation

In probability theory and statistics, standard deviation and variance give us a feel of how far some numbers are spread out from their mean. Let's briefly examine each.

Measuring variance

Variance gives us a feel for the overall amount of spread of the values from the mean. It is defined as

$$s^2 = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2$$

follows:

Essentially, this is stating that for each measurement, we calculate the value of the difference between the value and the mean. This can be a positive or negative value, so we square the result to make sure that negative values have cumulative effects on the result. These values are then summed up and divided by the number of measurements minus 1, giving an approximation of the average value of the differences.

In pandas, the variance is calculated using the `.var()` method. The following code calculates the variance

```
In [33]: # calc the variance of the values in each column
omh.var()

Out[33]: MSFT      0.870632
          AAPL     5.706231
          dtype: float64
```

of the price for both stocks:

Finding the standard deviation

Standard deviation is a similar measurement to variance. It is determined by calculating the square root of the variance and is defined as follows:

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2}$$

Remember that the variance squares the difference between all measurements and the mean. Because of this, the variance is not in the same units and the actual values. By using the square root of the variance, the standard deviation is in the same units as the values in the original dataset.

The standard deviation is calculated using the `.std()` method as is demonstrated here:

```
In [34]: # standard deviation
omh.std()

Out[34]: MSFT      0.933077
          AAPL      2.388772
          dtype: float64
```


Determining covariance and correlation

Covariance and correlation describe how two variables are related. This relation can be one of the following:

- Variables are positively related if they move in the same direction.
- Variables are inversely related if they move in opposite directions.

Both covariance and correlation indicate whether variables are positively or inversely related. Correlation also tells you the degree to which the variables tend to move together.

Calculating covariance

Covariance indicates how two variables are related. A positive covariance means the variables are positively related, while a negative covariance means they are inversely related:

$$cov_{x,y} = \frac{\sum_{i=1}^N (x_i - \bar{x})(y_i - \bar{y})}{N-1}$$

Covariance can be calculated using the `.cov()` method. The following code calculates the covariance between `MSFT` and `AAPL`:

```
In [35]: # covariance of MSFT vs AAPL  
omh.MSFT.cov(omh.AAPL)  
Out[35]: 1.9261240259740264
```


Determining correlation

Covariance can help determine whether values are related, but it does not give a sense of the degree to which the variables move together. To measure the degree to which variables move together, we need to calculate the correlation. Correlation is calculated by dividing the covariance by the product of the

$$r(x,y) = \frac{cov_{x,y}}{s_x s_y}$$

standard deviations of both sets of data:

Correlation standardizes the measure of interdependence between two variables and consequently tells you how closely the two variables move. The correlation measurement, called correlation coefficient, will always take a value between 1 and -1 and the interpretation of this value is as follows:

- If the correlation coefficient is 1.0, the variables have a perfect positive correlation. This means that if one variable moves by a given amount, the second moves proportionally in the same direction. A positive correlation coefficient of less than 1.0 but greater than 0.0 indicates a less-than-perfect positive correlation, with the strength of the correlation growing as the number approaches 1.0.
- If the correlation coefficient is 0.0, no relationship exists between the variables. If one variable moves, you can make no predictions about the movement of the other variable.
- If the correlation coefficient is -1.0, the variables are perfectly negatively correlated (or inversely correlated) and move opposite to each other. If one variable increases, the other variable decreases proportionally. A negative correlation coefficient greater than -1.0 but less than 0.0 indicates a less-than-perfect negative correlation, with the strength of the correlation growing as the number approaches -1.

Correlations in pandas are calculated using the `.corr()` method. The following code calculates the correlation of `MSFT` to `AAPL`.

```
In [36]: # correlation of MSFT relative to AAPL
omh.MSFT.corr(omh.AAPL)

Out[36]: 0.8641560684381171
```

This shows that the prices for `MSFT` and `AAPL` during this period demonstrate a high level of correlation. This does not mean that they are causal, with one affecting the other, but that there are likely shared influences on the values, such as being in similar markets.

Performing discretization and quantiling of data

Discretization is a means of slicing up continuous data into a set of "bins." Each value is then associated with a representative bin. The resulting distribution of the count of values in each bin can then be used to get an understanding of relative distribution of data across the different bins.

Discretization in pandas is performed using the `pd.cut()` and `pd.qcut()` functions. To demonstrate, let's start with the following set of 10000 random numbers created with a normal random number generator:

```
In [37]: # generate 10000 normal random #'s
np.random.seed(123456)
dist = np.random.normal(size = 10000)
dist

Out[37]: array([ 0.4691123, -0.28286334, -1.5090585, ..., 0.26296448,
   -0.83377412, -0.10418135])
```

This code shows us the mean and standard deviation of this dataset, which we expect to approach 0 and 1 as the sample size of the dataset increases (because it is normal):

```
In [38]: # show the mean and std
(dist.mean(), dist.std())

Out[38]: (-0.0028633240409066509, 1.0087162031998911)
```

We can split the numbers into equally sized bins using `pd.cut()`. The following code creates five evenly sized bins and determines the mapping of values:

```
In [39]: # cut into 5 equally sized bins
bins = pd.cut(dist, 5)
bins

Out[39]: [(-0.633, 0.81], (-0.633, 0.81], (-2.077, -0.633], (-2.077, -0.633], (0.81, 2.254], ..., (-2.077, -0.633], (-0.633, 0.81], (-0.633, 0.81], (-2.077, -0.633], (-0.633, 0.81])
Length: 10000
Categories (5, interval[float64]): [(-3.528, -2.077] < (-2.077, -0.633] < (-0.633, 0.81] < (0.81, 2.254] < (2.254, 3.698]]
```

The resulting `bins` object is a pandas Categorical variable. It consists of a set of labels and an index that describes how the data has been split. The `.categories` property will return the index and specify the intervals that pandas decided upon, given the range of values and the number of bins that was specified:

```
In [40]: # just the categories
bins.categories

Out[40]: IntervalIndex([(-3.528, -2.077], (-2.077, -0.633], (-0.633, 0.81], (0.81, 2.254], (2.254, 3.698]],
                      closed='right',
                      dtype='interval[float64]')
```

The `.codes` property is an array that specifies which of the bins (intervals) each item has been assigned:

```
In [41]: # codes tells us which bin each item is in
bins.codes

Out[41]: array([2, 2, 1, ..., 2, 1, 2], dtype=int8)
```

The notation for the intervals follows standard mathematical intervals, where a parenthesis represents that the end is open, while square brackets represent closed. Closed ends include values at that exact number. By default, pandas closes the right-hand side of the interval. The closed end can be moved to the left-hand

side of the interval using the `right=False` parameter of `pd.cut()`:

```
In [42]: # move the closed side of the interval to the left
pd.cut(dist, 5, right=False).categories

Out[42]: IntervalIndex([-3.521, -2.077), [-2.077, -0.633), [-0.633, 0.8
1), [0.81, 2.254), [2.254, 3.705])
closed='left',
dtype='interval[float64]')
```

Instead of passing an integer number of bins to cut data into, you can pass an array of values that represent the extent of the bins. A common example of this involves mapping ages into age range buckets. To demonstrate this, the following code generates 50 ages between 6 and 45:

```
In [43]: # generate 50 ages between 6 and 45
np.random.seed(123456)
ages = np.random.randint(6, 45, 50)
ages

Out[43]: array([ 7, 33, 38, 29, 42, 14, 16, 16, 18, 17, 26, 28, 44, 40, 2
0, 12, 8,
          10, 36, 29, 26, 26, 11, 29, 42, 17, 41, 35, 22, 40, 24, 2
1, 38, 33,
          26, 23, 16, 34, 26, 20, 18, 42, 27, 13, 37, 37, 10, 7, 1
0, 23])
```

We can specify ranges for the bins by passing them in an array where adjacent values represent the extent of each bin. This code cuts the data into the specified bins and reports the distribution of the ages found in each bin.

```
In [44]: # cut into ranges and then get descriptive stats
ranges = [6, 12, 18, 35, 50]
agebins = pd.cut(ages, ranges)
agebins.describe()

Out[44]:      counts   freqs
categories
(6, 12]           8    0.16
(12, 18]          9    0.18
(18, 35]          21   0.42
(35, 50]          12   0.24
```

To specify a name for each bin that is different from the standard mathematical notation, use the `labels` parameter:

```
In [45]: # add names for the bins
ranges = [6, 12, 18, 35, 50]
labels = ['Youth', 'Young Adult', 'Adult', 'Middle Aged']
agebins = pd.cut(ages, ranges, labels=labels)
agebins.describe()

Out[45]:      counts   freqs
categories
Youth           8    0.16
Young Adult     9    0.18
Adult           21   0.42
Middle Aged     12   0.24
```

 *This assignment of labels is convenient not only for textual output but also when plotting the bins as pandas will pass the bin names to be plotted on a chart.*

Data can also be sliced according to specified quantiles using `pd.qcut()`. This function will cut the values into bins such that each bin has the same number of items. From this result, we can determine the extent of the bins where the counts of values are evenly distributed.

The following code splits the random values from before into 5 quantile bins:

```
In [46]: # cut into quantiles
# 5 bins with an equal quantity of items
qbin = pd.qcut(dist, 5)
# this will tell us the range of values in each quantile
qbin.describe()

Out[46]:
```

categories	counts	freqs
(-3.522, -0.861]	2000	0.2
(-0.861, -0.241]	2000	0.2
(-0.241, 0.261]	2000	0.2
(0.261, 0.866]	2000	0.2
(0.866, 3.698]	2000	0.2

Instead of specifying an integer number of bins, it is also possible to specify the quantile ranges. The following code allocates bins based on plus and minus 3, 2, and 1 standard deviations. As this is normally distributed data, we would expect 0.1 percent, 2.1 percent, 13.6 percent, and 34.1 percent of the values on each side of the mean.

```
In [47]: # make the quantiles at the +/- 3, 2 and 1 std deviations
quantiles = [0,
             0.001,
             0.021,
             0.5-0.341,
             0.5,
             0.5+0.341,
             1.0-0.021,
             1.0-0.001,
             1.0]
qbin = pd.qcut(dist, quantiles)
# this data should be a perfect normal distribution
qbin.describe()

Out[47]:
```

categories	counts	freqs
(-3.522, -3.131]	10	0.001
(-3.131, -2.056]	200	0.020
(-2.056, -1.033]	1380	0.138
(-1.033, -0.00363]	3410	0.341
(-0.00363, 1.011]	3410	0.341
(1.011, 2.043]	1380	0.138
(2.043, 3.062]	200	0.020
(3.062, 3.698]	10	0.001

These are exactly the results we expect from this distribution.

Calculating the rank of values

Ranking helps us determine whether one of two items is *ranked higher* or *ranked lower* than another. Ranking reduces measures into a sequence of ordinal numbers that can be used to evaluate complex criteria based on the resulting order.

To demonstrate ranking, we will use the following series of data:

```
In [48]: # random data
np.random.seed(12345)
s = pd.Series(np.random.randn(5), index=list('abcde'))
s

Out[48]: a    -0.204708
         b     0.478943
         c    -0.519439
         d    -0.555730
         e    1.965781
         dtype: float64
```

These values can then be ranked using `.rank()`, which by default tells us the ordering of the labels from

```
In [49]: # rank the values
s.rank()

Out[49]: a    3.0
         b    4.0
         c    2.0
         d    1.0
         e    5.0
         dtype: float64
```

lowest value to highest value:

The result represents the order of the values numerically from smallest to largest (essentially, a sort). The lowest is `d` at `1.0` (which had the lowest value at `-0.555730`) and moving upwards in rank to `e` at `5.0` (a value of `1.965781`).



There are a number of options that you can use to change this default behavior, such as specifying a custom ranking function and how to decide a ranking when there is a tie.

Calculating the percent change at each sample of a series

The percentage change over a given number of periods can be calculated using the `.pct_change()` method. A sample use for percentage change is in calculating the rate of change of price of a stock. The following

```
In [50]: # calculate % change on MSFT
omh[['MSFT']].pct_change()[:5]
```



```
Out[50]:      MSFT
0      NaN
1 -0.003291
2 -0.007842
3  0.015807
4 -0.008600
```

code shows this for `MSFT`:

Performing moving-window operations

Pandas provides a number of functions to compute moving (also known as rolling) statistics. A rolling window computes the specified statistic on a specified interval of data. The window is then moved along the data by a specific interval and recalculated. The process continues until the window has been rolled across the entire dataset.

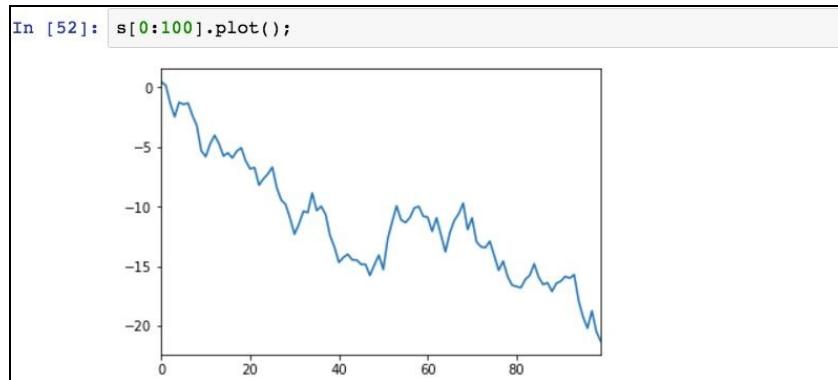
To demonstrate this, we will start with a series of 1000 random numbers that are cumulatively summed to

```
In [51]: # create a random walk
np.random.seed(123456)
s = pd.Series(np.random.randn(1000)).cumsum()
s[:5]

Out[51]: 0    0.469112
          1    0.186249
          2   -1.322810
          3   -2.458442
          4   -1.246330
          dtype: float64
```

form a random walk:

Zooming in on the first 100 values, we can see the movement of the data with the following plot:



To start creating a rolling window, we create a `Rolling` object using the `.rolling()` method with a specified window. In this case, we want to create a rolling window of 3:

```
In [53]: # calculate rolling window of three days
r = s.rolling(window=3)
r

Out[53]: Rolling [window=3,center=False,axis=0]
```

This `rolling` object specifies how wide we want the window to be but it has not performed the actual calculations. To do so, we can select one of the many methods of the `rolling` object representing the

```
r.
r.max
r.mean
r.median
r.min
r.name
r.ndim
r.quantile
r.skew
r.std
r.sum
...
```

statistical operation to apply (a few of which are shown in following image):

This code demonstrates calculating the rolling mean across the data:

```
In [54]: # the rolling mean at three days
means = r.mean()
means[:7]

Out[54]: 0      NaN
1      NaN
2    -0.222483
3    -1.198334
4    -1.675860
5    -1.708105
6    -1.322070
dtype: float64
```

Since our window size is $N=3$, the first mean in the result is at index label 2. We can verify that this value is the mean of the first three numbers:

```
In [55]: # check the mean of the first 3 numbers
s[0:3].mean()

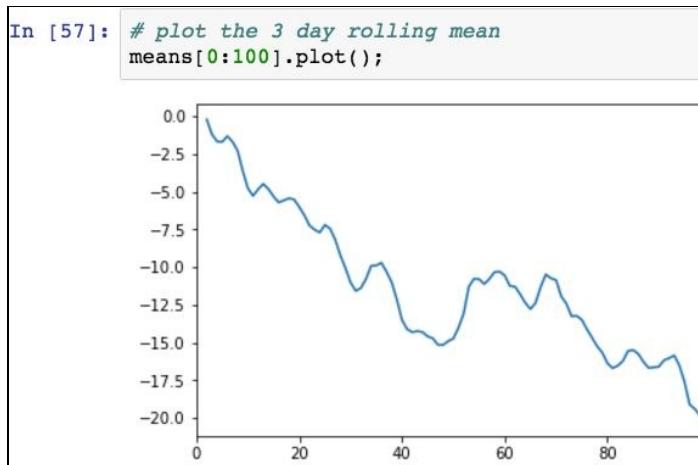
Out[55]: -0.22248276403642672
```

The window then rolls by one interval along the data. The next value calculated is at label 3 and represents the mean of values at labels 1, 2 and 3:

```
In [56]: # mean of 1 through 3
s[1:4].mean()

Out[56]: -1.1983341702095498
```

Plotting the result of a rolling mean on the first 100 values gives us this:



It can be seen by comparing this plot to the previous one that the rolling mean smoothens data across intervals.

Executing random sampling of data

Random sampling is a process of selecting values from a data sample at random positions. From pandas 0.19.2 on, this functionality has been added to the pandas `Series` and `DataFrame` objects, whereas in previous versions, you had to code this process on your own.

To demonstrate random sampling, let's start with the following `DataFrame`, representing four columns of 50

```
In [58]: # create a random sample of four columns of 50 items
np.random.seed(123456)
df = pd.DataFrame(np.random.randn(50, 4))
df[:5]
```

```
Out[58]:      0         1         2         3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
```

rows of random numbers:

We can take a sample of the data using the `.sample()` method while specifying the number of samples to retrieve. The following code samples three random rows:

```
In [59]: # sample three random rows
df.sample(n=3)
```

```
Out[59]:      0         1         2         3
15 -0.076467 -1.187678  1.130127 -1.436737
28 -2.182937  0.380396  0.084844  0.432390
48 -0.693921  1.613616  0.464000  0.227371
```

An alternative form is to specify a percentage of the data to randomly select. This code selects 10 percent of the rows.

```
In [60]: # sample 10% of the rows
df.sample(frac=0.1)
```

```
Out[60]:      0         1         2         3
37  1.126203 -0.977349  1.474071 -0.064034
10 -1.294524  0.413738  0.276662 -0.472035
4   -0.424972  0.567020  0.276232 -1.087401
14  0.410835  0.813850  0.132003 -0.827317
48 -0.693921  1.613616  0.464000  0.227371
```

Sampling in pandas can be performed with or without replacement, with the default being without replacement. To specify that we would like to use replacement, we simply use the `replace=True` parameter:

```
In [61]: # 10% with replacement
df.sample(frac=0.1, replace=True)
```

```
Out[61]:      0         1         2         3
27 -1.236269  0.896171 -0.487602 -0.082240
9   0.357021 -0.674600 -1.776904 -0.968914
27 -1.236269  0.896171 -0.487602 -0.082240
15 -0.076467 -1.187678  1.130127 -1.436737
9   0.357021 -0.674600 -1.776904 -0.968914
```


Summary

In this chapter, you learned how to perform numerical and statistical analyses on pandas objects. This included examination of many commonly used methods that will be used in calculating values and performing various analyses. We started with basic arithmetical operations and how data alignment factors into the operation and results. Then we covered many of the statistical operations provided in pandas, ranging from descriptive statistics to discretization through to rolling windows and random sampling. These lessons will set you up well for performing many real-world data analyses.

In the next chapter, we will change gears and look into how to load data from various data sources such as local files, databases, and remote web services.

Accessing Data

In almost any real-world data analysis, you need to load data from outside your program. Since pandas is built on Python, you can use any means available in Python to retrieve data. This makes it possible to access data from an almost unlimited set of sources, including but not limited to files, Excel spreadsheets, websites and services, databases, and cloud services.

However, when using standard Python functions to load data, you need to convert Python objects into pandas `Series` or `DataFrame` objects. This increases the complexity of your code. To help with managing this complexity, pandas offers a number of facilities to load data from various sources directly into pandas objects. We will examine many of these in this chapter.

Specifically, in this chapter, we will cover:

- Reading a CSV file into a DataFrame
- Specifying the index column when reading a CSV file
- Data type inference and specification
- Specifying column names
- Specifying specific columns to load
- Saving data to a CSV file
- Working with general field-delimited data
- Handling variants of formats in field-delimited data
- Reading and writing data in Excel format
- Reading and writing JSON files
- Reading HTML data from the web
- Reading and writing HDF5 format files
- Reading and writing from/to SQL databases
- Reading stock data from Yahoo! and Google Finance
- Reading options data from Google Finance
- Reading economic data from the FRED of St. Louis
- Accessing Kenneth French's data
- Accessing World Bank data

Configuring pandas

We start with the standard imports and options for pandas to facilitate the examples.

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set some pandas options controlling output format
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 90)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```


Working with CSV and text/tabular format data

CSV formatted data is likely to be one of the most common forms of data you may use in pandas. Many web-based services provide data in a CSV format, as well as many information systems within an enterprise. It is an easy format to use and is commonly used as an export format for spreadsheet applications such as Excel.

A CSV is a file consisting of multiple lines of text-based data, with values separated by commas. It can be thought of as a table of data similar to a single sheet in a spreadsheet program. Each row of the data is in its own line in the file, and each column for each row is stored in text format, with a comma separating the data in each column.



For more details on the specifics of CSV files, feel free to visit http://en.wikipedia.org/wiki/Comma-separated_values.

As CSV is so common and easily understood, we will spend most of the time describing how to read and write pandas data in this format. Lessons learned from CSV methods will apply to the other formats as well, and allow a little more expediency when covering these other formats.

Examining the sample CSV data set

We will start by reading a simple CSV file, `data/msft.csv` (in the book's source `data` folder). This file is a snapshot of stock values for the `MSFT` ticker. The first several lines of this file can be examined using the `!head` command (on a Windows system, use the `type` command):

```
In [2]: # view the first five lines of data/msft.csv
!head -n 5 data/msft.csv # mac or Linux
# type data/msft.csv # on windows, but shows the entire file

Date,Open,High,Low,Close,Volume
7/21/2014,83.46,83.53,81.81,81.93,2359300
7/18/2014,83.3,83.4,82.52,83.35,4020800
7/17/2014,84.35,84.63,83.33,83.63,1974000
7/16/2014,83.77,84.91,83.66,84.91,1755600
```

The first row of the file contains the names of all the columns represented in the data, each separated with a comma. Each row then represents a sampling of the values on the specific date.

Reading a CSV file into a DataFrame

The data in `data/msft.csv` is perfect to read into `DataFrame`. All its data is complete and has column names in the first row. All that we need to do to read this data into `DataFrame` is to use the pandas `pd.read_csv()`

```
In [3]: # read in msft.csv into a DataFrame
msft = pd.read_csv("data/msft.csv")
msft[:5]
```

	Date	Open	High	Low	Close	Volume
0	7/21/2014	83.46	83.53	81.81	81.93	2359300
1	7/18/2014	83.30	83.40	82.52	83.35	4020800
2	7/17/2014	84.35	84.63	83.33	83.63	1974000
3	7/16/2014	83.77	84.91	83.66	84.91	1755600
4	7/15/2014	84.30	84.38	83.20	83.58	1874700

function:

Wow, that was easy! Pandas has realized that the first line of the file contains the names of the columns and bulk read in the data to DataFrame.

Specifying the index column when reading a CSV file

In the previous example, the index is numerical, starting from 0, instead of by date. This is because pandas does not assume that any specific column in the file should be used as the index. To help this situation, you can specify which column(s) should be the index in the call to `read_csv()` using the `index_col` parameter by assigning it the zero-based position of the column to be used as the index.

The following reads the data and tells pandas to use the column at position 0 in the file as the index (the Date column):

```
In [4]: # use column 0 as the index
msft = pd.read_csv("data/msft.csv", index_col=0)
msft[:5]

Out[4]:      Open   High    Low  Close  Volume
Date
7/21/2014  83.46  83.53  81.81  81.93  2359300
7/18/2014  83.30  83.40  82.52  83.35  4020800
7/17/2014  84.35  84.63  83.33  83.63  1974000
7/16/2014  83.77  84.91  83.66  84.91  1755600
7/15/2014  84.30  84.38  83.20  83.58  1874700
```


Data type inference and specification

An examination of the types of each column shows that pandas has attempted to infer the types of the columns from their content:

```
In [5]: # examine the types of the columns in this DataFrame  
msft.dtypes
```

```
Out[5]: Open      float64  
High       float64  
Low        float64  
Close      float64  
Volume     int64  
dtype: object
```

To force the types of columns, use the `dtypes` parameter of `pd.read_csv()`. The following forces the `Volume` column to also be `float64`:

```
In [6]: # specify that the Volume column should be a float64  
msft = pd.read_csv("data/msft.csv",  
                   dtype = { 'Volume' : np.float64})  
msft.dtypes
```

```
Out[6]: Date      object  
Open       float64  
High       float64  
Low        float64  
Close      float64  
Volume     float64  
dtype: object
```


Specifying column names

It is also possible to specify the column names at the time of reading the data, by using the `names` parameter:

```
In [7]: # specify a new set of names for the columns
# all lower case, remove space in Adj Close
# also, header=0 skips the header row
df = pd.read_csv("data/msft.csv",
                  header=0,
                  names=['date', 'open', 'high', 'low',
                         'close', 'volume'])
df[:5]
```

```
Out[7]:      date    open    high    low   close  volume
0  7/21/2014  83.46  83.53  81.81  81.93  2359300
1  7/18/2014  83.30  83.40  82.52  83.35  4020800
2  7/17/2014  84.35  84.63  83.33  83.63  1974000
3  7/16/2014  83.77  84.91  83.66  84.91  1755600
4  7/15/2014  84.30  84.38  83.20  83.58  1874700
```

Since we specified the names of the columns, we need to skip over the 'column names' row in the file, which was performed with `header=0`. If not done, pandas will assume that the first row is part of the data, which will cause some issues later in processing.

Specifying specific columns to load

It is also possible to specify which columns to load when reading the file. This can be useful if you have a lot of columns in the file and some are of no interest to your analysis, and you want to save the time and memory required to read and store them. Specifying which columns to read is accomplished with the `usecols` parameter, which can be passed a list of column names or column offsets.

To demonstrate, the following reads only the `Date` and `Close` columns, and uses `Date` as the index:

```
In [8]: # read in data only in the Date and Close columns
# and index by the Date column
df2 = pd.read_csv("data/msft.csv",
                  usecols=['Date', 'Close'],
                  index_col=['Date'])
df2[:5]

Out[8]:          Close
Date
7/21/2014  81.93
7/18/2014  83.35
7/17/2014  83.63
7/16/2014  84.91
7/15/2014  83.58
```


Saving DataFrame to a CSV file

CSV files can be saved from `DataFrame` using the `.to_csv()` method. To demonstrate saving data to a CSV file, we will save the `df2` object with the revised column names to a new file named `data/msft_modified.csv`:

```
In [9]: # save df2 to a new csv file  
# also specify naming the index as date  
df2.to_csv("data/msft_modified.csv", index_label='date')
```

It is necessary to tell the method that the index label should be saved with the column name of `date` using `index_label='date'`. Otherwise, the index does not have a name added to the first row of the file, which will make it difficult to read back properly.

To examine that this worked properly, we can explore the new file to view some of its content using the `!head` command (and if on a Windows system, use the `!type` command):

```
In [10]: # view the start of the file just saved  
!head -n 5 data/msft_modified.csv  
#type data/msft_modified.csv # windows
```

```
date,Close  
7/21/2014,81.93  
7/18/2014,83.35  
7/17/2014,83.63  
7/16/2014,84.91
```


Working with general field-delimited data

CSV is actually a specific implementation of what is referred to as field-delimited data. In field-delimited data, items in each row are separated by a specific symbol. In the case of CSV, it happens to be a comma. However, other symbols are common, such as the | (pipe) symbol. When using a | character, the data is often called pipe-delimited data.

Pandas provides the `pd.read_table()` function to facilitate reading field-delimited data. The following example uses this function to read the `data/msft.csv` file by specifying a comma as the value to the `sep` parameter:

```
In [11]: # use read_table with sep=',' to read a CSV
df = pd.read_table("data/msft.csv", sep=',')
df[:5]
```

	Date	Open	High	Low	Close	Volume
0	7/21/2014	83.46	83.53	81.81	81.93	2359300
1	7/18/2014	83.30	83.40	82.52	83.35	4020800
2	7/17/2014	84.35	84.63	83.33	83.63	1974000
3	7/16/2014	83.77	84.91	83.66	84.91	1755600
4	7/15/2014	84.30	84.38	83.20	83.58	1874700

Pandas does not provide a `.to_table()` method as an analogous write method to `.to_csv()`. However, the `.to_csv()` method can be used to write field-delimited data using a different delimiter than a comma. As an example, the following writes a pipe-delimited version of the data:

```
In [12]: # save as pipe delimited
df.to_csv("data/msft_piped.txt", sep='|')
# check that it worked
!head -n 5 data/msft_piped.txt # osx or Linux
# type data/msft_piped.txt # on windows
```

	Date	Open	High	Low	Close	Volume
0	7/21/2014	83.46	83.53	81.81	81.93	2359300
1	7/18/2014	83.3	83.4	82.52	83.35	4020800
2	7/17/2014	84.35	84.63	83.33	83.63	1974000
3	7/16/2014	83.77	84.91	83.66	84.91	1755600

Handling variants of formats in field-delimited data

Data in a field-delimited file may contain extraneous headers and footers. Examples include company information at the top, such as an invoice number, addresses, and summary footers. There are also cases where data is stored on every other line. These situations will cause errors when loading data like this. To handle these scenarios, the pandas `pd.read_csv()` and `pd.read_table()` methods have some useful parameters to help us out.

To demonstrate, take the following variation on the `MSFT` stock data, which has extra rows of what can be referred to as noise information:

```
In [13]: # messy file
!head -n 6 data/msft2.csv # osx or Linux
# type data/msft2.csv # windows

This is fun because the data does not start on the first line,....
Date,Open,High,Low,Close,Volume
"...."
And there is space between the header row and data,....
7/21/2014,83.46,83.53,81.81,81.93,2359300
7/18/2014,83.3,83.4,82.52,83.35,4020800
```

This situation can be handled using the `skiprows` parameter, which informs pandas to skip rows 0, 2, and 3:

```
In [14]: # read, but skip rows 0, 2 and 3
df = pd.read_csv("data/msft2.csv", skiprows=[0, 2, 3])
df[:5]
```

```
Out[14]:      Date  Open  High  Low  Close  Volume
0  7/21/2014  83.46  83.53  81.81  81.93  2359300
1  7/18/2014  83.30  83.40  82.52  83.35  4020800
2  7/17/2014  84.35  84.63  83.33  83.63  1974000
3  7/16/2014  83.77  84.91  83.66  84.91  1755600
4  7/15/2014  84.30  84.38  83.20  83.58  1874700
```

Another common situation is where a file has content at the end of the file, which should be ignored to prevent an error. Take the following data as an example:

```
In [15]: # another messy file, with the mess at the end  
!cat data/msft_with_footer.csv # osx or Linux  
# type data/msft_with_footer.csv # windows
```

```
Date,Open,High,Low,Close,Volume,Adj Close  
2014-07-21,83.46,83.53,81.81,81.93,2359300,81.93  
2014-07-18,83.30,83.40,82.52,83.35,4020800,83.35
```

Uh oh, there is stuff at the end.

This file will cause an exception during reading, but it can be handled using the `skip_footer` parameter, which specifies how many lines at the end of the file to ignore:

```
In [16]: # skip only two lines at the end  
df = pd.read_csv("data/msft_with_footer.csv",  
                 skipfooter=2,  
                 engine = 'python')  
df
```

```
Out[16]:      Date    Open    High     Low   Close   Volume  
0  7/21/2014  83.46  83.53  81.81  81.93  2359300  
1  7/18/2014  83.30  83.40  82.52  83.35  4020800
```



Note that we had to specify `engine = 'python'`. At least with Anaconda, without this option a warning is given, as this option is not implemented by the default underlying C implementation. This forces it to use a Python implementation.

Suppose the file is large and you only want to read the first few rows, as you only want the data at the start of the file and do not want to read it all into the memory. This can be handled with the `nrows` parameter:

```
In [17]: # only process the first three rows  
pd.read_csv("data/msft.csv", nrows=3)
```

```
Out[17]:      Date    Open    High     Low   Close   Volume  
0  7/21/2014  83.46  83.53  81.81  81.93  2359300  
1  7/18/2014  83.30  83.40  82.52  83.35  4020800  
2  7/17/2014  84.35  84.63  83.33  83.63  1974000
```

You can also skip a specific number of rows at the start of a file and read to the end, or you can read just a few lines once you get to that point in the file. To do this, use the `skiprows` parameter. The following example skips 100 rows and then reads in the next 5:

```
In [18]: # skip 100 lines, then only process the next five
pd.read_csv("data/msft.csv", skiprows=100, nrows=5,
            header=0,
            names=['date', 'open', 'high', 'low',
                   'close', 'vol'])

Out[18]:      date    open    high     low   close    vol
0  3/3/2014  80.35  81.31  79.91  79.97  5004100
1  2/28/2014  82.40  83.42  82.17  83.42  2853200
2  2/27/2014  84.06  84.63  81.63  82.00  3676800
3  2/26/2014  82.92  84.03  82.43  83.81  2623600
4  2/25/2014  83.80  83.80  81.72  83.08  3579100
```



The preceding example also skipped reading the header line, so it was necessary to inform the process to not look for a header and to use the specified names.

Reading and writing data in Excel format

Pandas supports reading data in Excel 2003 and newer formats, using the `pd.read_excel()` function or via the `ExcelFile` class. Internally, both techniques use either the `xlrd` or `openpyxl` packages, so you will need to ensure that one of them is installed in your Python environment.

For demonstration, a `data/stocks.xlsx` file is provided with the sample data. If you open it in Excel, you will see something similar to what is shown in the following screenshot:

	Date	Open	High	Low	Close	Volume
1	7/21/2014	83.46	83.53	81.81	81.93	2359300
2	7/18/2014	83.3	83.4	82.52	83.35	4020800
3	7/17/2014	84.35	84.63	83.33	83.63	1974000

The workbook contains two sheets, `msft` and `aapl`, which hold the stock data for each respective stock.

The following then reads the `data/stocks.xlsx` file into a `DataFrame`:

```
In [19]: # read excel file
# only reads first sheet (msft in this case)
df = pd.read_excel("data/stocks.xlsx")
df[:5]
```

```
Out[19]:      Date    Open    High     Low   Close  Volume
0 2014-07-21  83.46  83.53  81.81  81.93  2359300
1 2014-07-18  83.30  83.40  82.52  83.35  4020800
2 2014-07-17  84.35  84.63  83.33  83.63  1974000
3 2014-07-16  83.77  84.91  83.66  84.91  1755600
4 2014-07-15  84.30  84.38  83.20  83.58  1874700
```

This has read only content from the first worksheet in the Excel file (the `msft` worksheet), and has used the contents of the first row as column names. To read the other worksheet, you can pass the name of the worksheet using the `sheetname` parameter:

```
In [20]: # read from the aapl worksheet
aapl = pd.read_excel("data/stocks.xlsx", sheetname='aapl')
aapl[:5]
```

```
Out[20]:      Date  Open  High  Low  Close  Volume
0 2014-07-21  94.99  95.00  93.72  93.94  38887700
1 2014-07-18  93.62  94.74  93.02  94.43  49898600
2 2014-07-17  95.03  95.28  92.57  93.09  57152000
3 2014-07-16  96.97  97.10  94.74  94.78  53396300
4 2014-07-15  96.80  96.85  95.03  95.32  45477900
```

Like with `pd.read_csv()`, many assumptions are made about column names, data types, and indexes. All the options we covered for `pd.read_csv()` to specify this information, also apply to the `pd.read_excel()` function.

Excel files can be written using the `.to_excel()` method of `DataFrame`. Writing to the XLS format requires the inclusion of the `xlwt` package, so make sure it is loaded in your Python environment before trying.

The following writes the data we just acquired to `stocks2.xls`. The default is to store `DataFrame` in the `Sheet1` worksheet:

```
In [21]: # save to an .XLS file, in worksheet 'Sheet1'
df.to_excel("data/stocks2.xls")
```

Opening this in Excel shows you the following:

	A	B	C	D	E	F	G
1		Date	Open	High	Low	Close	Volume
2	0	2014-07-21 00:00:00	83.46	83.53	81.81	81.93	2359300
3	1	2014-07-18 00:00:00	83.3	83.4	82.52	83.35	4020800
4	2	2014-07-17 00:00:00	84.35	84.63	83.33	83.63	1974000
5	3	2014-07-16 00:00:00	83.77	84.91	83.66	84.91	1755600

You can specify the name of the worksheet using the `sheet_name` parameter:

```
In [22]: # write making the worksheet name MSFT
df.to_excel("data/stocks_msft.xls", sheet_name='MSFT')
```

In Excel, we can see that the sheet has been named MSFT:

The screenshot shows a Microsoft Excel window with the title bar "stocks_msft.xls [Com... Michael Heydt]". The ribbon menu includes File, Home, Insert, Page I, Form I, Data, Review, View, Add-i, Team, Tell me, and Share. The active sheet is "MSFT". The data starts at row 1 with columns A through G labeled Date, Open, High, Low, Close, and Volume. Rows 2 through 5 show data for July 21, 18, 17, and 16 respectively. The status bar at the bottom right shows "Ready" and "100%".

	A	B	C	D	E	F	G
1		Date	Open	High	Low	Close	Volume
2	0	2014-07-21 00:00:00	83.46	83.53	81.81	81.93	2359300
3	1	2014-07-18 00:00:00	83.3	83.4	82.52	83.35	4020800
4	2	2014-07-17 00:00:00	84.35	84.63	83.33	83.63	1974000
5	3	2014-07-16 00:00:00	83.77	84.91	83.66	84.91	1755600

To write more than one `DataFrame` to a single Excel file using each `DataFrame` object on a separate worksheet, use the `ExcelWriter` object along with the `with` keyword. `ExcelWriter` is part of pandas, but you will need to make sure it is imported, as it is not in the top-level namespace of pandas. The following writes two `DataFrame` objects to two different worksheets in one Excel file:

```
In [23]: # write multiple sheets
# requires use of the ExcelWriter class
from pandas import ExcelWriter
with ExcelWriter("data/all_stocks.xls") as writer:
    aapl.to_excel(writer, sheet_name='AAPL')
    df.to_excel(writer, sheet_name='MSFT')
```

We can see that there are two worksheets in the Excel file:

The screenshot shows a Microsoft Excel window with the title bar "all_stocks.xls [Compa... Michael Heydt]". The ribbon menu includes File, Home, Insert, Page I, Form I, Data, Review, View, Add-i, Team, Tell me, and Share. The active sheet is "AAPL". The data starts at row 1 with columns A through G labeled Date, Open, High, Low, Close, and Volume. Rows 2 through 5 show data for July 21, 18, 17, and 16 respectively. The status bar at the bottom right shows "Ready" and "100%". Below the "AAPL" tab, the "MSFT" tab is visible.

	A	B	C	D	E	F	G
1		Date	Open	High	Low	Close	Volume
2	0	2014-07-21 00:00:00	94.99	95	93.72	93.94	38887700
3	1	2014-07-18 00:00:00	93.62	94.74	93.02	94.43	49898600
4	2	2014-07-17 00:00:00	95.03	95.28	92.57	93.09	57152000
5	3	2014-07-16 00:00:00	96.97	97.1	94.74	94.78	53396300

Writing to XLSX files uses the same function, but specifies `.xlsx` as the file extension:

```
In [24]: # write to xlsx
df.to_excel("data/msft2.xlsx")
```


Reading and writing JSON files

Pandas can read and write data stored in the **JavaScript Object Notation (JSON)** format. This is one of my favorites, due to its ability to be used across platforms and with many programming languages.

To demonstrate saving as JSON, we will first save the Excel data we just read into a JSON file and examine the contents:

```
In [25]: # write the excel data to a JSON file  
df[:5].to_json("data/stocks.json")  
!cat data/stocks.json # osx or Linux  
#type data/stocks.json # windows
```

```
{"Date": {"0": 1405900800000, "1": 1405641600000, "2": 1405555200000,  
"3": 1405468800000, "4": 1405382400000}, "Open": {"0": 83.46, "1": 83.  
3, "2": 84.35, "3": 83.77, "4": 84.3}, "High": {"0": 83.53, "1": 83.4, "2": 8  
4.63, "3": 84.91, "4": 84.38}, "Low": {"0": 81.81, "1": 82.52, "2": 83.3  
3, "3": 83.66, "4": 83.2}, "Close": {"0": 81.93, "1": 83.35, "2": 83.6  
3, "3": 84.91, "4": 83.58}, "Volume": {"0": 2359300, "1": 4020800, "2": 197  
4000, "3": 1755600, "4": 1874700}, "Adj Close": {"0": 81.93, "1": 83.3  
5, "2": 83.63, "3": 84.91, "4": 83.58}}
```

JSON-based data can be read with the `pd.read_json()` function:

```
In [26]: # read data in from JSON  
df_from_json = pd.read_json("data/stocks.json")  
df_from_json[:5]
```

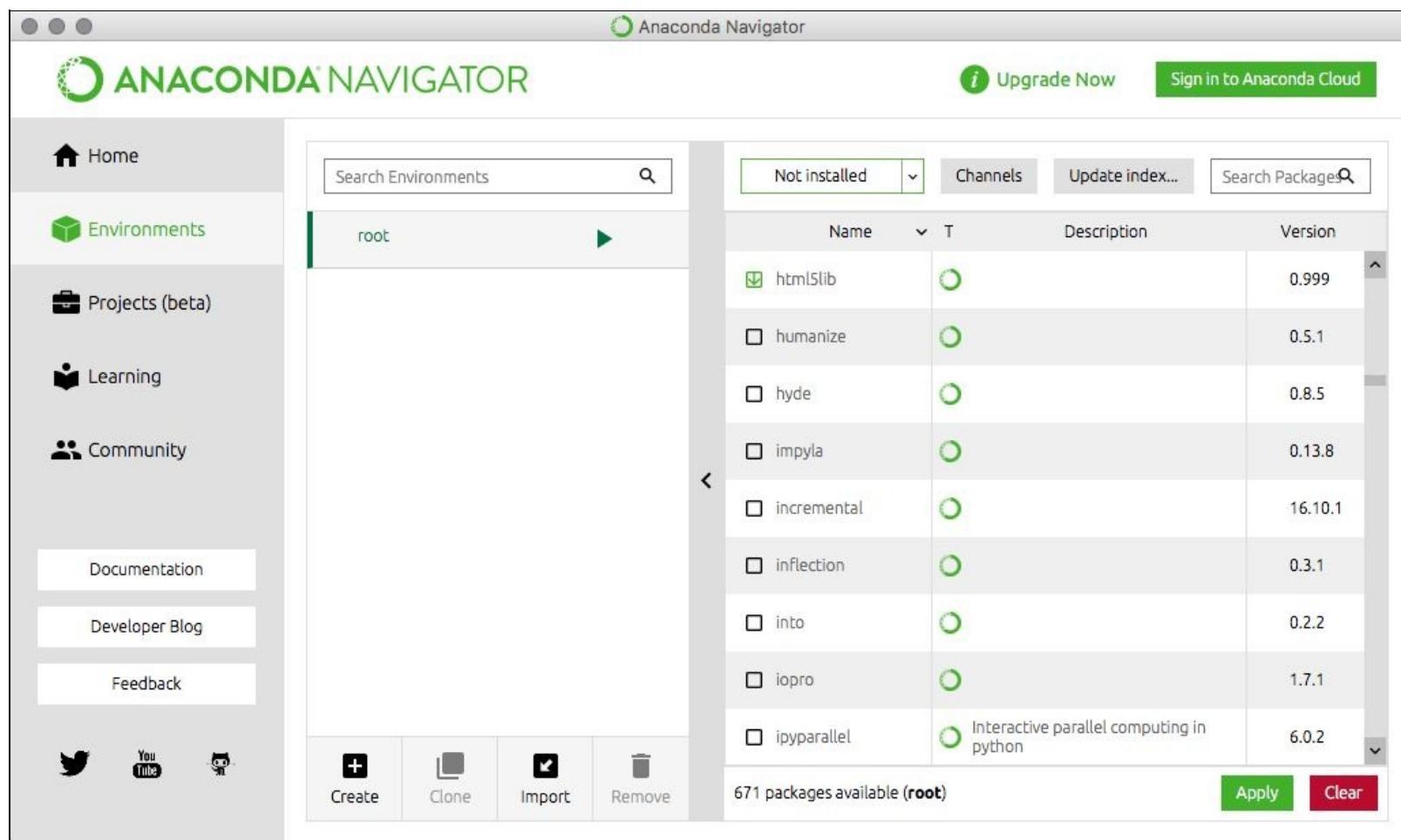
```
Out[26]:   Adj Close  Close      Date    High     Low    Open  Volume  
0        81.93  81.93 2014-07-21  83.53  81.81  83.46  2359300  
1        83.35  83.35 2014-07-18  83.40  82.52  83.30  4020800  
2        83.63  83.63 2014-07-17  84.63  83.33  84.35  1974000  
3        84.91  84.91 2014-07-16  84.91  83.66  83.77  1755600  
4        83.58  83.58 2014-07-15  84.38  83.20  84.30  1874700
```

Note the two slight differences here, caused by the reading/writing of data from JSON. First, the columns have been reordered alphabetically. Second, the index for `DataFrame`, although containing content, is sorted as a string. These issues can be fixed easily, but they will not be covered here for brevity.

Reading HTML data from the web

Pandas has support for reading data from HTML files (or HTML from URLs). Underneath the covers, pandas makes use of the `LXML`, `HTML5Lib`, and `BeautifulSoup4` packages. These packages provide some impressive capabilities for reading and writing HTML tables.

Your default installation of Anaconda may not include these packages. If you get errors using this function, install the appropriate library based on the error, using the Anaconda Navigator:



Else, you can use `pip`:

```
Michaels-iMac-2:~ michaelheydt$ /Users/michaelheydt/.anaconda/navigator/a.tool ; exit;
[~/Users/michaelheydt/anaconda) bash-3.2$ pip install html5lib
Collecting html5lib
  Downloading html5lib-0.999999999-py2.py3-none-any.whl (112kB)
    100% |████████████████████████████████| 122kB 126kB/s
Requirement already satisfied: six in ./anaconda/lib/python3.6/site-packages (from html5lib)
Collecting webencodings (from html5lib)
  Downloading webencodings-0.5.1-py2.py3-none-any.whl
Requirement already satisfied: setuptools>=18.5 in ./anaconda/lib/python3.6/site-packages/setuptools-27.2.0-py3.6.egg (from html5lib)
Installing collected packages: webencodings, html5lib
Successfully installed html5lib-0.999999999 webencodings-0.5.1
(/Users/michaelheydt/anaconda) bash-3.2$
```

The `pd.read_html()` function will read HTML from a file (or URL) and parse all HTML tables found in the

content into one or more pandas `DataFrame` objects. The function always returns a list of `DataFrame` objects (actually, zero or more, depending on the number of tables found in the HTML).

To demonstrate, we will read table data from the FDIC failed bank list, located at <https://www.fdic.gov/bank/individual/failed/banklist.html>. Viewing the page, you can see there is a list of quite a few failed banks.

This data is actually very simple to read with pandas and its `pd.read_html()` function:

```
In [27]: # the URL to read
url = "http://www.fdic.gov/bank/individual/failed/banklist.html"
# read it
banks = pd.read_html(url)
```

```
In [28]: # examine a subset of the first table read
banks[0][0:5].iloc[:,0:2]
```

```
Out[28]:
```

	Bank Name	City
0	Fayette County Bank	Saint Elmo
1	Guaranty Bank, (d/b/a BestBank in Georgia & Mi...	Milwaukee
2	First NBC Bank	New Orleans
3	Proficio Bank	Cottonwood Heights
4	Seaway Bank and Trust Company	Chicago

Again, that was almost too easy!

A `DataFrame` can be written to an HTML file with the `.to_html()` method. This method creates a file containing only the `<table>` tag for the data (not the entire HTML document). The following writes the stock data we read earlier to an HTML file:

```
In [29]: # read the stock data
df = pd.read_excel("data/stocks.xlsx")
# write the first two rows to HTML
df.head(2).to_html("data/stocks.html")
# check the first 28 lines of the output
!head -n 10 data/stocks.html # max on Linux
# type data/stocks.html # window, but prints the entire file
```

```
<table border="1" class="dataframe">
<thead>
<tr style="text-align: right;">
<th></th>
<th>Date</th>
<th>Open</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Volume</th>
```

Viewing this in the browser looks like what is shown in the following screenshot:

The screenshot shows a web browser window with the title 'stocks.html'. The address bar indicates the file is located at 'file:///Users/michaelheydt/Dro...'. The browser interface includes standard navigation buttons (back, forward, search) and a tab labeled 'Michael'. Below the header, there is a table with the following data:

	Date	Open	High	Low	Close	Volume
0	2014-07-21	83.46	83.53	81.81	81.93	2359300
1	2014-07-18	83.30	83.40	82.52	83.35	4020800

This is useful, as you can use pandas to write HTML fragments to be included in websites, update them when needed, and thereby have the new data available to the site statically instead of through a more complicated data query or service call.

Reading and writing HDF5 format files

HDF5 is a data model, library, and file format to store and manage data. It is commonly used in scientific computing environments. It supports an unlimited variety of data types, and is designed for flexible and efficient I/O, and for high volume and complex data.

HDF5 is portable and extensible, allowing applications to evolve in their use of HDF5. The HDF5 Technology Suite includes tools and applications to manage, manipulate, view, and analyze data in the HDF5 format. HDF5 is:

- A versatile data model that can represent very complex data objects and a wide variety of metadata
- A completely portable file format with no limit on the number or size of data objects in the collection
- A software library that runs on a range of computational platforms, from laptops to massively parallel systems, and implements a high-level API with C, C++, Fortran 90, and Java interfaces
- A rich set of integrated performance features that allow for access time and storage space optimizations
- Tools and applications to manage, manipulate, view, and analyze the data in the collection

`HDFStore` is a hierarchical, dictionary-like object that reads and writes pandas objects to the HDF5 format. Under the covers, `HDFStore` uses the `PyTables` library, so make sure that it is installed if you want to use this format.

The following demonstrates writing `DataFrame` into an HDF5 format. The output shows that the HDF5 store has a root level object named `df`, which is a `DataFrame` and whose shape is eight rows of three columns:

```
In [30]: # seed for replication
np.random.seed(123456)
# create a DataFrame of dates and random numbers in three columns
df = pd.DataFrame(np.random.randn(8, 3),
                  index=pd.date_range('1/1/2000', periods=8),
                  columns=[ 'A', 'B', 'C'])

# create HDF5 store
store = pd.HDFStore('data/store.h5')
store['df'] = df # persisting happened here
store
```

```
Out[30]: <class 'pandas.io.pytables.HDFStore'>
File path: data/store.h5
/df          frame      (shape->[8,3])
```

The following reads the HDF5 store and retrieves a `DataFrame`:

```
In [31]: # read in data from HDF5
store = pd.HDFStore("data/store.h5")
df = store['df']
df[:5]
```

```
Out[31]:
```

	A	B	C
2000-01-01	0.469112	-0.282863	-1.509059
2000-01-02	-1.135632	1.212112	-0.173215
2000-01-03	0.119209	-1.044236	-0.861849
2000-01-04	-2.104569	-0.494929	1.071804
2000-01-05	0.721555	-0.706771	-1.039575

A `DataFrame` is written to the HDF5 file at the point it is assigned to the `store` object. Changes to `DataFrame` made after that point are not persisted, at least not until the object is assigned to the data store object again. The following demonstrates this by making a change to `DataFrame` and then reassigning it to the HDF5 store, thereby updating the data store:

```
In [32]: # this changes the DataFrame, but did not persist
df.iloc[0].A = 1
# to persist the change, assign the DataFrame to the
# HDF5 store object
store['df'] = df
# it is now persisted
# the following loads the store and
# shows the first two rows, demonstrating
# the persisting was done
pd.HDFStore("data/store.h5")['df'][:5] # it's now in there
```

```
Out[32]:
```

	A	B	C
2000-01-01	1.000000	-0.282863	-1.509059
2000-01-02	-1.135632	1.212112	-0.173215
2000-01-03	0.119209	-1.044236	-0.861849
2000-01-04	-2.104569	-0.494929	1.071804
2000-01-05	0.721555	-0.706771	-1.039575

Accessing CSV data on the web

It is quite common to read data off the web and from the internet. Pandas makes it easy to read data from the web. All the pandas functions that we have examined can also be given an HTTP URL, FTP address, or S3 address, instead of a local file path, and all of them work just the same as they work with a local file.

The following demonstrates how easy it is to directly make HTTP requests using the existing `pd.read_csv()` function. The following retrieves the daily stock data for Microsoft in April 2017, directly from the Google Finance web service via its HTTP query string model:

```
In [33]: # read csv directly from Yahoo! Finance from a URL
msft_hist = pd.read_csv(
    "http://www.google.com/finance/historical?" +
    "q=NASDAQ:MSFT&startdate=Apr+01%2C+2017&" +
    "enddate=Apr+30%2C+2017&output=csv")
msft_hist[:5]
```

```
Out[33]:
```

	Date	Open	High	Low	Close	Volume
0	28-Apr-17	68.91	69.14	67.69	68.46	39548818
1	27-Apr-17	68.15	68.38	67.58	68.27	34970953
2	26-Apr-17	68.08	68.31	67.62	67.83	26190770
3	25-Apr-17	67.90	68.04	67.60	67.92	30242730
4	24-Apr-17	67.48	67.66	67.10	67.53	29769976

Reading and writing from/to SQL databases

Pandas can read data from any SQL database that supports Python data adapters that respect the Python DB-API. Reading is performed using the `pandas.io.sql.read_sql()` function, and writing to SQL databases is done using the `.to_sql()` method of `DataFrame`.

To demonstrate, the following reads the stock data from `msft.csv` and `aapl.csv`. It then makes a connection to an SQLite3 database file. If the file does not exist, it is created on the fly. It then writes the `MSFT` data to a table named `STOCK_DATA`. If the table does not exist, it is created as well. If it does exist, all the data is replaced with the `MSFT` data. Finally, it then appends the `AAPL` stock data to that table:

```
In [34]: # reference SQLite
import sqlite3

# read in the stock data from CSV
msft = pd.read_csv("data/msft.csv")
msft["Symbol"] = "MSFT"
aapl = pd.read_csv("data/aapl.csv")
aapl["Symbol"] = "AAPL"

# create connection
connection = sqlite3.connect("data/stocks.sqlite")
# .to_sql() will create SQL to store the DataFrame
# in the specified table. if_exists specifies
# what to do if the table already exists
msft.to_sql("STOCK_DATA", connection, if_exists="replace")
aapl.to_sql("STOCK_DATA", connection, if_exists="append")

# commit the SQL and close the connection
connection.commit()
connection.close()
```

To demonstrate that this data was created, you can open the database file with a tool such as SQLite Data Browser (available at <https://github.com/sqlitebrowser/sqlitebrowser>). The following screenshot shows you a few rows of the data in the database file:

The screenshot shows the DB Browser for SQLite interface. On the left, there's a table view for 'STOCK_DATA' with various filters applied. On the right, there's a modal window titled 'Edit Database Cell' where a cell is being edited. Below the table, there's a navigation bar with buttons for Go to: and a search field.

Data can be read using SQL from the database using the `pd.io.sql.read_sql()` function. The following demonstrates a query of the data from `stocks.sqlite` using SQL and reports it to the user:

```
In [35]: # connect to the database file
connection = sqlite3.connect("data/stocks.sqlite")

# query all records in STOCK_DATA
# returns a DataFrame
# inde_col specifies which column to make the DataFrame index
stocks = pd.io.sql.read_sql("SELECT * FROM STOCK_DATA;",
                           connection, index_col='index')

# close the connection
connection.close()

# report the head of the data retrieved
stocks[:5]
```

Out[35]:	Date	Open	High	Low	Close	Volume	Symbol	
	index							
	0	7/21/2014	83.46	83.53	81.81	81.93	2359300	MSFT
	1	7/18/2014	83.30	83.40	82.52	83.35	4020800	MSFT
	2	7/17/2014	84.35	84.63	83.33	83.63	1974000	MSFT
	3	7/16/2014	83.77	84.91	83.66	84.91	1755600	MSFT
	4	7/15/2014	84.30	84.38	83.20	83.58	1874700	MSFT

It is also possible to use the `WHERE` clause in the SQL as well as to select columns. To demonstrate, the following selects the records where MSFT's volume is greater than 29200100:

```
In [36]: # open the connection
connection = sqlite3.connect("data/stocks.sqlite")
# construct the query string
query = "SELECT * FROM STOCK_DATA WHERE " + \
        "Volume>29200100 AND Symbol='MSFT';"
# execute and close connection
items = pd.io.sql.read_sql(query, connection, index_col='index')
connection.close()
# report the query result
items
```

	Date	Open	High	Low	Close	Volume	Symbol
index							
1081	5/21/2010	42.22	42.35	40.99	42.00	33610800	MSFT
1097	4/29/2010	46.80	46.95	44.65	45.92	47076200	MSFT
1826	6/15/2007	89.80	92.10	89.55	92.04	30656400	MSFT
3455	3/16/2001	47.00	47.80	46.10	45.33	40806400	MSFT
3712	3/17/2000	49.50	50.00	48.29	50.00	50860500	MSFT

A final point is that most of the code in these examples was SQLite3 code. The only pandas part of these examples is the use of the `.to_sql()` and `.read_sql()` methods, as these functions take a connection object, which can be any Python DB-API-compatible data adapter, you can more or less work with any supported database data by simply creating an appropriate connection object. The code at the pandas level should remain the same for any supported database.

Reading data from remote data services

Pandas prior to 0.19.0 had direct support for various web-based data source classes in the `pandas.io.data` namespace. This has changed, as the functionality has been refactored out of pandas and into the `pandas-datareader` package.

This package provides access to many useful data sources, including:

- Daily historical stock prices from either Yahoo! or Google Finance
- Yahoo! and Google Options
- Enigma, a provider of structured data
- The Federal Reserve Economic Data Library
- Kenneth French's Data Library
- The World Bank
- OECD
- Eurostat
- EDGAR Index
- TSP Fund Data
- Oanda currency historical rates
- Nasdaq Trader symbol definitions



Note that because this data is coming from an external data source and that the actual values can change over time, it may be possible that when you run the code, you get different values than those in the book.

Reading stock data from Yahoo! and Google Finance

First, I want to say that unfortunately, Yahoo! has changed their API and currently this breaks the implementation in `pandas-datareader`. It is not known if this will be fixed. Therefore, the examples will only use Google Finance.

To use the `pandas-datareader` package, we use the following import:

```
In [37]: # import data reader package
import pandas_datareader as pdr
```

Stock data from Google Finance can then be objected using the `DataReader` function by passing it the stock symbol, the data source (in this case, '`google`'), and the start and end dates:

```
In [38]: # read from google and display the head of the data
start = datetime(2017, 4, 1)
end = datetime(2017, 4, 30)
goog = pdr.data.DataReader("MSFT", 'google', start, end)
goog[:5]
```

```
Out[38]:
```

Date	Open	High	Low	Close	Volume
2017-04-03	65.81	65.94	65.19	65.55	20400871
2017-04-04	65.39	65.81	65.28	65.73	12997449
2017-04-05	66.30	66.35	65.44	65.56	21448594
2017-04-06	65.60	66.06	65.48	65.73	18103453
2017-04-07	65.85	65.96	65.44	65.68	14108533

Retrieving options data from Google Finance

Pandas provides experimental support for Google Finance Options data to be retrieved via the `options` class. In the following example, the `.get_all_data()` method is used to download options data for `AAPL` from Google:

```
In [39]: # read options for MSFT
          options = pdr.data.Options('MSFT', 'google')
```

Google:

With this result, it is possible to determine the expiration dates by using the `.expiry_dates` property:

```
In [40]: options.expiry_dates
```

```
Out[40]: [datetime.date(2018, 1, 19)]
```

The actual data is read using `.get_options_data()`:

```
In [41]: data = options.get_options_data(expiry=options.expiry_dates[0])
          data.iloc[:5,:3]
```

```
Out[41]:
```

	Strike	Expiry	Type	Symbol	Last	Bid	Ask
23.0	2018-01-19	call	MSFT180119C00023000	45.34	45.55	48.85	
		put	MSFT180119P00023000	0.02	0.01	0.04	
25.0	2018-01-19	call	MSFT180119C00025000	43.25	43.80	46.70	
		put	MSFT180119P00025000	0.04	0.03	0.05	
28.0	2018-01-19	call	MSFT180119C00028000	41.55	42.00	42.80	

The resulting `DataFrame` object contains a hierarchical index which can be used to easily extract specific subsets of data. To demonstrate, let's examine several examples of slicing by the values using the index.

The following will return all the `put` options at a `strike` price of \$30. Using `slice(None)` as one of the values in the tuple used to select by index will include all `Expiry` dates:

```
In [42]: # get all puts at strike price of $30 (first four columns only)
          data.loc[(30, slice(None), 'put'), :].iloc[0:5, 0:3]
```

```
Out[42]:
```

	Strike	Expiry	Type	Symbol	Last	Bid	Ask
30.0	2018-01-19	put	MSFT180119P00030000	0.06	0.05	0.08	

We can narrow the date range by specifying a date slice instead of `slice(None)`. The following narrows the results down to those where the `Expiry` date is between `2015-01-17` and `2015-04-17`:

```
In [43]: # put options at strike of $80, between 2017-06-01 and 2017-06-30
          data.loc[(30, slice('20180119','20180130'), 'put'), :].iloc[:, 0:3]
```

```
Out[43]:
```

	Strike	Expiry	Type	Symbol	Last	Bid	Ask
30.0	2018-01-19	put	MSFT180119P00030000	0.06	0.05	0.08	

Reading economic data from the Federal Reserve Bank of St. Louis

The **Federal Reserve Economic Data (FRED)** of St. Louis (<http://research.stlouisfed.org/fred2/>) provides downloads of over 240,000 US and International time series from over 76 data sources, and it is constantly growing.

FRED data can be specified by using the `FredReader` class and by passing the specific series tag as the `name` parameter. As an example, the following retrieves GDP information between two specified dates:

```
In [44]: # read GDP data from FRED
gdp = pdr.data.FredReader("GDP",
                           date(2012, 1, 1),
                           date(2014, 1, 27))
gdp.read()[:5]
```

```
Out[44]:          GDP
DATE
2012-01-01  15973.9
2012-04-01  16121.9
2012-07-01  16227.9
2012-10-01  16297.3
2013-01-01  16475.4
```

To select another series, simply specify the series identifier in the first parameter. The site can be conveniently navigated through series and data visualized directly on the site. For example, the following screenshot shows you the series Compensation of employees: Wages and salaries:



★ Compensation of employees: Wages and salaries (A576RC1A027NBEA)

DOWNLOAD ↴

Observation:
2014: 7,477.8 (+
more)
Updated: Aug 5, 2015

Units:
Billions of Dollars,
Not Seasonally Adjusted

Frequency:
Annual

1Y | 5Y | 10Y |
Max
1929-01-01 to 2014-01-01

EDIT GRAPH ⚙

FRED Compensation of employees: Wages and salaries


Send feedback

This data series is represented by the A576RC1A027NBEA ID and we can download it with the following code:

```
In [45]: # Get Compensation of employees: Wages and salaries
pdr.data.FredReader("A576RC1A027NBEA",
                     date(1929, 1, 1),
                     date(2013, 1, 1)).read()[:5]
```

Out[45]: A576RC1A027NBEA

DATE	
1929-01-01	50.5
1930-01-01	46.2
1931-01-01	39.2
1932-01-01	30.5
1933-01-01	29.0

Accessing Kenneth French's data

Kenneth R. French is a professor of finance at the Tuck School of Business at Dartmouth University. He has created an extensive library of economic data that is available for download over the web. The website for his data is http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html, and it contains a detailed explanation of the data sets.

The data available at the site is downloadable in ZIP files and can be read directly into DataFrame by specifying the dataset's filename (without .zip) and using the `FamaFrenchReader` function. As an example, the following reads the Global Factors data:

```
In [46]: # read from Kenneth French fama global factors data set
factors = pdr.data.FamaFrenchReader("Global_Factors").read()
factors[0][:5]
```

```
Out[46]:
```

	Mkt-RF	SMB	HML	WML	RF
Date					
2010-01	-3.70	2.70	-0.29	-2.23	0.00
2010-02	1.24	0.14	0.10	1.59	0.00
2010-03	6.30	-0.26	3.18	4.26	0.01
2010-04	0.44	3.78	0.77	1.60	0.01
2010-05	-9.52	0.17	-2.54	-0.56	0.01

Reading from the World Bank

Thousands of data feeds are available from the World Bank and can be read directly into pandas `DataFrame` objects. The World Bank data catalog can be explored at <http://www.worldbank.org/>.

World Bank datasets are identified using indicators, a text code that represents each dataset. A full list of indicators can be retrieved using the `pandas.io.wb.get_indicators()` function. At the time of writing, there were 16,167 indicators. The following retrieves the indicators and displays the first five:

```
In [47]: # get all indicators
from pandas_datareader import wb
all_indicators = pdr.wb.get_indicators()
all_indicators.iloc[:5,:2]
```

	id	name
0	1.0.HCount.1.90usd	Poverty Headcount (\$1.90 a day)
1	1.0.HCount.2.5usd	Poverty Headcount (\$2.50 a day)
2	1.0.HCount.Mid10to50	Middle Class (\$10-50 a day) Headcount
3	1.0.HCount.Ofcl	Official Moderate Poverty Rate-National
4	1.0.HCount.Poor4uds	Poverty Headcount (\$4 a day)

These indicators can be investigated using the World Bank website, but if you have an idea of the indicator you would like to sample, you can simply perform a search. As an example, the following uses the `wb.search()` function to search for indicators with data related to life expectancy:

```
In [48]: # search of life expectancy indicators
le_indicators = pdr.wb.search("life expectancy")
# report first three rows, first two columns
le_indicators.iloc[:5,:2]
```

	id	name
8413	SE.SCH.LIFE	School life expectancy, primary to tertiary, b...
9626	SP.DYN.LE00.FE.IN	Life expectancy at birth, female (years)
9627	SP.DYN.LE00.IN	Life expectancy at birth, total (years)
9628	SP.DYN.LE00.MA.IN	Life expectancy at birth, male (years)
9629	SP.DYN.LE60.FE.IN	Life expectancy at age 60, female

Each indicator is broken down into various countries. A full list of country data can be retrieved using the `wb.get_countries()` function:

```
In [49]: # get countries and show the 3 digit code and name
countries = pdr.wb.get_countries()
# show a subset of the country data
countries.loc[0:5,['name', 'capitalCity', 'iso2c']]
```

```
Out[49]:      name    capitalCity iso2c
0     Aruba        Oranjestad    AW
1  Afghanistan         Kabul    AF
2     Africa
3     Angola        Luanda    AO
4    Albania        Tirane    AL
5   Andorra  Andorra la Vella    AD
```

Data for an indicator can be downloaded using the `wb.download()` function and by specifying the dataset using the `indicator` parameter. The following downloads the life expectancy data for countries from 1980 through 2014:

```
In [50]: # get life expectancy at birth for all countries from 1980 to 2014
le_data_all = pdr.wb.download(indicator="SP.DYN.LE00.IN",
                               start='1980',
                               end='2014')
le_data_all
```

```
Out[50]:          SP.DYN.LE00.IN
country      year
Canada       2014    81.956610
              2013    81.765049
              2012    81.562439
              2011    81.448780
              2010    81.197561
...
United States  1984    74.563415
              1983    74.463415
              1982    74.360976
              1981    74.009756
              1980    73.609756
```

[105 rows x 1 columns]

By default, the data is only returned for the United States, Canada, and Mexico. This can be seen by examining the index of the results of the previous query:

```
In [51]: # only US, CAN, and MEX are returned by default
le_data_all.index.levels[0]
```

```
Out[51]: Index(['Canada', 'Mexico', 'United States'], dtype='object', name='country')
```

To get data for more countries, specify them explicitly using the `country` parameter. The following gets the

data for all known countries:

```
In [52]: # retrieve life expectancy at birth for all countries
# from 1980 to 2014
le_data_all = wb.download(indicator="SP.DYN.LE00.IN",
                           country = countries['iso2c'],
                           start='1980',
                           end='2012')
le_data_all
```

```
Out[52]:          SP.DYN.LE00.IN
    country   year
    Aruba     2012    75.205756
              2011    75.081390
              2010    74.953537
              2009    74.818146
              2008    74.675732
...
    ...
    Zimbabwe 1984    61.583951
              1983    61.148171
              1982    60.605512
              1981    60.004829
              1980    59.388024
```

[8679 rows x 1 columns]

We can do some interesting things with this data. The example we will look at determines which country has the lowest life expectancy for each year. To do this, we first need to pivot this data so that the index is the country name and the year is the column. We will look at pivoting in more detail in later chapters, but for now, just know that the following reorganizes the data into the country along the index and the year across the columns. Also, each value is the life expectancy for each country for that specific year:

```
In [53]: #le_data_all.pivot(index='country', columns='year')
le_data = le_data_all.reset_index().pivot(index='country',
                                           columns='year')

# examine pivoted data
le_data.iloc[:5,0:3]
```

```
Out[53]:          SP.DYN.LE00.IN
    year        1980        1981        1982
    country
    Afghanistan  41.867537  42.526927  43.230732
    Albania      70.235976  70.454463  70.685122
    Algeria      58.164024  59.486756  60.786341
    American Samoa       NaN       NaN       NaN
    Andorra       NaN       NaN       NaN
```

With the data in this format, we can determine which country has the lowest life expectancy for each year using `.idxmin(axis=0)`:

```
In [54]: # ask what is the name of country for each year  
# with the least life expectancy  
country_with_least_expectancy = le_data.idxmin(axis=0)  
country_with_least_expectancy[:5]
```

```
Out[54]:      year  
SP.DYN.LE00.IN 1980      Cambodia  
                1981      Cambodia  
                1982    Timor-Leste  
                1983   South Sudan  
                1984   South Sudan  
dtype: object
```

The actual minimum value for each year can be retrieved using `.min(axis=0)`:

```
In [55]: # and what is the minimum life expectancy for each year  
expectancy_for_least_country = le_data.min(axis=0)  
expectancy_for_least_country[:5]
```

```
Out[55]:      year  
SP.DYN.LE00.IN 1980      27.738976  
                1981      33.449927  
                1982      38.186220  
                1983      39.666488  
                1984      39.999537  
dtype: float64
```

These two results can then be combined into a new `DataFrame` that tells us which country has had the least life expectancy for each year and what that value is:

Summary

In this chapter, we examined how pandas makes it simple to access data in various locations and formats, providing automatic mapping of data in these formats into DataFrame objects. We started with learning how to read and write data from local files in CSV, HTML, JSON, HDF5, and Excel formats, reading into, and writing directly from DataFrame objects without having to worry about the details of mapping the contained data into these various formats.

We then examined how to access data from remote sources. First, we saw that the functions and methods that work with local files can also read from web and cloud data sources. We then looked at pandas support for accessing various forms of web and web-service-based data, such as Yahoo! Finance and the World Bank.

Now that we are able to load the data, the next step in using it is to perform the cleaning of the data, because it is often the case that the retrieved data has issues such as missing information and ill-formed content. The next chapter will focus on these types of issues in a process commonly known as **tidying your data**.

Tidying Up Your Data

We are at that point in the data processing pipeline where we need to look at the data that we have retrieved and address any anomalies that may present themselves during analysis. These anomalies can exist for a multitude of reasons. Sometimes, certain parts of the data are not recorded or perhaps get lost. Maybe there are units that don't match your system's units. Many times, certain data points can be duplicated.

This process of dealing with anomalous data is often referred to as **tidying** your data, and you will see this term used many times in data analysis. This is a very important step in the pipeline, and it can consume much of your time before you even get to working on simple analyses.

Tidying of data can be a tedious problem, particularly when using programming tools that are not designed for the specific task of data cleanup. Fortunately for us, pandas has many tools that can be used to address these issues, and also help us be very efficient at the same time.

In this chapter, we will cover many of the tasks involved in tidying data. Specifically, you will learn:

- The concept of tidy data
- How to work with missing data
- How to find `NaN` values in data
- How to filter (drop) missing data
- How pandas handles missing values in calculations
- How to find, filter, and fix unknown values
- Performing interpolation of missing values
- How to identify and remove duplicate data
- How to transform values using `replace`, `map`, and `apply`

Configuring pandas

The examples in this chapter utilize the following configuration of pandas and Jupyter:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set some pandas options controlling output format
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```


What is tidying your data?

Tidy data is a term that was coined in a paper named "Tidy Data" by Hadley Wickham. I highly recommend that you read this paper. It can be downloaded from <http://vita.had.co.nz/papers/tidy-data.pdf>.

The paper covers many details of the process of creating tidy data, the end result of which is that you have data that is free of surprises and is ready for analysis.

We will examine many of the tools in pandas for tidying your data. These exist because we need to handle the following situations:

- The names of the variables are different from what you require
- There is *missing* data
- Values are not in the units that you require
- The period of sampling of records is not what you need
- Variables are categorical and you need quantitative values
- There is *noise* in the data
- Information is of an incorrect type
- Data is organized around incorrect axes
- Data is at the wrong level of normalization
- Data is duplicated

This is quite a list, and I assure you it is not complete. But these are all issues that I have personally come across (and I'm sure you will too). When using tools and languages not explicitly built for handling these issues (like pandas is), they have been quite difficult to resolve. We will examine how easy it is to fix these problems with pandas during this chapter.

How to work with missing data

Data is **missing** in pandas when it has a value of `NaN` (also seen as `np.nan` - the form from NumPy). This `NaN` value means that there is no value specified for the particular index label in a particular series.

How can data be missing? There are a number of reasons why a value can be `NaN`:

- A join of two sets of data does not have matched values
- Data that you retrieved from an external source is incomplete
- The `NaN` value is not known at a given point in time and will be filled in later
- There is a data collection error retrieving a value, but the event must still be recorded in the index
- Reindexing of data has resulted in an index that does not have a value
- The shape of data has changed and there are now additional rows or columns, which at the time of reshaping could not be determined
- There are likely more reasons, but the general point is that these situations do occur and you, as a user of pandas, will need to address these situations to be able to perform effective data analysis

Let's start an examination of how to work with missing data by creating a DataFrame that has some

```
In [2]: # create a DataFrame with 5 rows and 3 columns
df = pd.DataFrame(np.arange(0, 15).reshape(5, 3),
                  index=['a', 'b', 'c', 'd', 'e'],
                  columns=['c1', 'c2', 'c3'])
df

Out[2]:   c1  c2  c3
a    0   1   2
b    3   4   5
c    6   7   8
d    9  10  11
e   12  13  14
```

missing data points:

There is here is not any missing data so let's add some:

```
In [3]: # add some columns and rows to the DataFrame
# column c4 with NaN values
df['c4'] = np.nan
# row 'f' with 15 through 18
df.loc['f'] = np.arange(15, 19)
# row 'g' will all NaN
df.loc['g'] = np.nan
# column 'C5' with NaN's
df['c5'] = np.nan
# change value in col 'c4' row 'a'
df['c4']['a'] = 20
df
```

```
Out[3]:   c1  c2  c3  c4  c5
a  0.0  1.0  2.0  20.0  NaN
b  3.0  4.0  5.0  NaN  NaN
c  6.0  7.0  8.0  NaN  NaN
d  9.0  10.0 11.0  NaN  NaN
e 12.0  13.0 14.0  NaN  NaN
f 15.0  16.0 17.0  18.0  NaN
g  NaN  NaN  NaN  NaN  NaN
```

This DataFrame now has missing data which exhibits the following characteristics:

- One row consisting only of `NaN` values
- One column consisting only of `NaN` values

- Several rows and columns consisting of both numeric values and `NaN` values
- Now let's examine various techniques to work with this missing data.

Determining NaN values in pandas objects

The `NaN` values in a `DataFrame` object can be identified using the `.isnull()` method. Any `True` value means that the item at that location is a `NaN` value:

```
In [4]: # which items are NaN?  
df.isnull()
```

```
Out[4]:   c1    c2    c3    c4    c5  
a  False  False  False  False  True  
b  False  False  False  True  True  
c  False  False  False  True  True  
d  False  False  False  True  True  
e  False  False  False  True  True  
f  False  False  False  False  True  
g  True   True   True   True  True
```

We can use the fact that the `.sum()` method treats `True` as 1 and `False` as 0 to determine the number of `NaN` values in a `DataFrame` object:

```
In [5]: # count the number of NaN's in each column  
df.isnull().sum()
```

```
Out[5]: c1    1  
c2    1  
c3    1  
c4    5  
c5    7  
dtype: int64
```

Applying `.sum()` to the resulting series gives the total number of `NaN` values in the original `DataFrame` object:

```
In [6]: # total count of NaN values  
df.isnull().sum().sum()
```

```
Out[6]: 15
```

This is great, as it can be an easy way to identify if data is missing early in a process. If you are expecting your data to be complete, and this simple check turns up a value other than 0, then you know you need to look deeper.

Another way to determine this is to use the `.count()` method of a `Series` object and `DataFrame`. For a `Series` method, this method will return the number of non-`NaN` values. For a `DataFrame` object, it will count the number of non-`NaN` values in each column:

```
In [7]: # number of non-NaN values in each column  
df.count()
```

```
Out[7]: c1    6  
        c2    6  
        c3    6  
        c4    2  
        c5    0  
       dtype: int64
```

This then needs to be flipped around to sum the number of `NaN` values, which can be calculated as follows:

```
In [8]: # and this counts the number of NaN's too  
(len(df) - df.count()).sum()
```

```
Out[8]: 15
```

We can also determine whether an item is not `NaN` using the `.notnull()` method, which returns `True` if the value is not a `NaN` value; otherwise, it returns `False`:

```
In [9]: # which items are not null?  
df.notnull()
```

```
Out[9]:      c1      c2      c3      c4      c5  
a    True    True    True    True   False  
b    True    True    True   False   False  
c    True    True    True   False   False  
d    True    True    True   False   False  
e    True    True    True   False   False  
f    True    True    True    True   False  
g  False   False   False   False   False
```


Selecting out or dropping missing data

One technique of handling missing data is to simply remove it from your dataset. A scenario for this would be where data is sampled at regular intervals but devices are offline and hence a reading is not recorded.

The pandas library makes this possible using several techniques. One is through Boolean selection using the results of `.isnull()` or `.notnull()` to retrieve the values that are `NaN` or non `NaN` out of a `Series` object. The following example demonstrates selection of all non-`NaN` values from the `c4` column of a `DataFrame`:

```
In [10]: # select the non-NaN items in column c4  
df.c4[df.c4.notnull()]
```

```
Out[10]: a    20.0  
          f    18.0  
          Name: c4, dtype: float64
```

Pandas also provides a convenience function, `.dropna()`, which drops the items in a `Series` where the value is `NaN`:

```
In [11]: # .dropna will also return non NaN values  
# this gets all non NaN items in column c4  
df.c4.dropna()
```

```
Out[11]: a    20.0  
          f    18.0  
          Name: c4, dtype: float64
```

Note that `.dropna()` has actually returned a copy of `DataFrame` without the rows. The original `DataFrame` is not changed:

```
In [12]: # dropna returns a copy with the values dropped  
# the source DataFrame / column is not changed  
df.c4
```

```
Out[12]: a    20.0  
          b    NaN  
          c    NaN  
          d    NaN  
          e    NaN  
          f    18.0  
          g    NaN  
          Name: c4, dtype: float64
```

When `.dropna()` is applied to a `DataFrame` object, it drops all rows from the `DataFrame` object that have at least one `NaN` value. The following code demonstrates this in action, and since each row has at least one `NaN` value, there are zero rows in the result:

```
In [13]: # on a DataFrame this will drop entire rows
# where there is at least one NaN
# in this case, that is all rows
df.dropna()
```

```
Out[13]: Empty DataFrame
Columns: [c1, c2, c3, c4, c5]
Index: []
```

If you want to drop only those rows where all values are `NaN`, you can use the `how='all'` parameter. The following sample drops only the `g` row, since it has all `NaN` values:

```
In [14]: # using how='all', only rows that have all values
# as NaN will be dropped
df.dropna(how = 'all')
```

```
Out[14]:      c1    c2    c3    c4    c5
a    0.0    1.0    2.0   20.0  NaN
b    3.0    4.0    5.0    NaN  NaN
c    6.0    7.0    8.0    NaN  NaN
d    9.0   10.0   11.0    NaN  NaN
e   12.0   13.0   14.0    NaN  NaN
f   15.0   16.0   17.0   18.0  NaN
```

This can also be applied to the columns instead of the rows, by changing the `axis` parameter to `axis=1`. The following drops the `c5` column, as it is the only one with all `NaN` values:

```
In [15]: # flip to drop columns instead of rows
df.dropna(how='all', axis=1) # say goodbye to c5
```

```
Out[15]:      c1    c2    c3    c4
a    0.0    1.0    2.0   20.0
b    3.0    4.0    5.0    NaN
c    6.0    7.0    8.0    NaN
d    9.0   10.0   11.0    NaN
e   12.0   13.0   14.0    NaN
f   15.0   16.0   17.0   18.0
g    NaN    NaN    NaN    NaN
```

Now let's examine this process using a slightly different `DataFrame` object, which has columns `c1` and `c3` with all values that are not `NaN`. In this case, all columns except `c1` and `c3` will be dropped:

```
In [16]: # make a copy of df
df2 = df.copy()
# replace two NaN cells with values
df2.loc['g'].c1 = 0
df2.loc['g'].c3 = 0
df2
```

```
Out[16]:   c1    c2    c3    c4    c5
a    0.0    1.0    2.0   20.0  NaN
b    3.0    4.0    5.0  NaN  NaN
c    6.0    7.0    8.0  NaN  NaN
d    9.0   10.0   11.0  NaN  NaN
e   12.0   13.0   14.0  NaN  NaN
f   15.0   16.0   17.0   18.0  NaN
g    0.0    NaN    0.0  NaN  NaN
```

```
In [17]: # now drop columns with any NaN values
df2.dropna(how='any', axis=1)
```

```
Out[17]:   c1    c3
a    0.0    2.0
b    3.0    5.0
c    6.0    8.0
d    9.0   11.0
e   12.0   14.0
f   15.0   17.0
g    0.0    0.0
```

The `.dropna()` method also has a parameter `thresh`, which when given an integer value, specifies the minimum number of `NaN` values that must exist before the drop is performed. The following code drops all the columns with at least five `NaN` values (in this case, these are the `c4` and `c5` columns):

```
In [18]: # only drop columns with at least 5 NaN values
df.dropna(thresh=5, axis=1)
```

```
Out[18]:   c1    c2    c3
a    0.0    1.0    2.0
b    3.0    4.0    5.0
c    6.0    7.0    8.0
d    9.0   10.0   11.0
e   12.0   13.0   14.0
f   15.0   16.0   17.0
g    NaN    NaN    NaN
```

Again, note that the `.dropna()` method (and the Boolean selection) returns a copy of the `DataFrame` object, and the data is dropped from that copy. If you want to drop the data in the actual `DataFrame`, use the `inplace=True` parameter.

Handling of NaN values in mathematical operations

The `NaN` values are handled differently in pandas than in NumPy. We have seen this in earlier chapters, but it is worth revisiting here. This is demonstrated using the following example:

```
In [19]: # create a NumPy array with one NaN value
a = np.array([1, 2, np.nan, 3])
# create a Series from the array
s = pd.Series(a)
# the mean of each is different
a.mean(), s.mean()
```

```
Out[19]: (nan, 2.0)
```

When a NumPy function encounters a `NaN` value, it returns `NaN`. Pandas functions typically ignore the `NaN` values and continue processing the function as though the `NaN` values were not part of the `Series` object.

 *Note that the mean of the preceding series was calculated as $(1+2+3)/3 = 2$, not $(1+2+3)/4$ or $(1+2+0+4)/4$. This verifies that `NaN` is totally ignored and not even counted as an item in `Series`.*

More specifically, the way that Pandas handles the `NaN` values is as follows:

- Summing of data treats `NaN` as 0
- If all values are `NaN`, the result is `NaN`
- Methods like `.cumsum()` and `.cumprod()` ignore the `NaN` values, but preserve them in the resulting arrays

The following demonstrates all these concepts:

```
In [20]: # demonstrate sum, mean and cumsum handling of NaN
# get one column
s = df.c4
s.sum(), # NaN's treated as 0
```

```
Out[20]: (38.0,)
```

```
In [21]: s.mean() # NaN also treated as 0
```

```
Out[21]: 19.0
```

```
In [22]: # as 0 in the cumsum, but NaN's preserved in result Series  
s.cumsum()
```

```
Out[22]: a    20.0  
         b    NaN  
         c    NaN  
         d    NaN  
         e    NaN  
         f    38.0  
         g    NaN  
Name: c4, dtype: float64
```

But when using traditional mathematical operators, `NaN` will be propagated through to the result:

```
In [23]: # in arithmetic, a NaN value will result in NaN  
df.c4 + 1
```

```
Out[23]: a    21.0  
         b    NaN  
         c    NaN  
         d    NaN  
         e    NaN  
         f    19.0  
         g    NaN  
Name: c4, dtype: float64
```


Filling in missing data

The `.fillna()` method can be used to replace the `NaN` values with a specific value, instead of having them propagated or flat out ignored. The following demonstrates this by filling the `NaN` values with `0`:

```
In [24]: # return a new DataFrame with NaN's filled with 0
filled = df.fillna(0)
filled
```

```
Out[24]:   c1    c2    c3    c4    c5
a    0.0    1.0    2.0   20.0   0.0
b    3.0    4.0    5.0    0.0   0.0
c    6.0    7.0    8.0    0.0   0.0
d    9.0   10.0   11.0    0.0   0.0
e   12.0   13.0   14.0    0.0   0.0
f   15.0   16.0   17.0   18.0   0.0
g    0.0    0.0    0.0    0.0   0.0
```

Be aware that this causes differences in the resulting values. As an example, the following code shows the result of applying the `.mean()` method to a `DataFrame` object with the `NaN` values, as compared to a `DataFrame` object that has its `NaN` values filled with `0`:

```
In [25]: # NaN's don't count as an item in calculating
# the means
df.mean()
```

```
Out[25]: c1    7.5
          c2    8.5
          c3    9.5
          c4   19.0
          c5    NaN
          dtype: float64
```

```
In [26]: # having replaced NaN with 0 can make
# operations such as mean have different results
filled.mean()
```

```
Out[26]: c1    6.428571
          c2    7.285714
          c3    8.142857
          c4    5.428571
          c5    0.000000
          dtype: float64
```


Forward and backward filling of missing values

Gaps in data can be filled by propagating the non-`NaN` values forward or backward along a `Series`. To demonstrate, the following example will *fill forward* the `c4` column of `DataFrame`:



When working with time series data, this technique of filling is often referred to as the "last known value". We will revisit this in the chapter on time-series data.

The direction of the fill can be reversed using `method='bfill'`:

```
In [28]: # perform a backwards fill  
df.c4.fillna(method="bfill")
```

```
Out[28]: a    20.0  
         b    18.0  
         c    18.0  
         d    18.0  
         e    18.0  
         f    18.0  
         g    NaN  
Name: c4, dtype: float64
```

To save a little typing, pandas also has global level functions `pd.ffill()` and `pd.bfill()`, which are equivalent to `.fillna(method="ffill")` and `.fillna(method="bfill")`.

Filling using index labels

Data can be filled using the labels of a `Series` or keys of a Python dictionary. This allows you to specify different fill values for different elements, based upon the value of the index label:

```
In [29]: # create a new Series of values to be
# used to fill NaN's where index label matches
fill_values = pd.Series([100, 101, 102], index=['a', 'e', 'g'])
fill_values
```

```
Out[29]: a    100
          e    101
          g    102
          dtype: int64
```

```
In [30]: # using c4, fill using fill_values
# a, e and g will be filled with matching values
df.c4.fillna(fill_values)
```

```
Out[30]: a    20.0
          b    NaN
          c    NaN
          d    NaN
          e    101.0
          f    18.0
          g    102.0
          Name: c4, dtype: float64
```

Only values of `NaN` are filled. Note that the values with label `a` are not changed.

Another common scenario is to fill all the `NaN` values in a column with the mean of the column:

```
In [31]: # fill NaN values in each column with the
# mean of the values in that column
df.fillna(df.mean())
```

```
Out[31]:      c1    c2    c3    c4    c5
          a    0.0   1.0   2.0  20.0  NaN
          b    3.0   4.0   5.0  19.0  NaN
          c    6.0   7.0   8.0  19.0  NaN
          d    9.0  10.0  11.0  19.0  NaN
          e   12.0  13.0  14.0  19.0  NaN
          f   15.0  16.0  17.0  18.0  NaN
          g    7.5   8.5   9.5  19.0  NaN
```

This is convenient, as missing values replaced in this manner skew the statistical average less than if 0's are inserted. In certain statistical analyses, this may be acceptable where the larger deviation using 0 values can cause false failures.

Performing interpolation of missing values

Both `DataFrame` and `Series` have an `.interpolate()` method that, by default, performs a linear interpolation of missing values:

```
In [32]: # linear interpolate the NaN values from 1 through 2
s = pd.Series([1, np.nan, np.nan, np.nan, 2])
s.interpolate()
```

```
Out[32]: 0    1.00
1    1.25
2    1.50
3    1.75
4    2.00
dtype: float64
```

The value of the interpolation is calculated by taking the first value before and after any sequence of the `NaN` values, and then incrementally adding that value from the start and substituting the `NaN` values. In this case, 2.0 and 1.0 are the surrounding values, resulting in $(2.0 - 1.0)/(5-1) = 0.25$, which is then added incrementally through all the `NaN` values.

This is important. Imagine if your data represents an increasing set of values, such as increasing temperature during the day. If the sensor stops responding for a few sample periods, the missing data can be inferred through interpolation with a high level of certainty. It is definitely better than setting the values to 0.

The interpolation method also has the ability to specify a specific method of interpolation. One of the common methods is to use time-based interpolation. Consider the following `Series` of dates and values:

```
In [33]: # create a time series, but missing one date in the Series
ts = pd.Series([1, np.nan, 2],
               index=[datetime(2014, 1, 1),
                      datetime(2014, 2, 1),
                      datetime(2014, 4, 1)])
ts
```

```
Out[33]: 2014-01-01    1.0
2014-02-01    NaN
2014-04-01    2.0
dtype: float64
```

The previous form of interpolation results in the following:

```
In [34]: # linear interpolate based on number of items in the Series
ts.interpolate()

Out[34]: 2014-01-01    1.0
          2014-02-01    1.5
          2014-04-01    2.0
dtype: float64
```

The value for 2014-02-01 is calculated as $1.0 + (2.0 - 1.0)/2 = 1.5$, since there is one `NaN` value between the values 2.0 and 1.0.

The important thing to note is that the series is missing an entry for 2014-03-01. If we were expecting to interpolate daily values, there should be two values calculated, one for 2014-02-01 and another for 2014-03-01, resulting in one more value in the numerator of the interpolation.

This can be corrected by specifying the method of interpolation as `time`:

```
In [35]: # this accounts for the fact that we don't have
# an entry for 2014-03-01
ts.interpolate(method="time")

Out[35]: 2014-01-01    1.000000
          2014-02-01    1.344444
          2014-04-01    2.000000
dtype: float64
```

This is the correct interpolation for 2014-02-01, based upon dates. Also note that the index label and value for 2014-03-01 is not added to `series`; it is just factored into the interpolation.

Interpolation can also be specified to calculate values relative to the index values when using numeric index labels. To demonstrate this, we will use the following `Series`:

```
In [36]: # a Series to demonstrate index label based interpolation
s = pd.Series([0, np.nan, 100], index=[0, 1, 10])
s

Out[36]: 0      0.0
          1      NaN
          10     100.0
dtype: float64
```

If we perform a linear interpolation, we get the following value for label 1, which is correct for a linear interpolation:

```
In [37]: # linear interpolate
s.interpolate()

Out[37]: 0      0.0
          1      50.0
          10     100.0
dtype: float64
```

However, what if we want to interpolate the value to be relative to the index value? To do this, we can use `method="values"`:

```
In [38]: # interpolate based upon the values in the index  
s.interpolate(method="values")
```

```
Out[38]: 0      0.0  
1      10.0  
10     100.0  
dtype: float64
```

The value calculated for `NaN` is now interpolated using relative positioning, based on the labels in the index. The `NaN` value has a label of `1`, which is one tenth of the way between `0` and `10`, so the interpolated value will be $0 + (100-0)/10$, or `10`.

Handling duplicate data

The data in your sample can often contain duplicate rows. This is just a reality of dealing with data that is collected automatically, or even a situation created when manually collecting data. In these situations, it is often considered best to error on the side of having duplicates instead of missing data, especially if the data can be considered to be idempotent. However, duplicate data can increase the size of the dataset, and if it is not idempotent, then it would not be appropriate to process the duplicates.

Pandas provides the `.duplicates()` method to facilitate finding duplicate data. This method returns a Boolean `Series`, where each entry represents whether or not the row is a duplicate. A `True` value represents that the specific row has appeared earlier in the `DataFrame` object, with all the column values identical.

The following demonstrates this in action by creating a `DataFrame` object with duplicate rows:

```
In [39]: # a DataFrame with lots of duplicate data
data = pd.DataFrame({'a': ['x'] * 3 + ['y'] * 4,
                     'b': [1, 1, 2, 3, 3, 4, 4]})
data
```

```
Out[39]:   a   b
0   x   1
1   x   1
2   x   2
3   y   3
4   y   3
5   y   4
6   y   4
```

Now let's check for duplicates:

```
In [40]: # reports which rows are duplicates based upon
# if the data in all columns was seen before
data.duplicated()
```

```
Out[40]: 0    False
1     True
2    False
3    False
4     True
5    False
6     True
dtype: bool
```

Duplicate rows can be dropped from a `DataFrame` by using the `.drop_duplicates()` method. This method returns a copy of the `DataFrame` with the duplicate rows removed:

```
In [41]: # drop duplicate rows retaining first row of the duplicates  
data.drop_duplicates()
```

```
Out[41]:   a   b  
0   x   1  
2   x   2  
3   y   3  
5   y   4
```

It is also possible to use the `inplace=True` parameter to remove the rows without making a copy.

Note that there is a ramification to which indexes remain when dropping duplicates. The duplicate records may have different index labels (labels are not taken into account in calculating a duplicate). So, the row that is kept can affect the set of labels in the resulting `DataFrame` object.

The default operation is to keep the first row of the duplicates. If you want to keep the last row of the duplicates, use the `keep='last'` parameter.

The following demonstrates how the result differs using this parameter:

```
In [42]: # drop duplicate rows, only keeping the first  
# instance of any data  
data.drop_duplicates(keep='last')
```

```
Out[42]:   a   b  
1   x   1  
2   x   2  
4   y   3  
6   y   4
```

If you want to check for duplicates based on a smaller set of columns, you can specify a list of column names:

```
In [43]: # add a column c with values 0..6  
# this makes .duplicated() report no duplicate rows  
data['c'] = range(7)  
data.duplicated()
```

```
Out[43]: 0    False  
1    False  
2    False  
3    False  
4    False  
5    False  
6    False  
dtype: bool
```

```
In [44]: # but if we specify duplicates to be dropped only in columns a & b  
# they will be dropped  
data.drop_duplicates(['a', 'b'])
```

```
Out[44]:   a   b   c  
0  x   1   0  
2  x   2   2  
3  y   3   3  
5  y   4   5
```

Transforming data

Another part of tidying data involves transforming the existing data into another presentation. This may be needed for the following reasons:

- Values are not in the correct units
- Values are qualitative and need to be converted to appropriate numeric values
- There is extraneous data that either wastes memory and processing time, or can affect results simply by being included

To address these situations, we can take one or more of the following actions:

- Map values to other values using a table lookup process
- Explicitly replace certain values with other values (or even another type of data)
- Apply methods to transform the values based on an algorithm
- Simply remove extraneous columns and rows

We have already seen how to delete rows and columns with several techniques, so we will not reiterate those here. Now we will cover the facilities provided by pandas for mapping, replacing, and applying functions to transform data based upon its content.

Mapping data into different values

One of the basic tasks in data transformations is the mapping of a set of values to another set. Pandas provides a generic ability to map values using a lookup table (via a Python dictionary or a pandas `Series`) using the `.map()` method.

This method performs the mapping by first matching the values of the outer `Series` with the index labels of the inner `Series`. It then returns a new `Series`, with the index labels of the outer `Series` but the values from the inner `Series`.

The following example shows how to map the labels in the index of `x` to the values of `y`:

```
In [45]: # create two Series objects to demonstrate mapping
x = pd.Series({"one": 1, "two": 2, "three": 3})
y = pd.Series({1: "a", 2: "b", 3: "c"})
x
```

```
Out[45]: one      1
         three    3
         two      2
        dtype: int64
```

```
In [46]: y
```

```
Out[46]: 1      a
         2      b
         3      c
        dtype: object
```

```
In [47]: # map values in x to values in y
x.map(y)
```

```
Out[47]: one      a
         three    NaN
         two      b
        dtype: object
```

As with other alignment operations, if pandas does not find a map between the value of the outer `Series` and an index label of the inner `Series`, it fills the value with `NaN`. To demonstrate, the following removes the `3` key from the outer `Series`, which causes the alignment to fail for that record, and results in a `NaN` value being introduced:

```
In [48]: # three in x will not align / map to a value in y
x = pd.Series({"one": 1, "two": 2, "three": 3})
y = pd.Series({1: "a", 2: "b"})
x.map(y)
```

```
Out[48]: one      a
         three    NaN
         two      b
        dtype: object
```

Replacing values

We previously saw how the `.fillna()` method can be used to replace the `NaN` values with a value of your own decision. The `.fillna()` method can actually be thought of as an implementation of the `.map()` method that maps a single value, `NaN`, to a specific value.

Even more generically, the `.fillna()` method itself can be considered a specialization of a more general replacement that is provided by the `.replace()` method. This method provides more flexibility by being able to replace any value (not just `NaN`) with another value.

The most basic use of the `.replace()` method is to replace an individual value with another:

```
In [49]: # create a Series to demonstrate replace
s = pd.Series([0., 1., 2., 3., 2., 4.])
s
```

```
Out[49]: 0    0.0
1    1.0
2    2.0
3    3.0
4    2.0
5    4.0
dtype: float64
```

```
In [50]: # replace all items with index label 2 with value 5
s.replace(2, 5)
```

```
Out[50]: 0    0.0
1    1.0
2    5.0
3    3.0
4    5.0
5    4.0
dtype: float64
```

It is also possible to specify multiple items to replace, and also to specify their substitute values by passing two lists (the first of the values to replace, and the second, the replacements):

```
In [51]: # replace all items with new values  
s.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
```

```
Out[51]: 0    4.0  
1    3.0  
2    2.0  
3    1.0  
4    2.0  
5    0.0  
dtype: float64
```

Replacement can also be performed by specifying a dictionary for lookup (a variant of the map process in the previous section):

```
In [52]: # replace using entries in a dictionary  
s.replace({0: 10, 1: 100})
```

```
Out[52]: 0    10.0  
1    100.0  
2    2.0  
3    3.0  
4    2.0  
5    4.0  
dtype: float64
```

If using `.replace()` on a `DataFrame`, it is possible to specify different replacement values for each column. This is performed by passing a Python dictionary to the `.replace()` method. In this dictionary, the keys represent the names of the columns where replacement is to occur, and the values of the dictionary specify where the replacement is to occur. The second parameter to the method is the value that is used to replace where there are matches.

The following code demonstrates this by creating a `DataFrame` object and then replacing specific values in each of the columns with 100:

```
In [53]: # DataFrame with two columns  
df = pd.DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})  
df
```

```
Out[53]:   a  b  
0  0  5  
1  1  6  
2  2  7  
3  3  8  
4  4  9
```

```
In [54]: # specify different replacement values for each column  
df.replace({'a': 1, 'b': 8}, 100)
```

```
Out[54]:      a    b  
0      0    5  
1    100    6  
2      2    7  
3      3  100  
4      4    9
```

Replacing specific values in each of the columns is very convenient, as it provides a shorthand for what otherwise would require coding a loop through all the columns.

It is also possible to replace items at specific index positions, as though they are missing values. The following code demonstrates this by forward filling the value at index position 0 into locations 1, 2, and 3:

```
In [56]: # replace items with index label 1, 2, 3, using fill from the  
# most recent value prior to the specified labels (10)  
s.replace([1, 2, 3], method='pad')
```

```
Out[56]: 0    10.0  
1    10.0  
2    10.0  
3    10.0  
4    10.0  
5     4.0  
dtype: float64
```



It is also possible to forward and backwards fill by using `ffill` and `bfill` as the specified methods, but those are left as an exercise for you to try on your own.

Applying functions to transform data

In situations where a direct mapping or substitution will not suffice, it is possible to apply a function to the data to perform an algorithm on the data. Pandas provides the ability to apply functions to individual items, entire columns, or entire rows, providing incredible flexibility in transformation.

Functions can be applied using the conveniently named `.apply()` method. When given a Python function, this method iteratively calls the function while passing in each value from a `Series`. If applied to a `DataFrame`, pandas will pass in each column as a `Series`, or if applied along `axis=1`, it will pass in a `Series` representing each row.

The following demonstrates this by applying a lambda function to each item of a `Series`:

```
In [57]: # demonstrate applying a function to every item of a Series
s = pd.Series(np.arange(0, 5))
s.apply(lambda v: v * 2)
```

```
Out[57]: 0    0
          1    2
          2    4
          3    6
          4    8
         dtype: int64
```

When applying a function to items in a `Series`, only the value for each `Series` item is passed to the function, not the index label and the value.

When a function is applied to a `DataFrame`, the default is to apply the method to each column. Pandas iterates through all the columns, passing each as a `Series` to your function. The result is a `Series` object with index labels matching the column names, and the result of the function applied to the column:

```
In [59]: # calculate cumulative sum of items in each column
df.apply(lambda col: col.sum())
```

```
Out[59]: a    18
          b    22
          c    26
         dtype: int64
```

Application of the function can be switched to the values from each row by specifying `axis=1`:

```
In [60]: # calculate sum of items in each row  
df.apply(lambda row: row.sum(), axis=1)
```

```
Out[60]: 0      3  
1     12  
2     21  
3    30  
dtype: int64
```

A common practice is to take the result of an apply operation and add it as a new column of the DataFrame. This is convenient, as you can add onto the DataFrame the result of one or more successive calculations, providing yourself with progressive representations of the derivation of results through every step of the process.

The following code demonstrates this process. The first step multiplies column a by column b, and creates a new column named interim. The second step adds these values and column c, and creates a result column with those values:

```
In [62]: # and now a 'result' column with 'interim' + 'c'  
df['result'] = df.apply(lambda r: r.interim + r.c, axis=1)  
df
```

```
Out[62]:   a   b   c  interim  result  
0   0   1   2        0       2  
1   3   4   5       12      17  
2   6   7   8       42      50  
3   9  10  11      90     101
```

If you would like to change the values in the existing column, simply assign the result to an already existing column. The following changes the a column's values to be the sum of the values in the row:

```
In [63]: # replace column a with the sum of columns a, b and c  
df.a = df.a + df.b + df.c  
df
```

```
Out[63]:   a   b   c  interim  result  
0   3   1   2        0       2  
1  12   4   5       12      17  
2  21   7   8       42      50  
3  30  10  11      90     101
```

As a matter of practice, replacing a column with completely new values is not the best way to do things and often leads to situations of temporary (and potentially permanent) insanity, trying to debug problems caused by data that is lost. Therefore, in pandas, it is a better and common practice to just add new rows or columns (or totally new objects), and if memory or performance becomes a problem later on, do the optimizations as required.

Another point to note is that a pandas `DataFrame` is not a spreadsheet, where cells are assigned formulas and can be recalculated when cells that are referenced by the formula change. If you desire this to happen, you need to execute the formulas whenever the dependent data changes. On the flip side, this is more efficient than spreadsheets, as every little change does not cause a cascade of operations to occur.

The `.apply()` method always applies the provided function to all the items in `Series`, column, or row. If you want to apply the function to a subset of these, then first perform a Boolean selection to filter the items you do not want processed.

The following demonstrates this by creating a `DataFrame` of values and inserting one `NaN` value into the second row. It then applies a function to only those rows where all the values are not `NaN`:

```
In [64]: # create a 3x5 DataFrame
# only second row has a NaN
df = pd.DataFrame(np.arange(0, 15).reshape(3,5))
df.loc[1, 2] = np.nan
df
```

```
Out[64]:   0    1    2    3    4
0    0    1    2.0   3    4
1    5    6    NaN   8    9
2   10   11   12.0  13   14
```

```
In [65]: # demonstrate applying a function to only rows having
# a count of 0 NaN values
df.dropna().apply(lambda x: x.sum(), axis=1)
```

```
Out[65]: 0    10.0
2    60.0
dtype: float64
```

The final method we will cover in this chapter is to apply functions using the `.applymap()` method of `DataFrame`. While the `.apply()` method was always passed an entire row or column, the `.applymap()` function applies the function to each and every individual value.

The following demonstrates a practical use by using the `.applymap()` method to format each value in a `DataFrame` to a specified number of decimal points:

```
In [66]: # use applymap to format all items of the DataFrame
df.applymap(lambda x: '%.2f' % x)
```

```
Out[66]:      0    1    2    3    4
0    0.00  1.00  2.00  3.00  4.00
1    5.00  6.00  nan   8.00  9.00
2   10.00 11.00 12.00 13.00 14.00
```


Summary

In this chapter, we have examined various techniques of tidying data. We covered identifying missing data, replacing it with other values, or dropping it from the overall set of data. We then covered how to transform values into other values that may be better suited for further analysis.

Now that we have tidied up our data within DataFrame or series, we want to move from focusing on the cleanliness of the data to more elaborate forms of modifying the structure of the data, such as concatenation, merges, joins, and pivoting data. This will be the focus of the next chapter.

Combining, Relating, and Reshaping Data

Data is often modeled as a set of entities, the logical structures of related values referenced by name (properties/variables), and with multiple samples or instances that are organized by rows. Entities tend to represent real-world things, such as a person, or in the Internet of Things, a sensor. Each specific entity and its measurements are then modeled using a single DataFrame.

There will often need to be various tasks performed upon and between entities in a model. It may be required to combine the data for multiple customer entities, which are sourced from multiple locations, into single pandas object. Customer and order entities are often related to find a shipping address for an order. It is also possible that data stored in one model may need to be reshaped into another model, simply because different sources model the same type of entities differently.

In this chapter, we will examine these operations that let us combine, relate, and reshape data in our model. Specifically, in this chapter, we will examine the following concepts:

- Concatenating data in multiple pandas objects
- Merging data in multiple pandas objects
- How to control the type of join used in a merge
- Pivoting data to and from values and indexes
- Stacking and unstacking data
- Melting data to and from wide and long format

Configuring pandas

We start the examples in the chapter using the following imports and configuration statements:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set some pandas options controlling output format
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```


Concatenating data in multiple objects

Concatenation in pandas is the process of combining the data from two or more pandas objects into a new object. Concatenation of the `Series` objects simply results in a new `Series`, with the values copied in sequence.

The process of concatenating the `DataFrame` objects is more complex. The concatenation can be applied to either axis of the specified objects, and along that axis pandas performs relational join logic to the index labels. Then, along the opposite axis, pandas performs alignment of the labels and filling of missing values.

Because there are a number of factors to consider, we will break down the examples for concatenation into the following topics:

- Understanding the default semantics of concatenation
- Switching the axis of alignment
- Specifying the join type
- Appending data instead of concatenation
- Ignoring the index labels

Understanding the default semantics of concatenation

Concatenation is performed using the pandas function `pd.concat()`. The general syntax to concatenate data is to pass a list of objects to be concatenated. The following demonstrates a simple concatenation of two Series objects, `s1` and `s2`:

```
In [4]: # concatenate them  
pd.concat([s1, s2])  
  
Out[4]: 0    0  
1    1  
2    2  
0    5  
1    6  
2    7  
dtype: int64
```

This concatenated the index labels and values of `s2` to the end of those from `s1`. This resulted in duplicate index labels, as alignment is not performed during the process.

Two DataFrame objects can also be concatenated in a similar manner:

```
In [7]: # do the concat  
pd.concat([df1, df2])  
  
Out[7]:   a   b   c  
0    0   1   2  
1    3   4   5  
2    6   7   8  
0    9  10  11  
1   12  13  14  
2   15  16  17
```

The default functionality results in the rows being appended in order, and can result in duplicate index labels along the rows index.

The resulting set of column labels is defined by the union of the index labels in the specified DataFrame objects. This is an alignment that is applied across all the source objects (there can be more than two). Pandas will insert the `NaN` values if a column in the result does not exist in the DataFrame object currently being processed.

The following demonstrates the alignment of two DataFrame objects during concatenation, where there are common columns (`a` and `c`) and also distinct columns (`b` in `df1`, and `d` in `df2`):

```
In [10]: # do the concat, NaN's will be filled in for  
# the d column for df1 and b column for df2  
pd.concat([df1, df2])
```

```
Out[10]:   a    b    c    d  
0    0  1.0    2  NaN  
1    3  4.0    5  NaN  
2    6  7.0    8  NaN  
0    9  NaN   10  11.0  
1   12  NaN   13  14.0  
2   15  NaN   16  17.0
```

df1 does not contain column d, so values in that part of the result are NaN. The same occurs for df2 and column b.

It is possible to give each group of data in the result its own name using the keys parameter. This creates a hierarchical index on the DataFrame object that lets you refer to each group of data independently via the DataFrame object's .loc property. This is convenient if you later need to determine where data in the resulting DataFrame object was sourced.

The following demonstrates this concept by assigning names to each original DataFrame object and then retrieving the rows that originated in the df2 object (now indexed with the 'df2' label):

These keys can then be used to sub-select a specific set of data:

```
In [12]: # we can extract the data originating from  
# the first or second source DataFrame  
c.loc['df2']
```

```
Out[12]:   a    b    c    d  
0    9  NaN   10  11.0  
1   12  NaN   13  14.0  
2   15  NaN   16  17.0
```


Switching axes of alignment

The `pd.concat()` function allows you to specify the axis on which to apply the alignment during the concatenation. The following concatenates the two `DataFrame` objects along the column axis, switching alignment to be along the rows index:

```
In [13]: # concat df1 and df2 along columns  
# aligns on row labels, has duplicate columns  
pd.concat([df1, df2], axis=1)
```

```
Out[13]:   a   b   c   a   c   d  
0    0   1   2   9  10  11  
1    3   4   5  12  13  14  
2    6   7   8  15  16  17
```

This result now contains duplicate columns. This is because the concatenation first aligns by the row index labels of each `DataFrame` object, and then fills in the columns from the first `DataFrame` object and then the second without regard to the row index labels.

The following demonstrates a concatenation along the column axis with two `DataFrame` objects that have several row index labels in common (2 and 3), as well as along with disjoint rows (0 in `df1` and 4 in `df3`). Additionally, several of the columns in `df3` overlap with `df1` (a) as well as being disjointed (d):

```
In [15]: # concat them. Alignment is along row labels  
# columns first from df1 and then df3, with duplicates.  
# NaN filled in where those columns do not exist in the source  
pd.concat([df1, df3], axis=1)
```

```
Out[15]:      a      b      c      a      d  
0    0.0    1.0    2.0    NaN    NaN  
1    3.0    4.0    5.0    NaN    NaN  
2    6.0    7.0    8.0   20.0   21.0  
3    NaN    NaN    NaN   22.0   23.0  
4    NaN    NaN    NaN   24.0   25.0
```

Since the alignment was along the row labels, the columns end up having duplicates. The rows then have `NaN` values where the columns did not exist in the source objects.

Specifying join type

A default concatenation actually performs an **outer join** operation along the index labels on the axis opposite of the concatenation (the rows index). This makes the resulting set of labels similar to having performed a union of those labels.

The type of join can be changed to an **inner join** by specifying `join='inner'` as a parameter. The inner join then logically performs an intersection of labels instead of a union. The following demonstrates this and results in a single row, because `2` is the only row index label in common:

```
In [16]: # do an inner join instead of outer
          # results in one row
          pd.concat([df1, df3], axis=1, join='inner')

Out[16]:   a   b   c   a   d
           2   6   7   8  20  21
```

It is also possible to label groups of data along the columns using the `keys` parameter when applying the concatenation along `axis=1`:

```
In [17]: # add keys to the columns
          df = pd.concat([df1, df2],
                         axis=1,
                         keys=['df1', 'df2'])
          df

Out[17]:    df1        df2
            a   b   c   a   c   d
            0   0   1   2   9  10  11
            1   3   4   5  12  13  14
            2   6   7   8  15  16  17
```

The different groups can be accessed using the `.loc` property and slicing:

```
In [18]: # retrieve the data that originated from the
          # DataFrame with key 'df2'
          df.loc[:, 'df2']

Out[18]:   a   c   d
           0   9  10  11
           1  12  13  14
           2  15  16  17
```


Appending versus concatenation

A `DataFrame` (and `Series`) object also contains a `.append()` method, which concatenates the two specified `DataFrame` objects along the row index labels:

```
In [19]: # append does a concatenate along axis=0
          # duplicate row index labels can result
          df1.append(df2)
```

```
Out[19]:   a    b    c    d
0    0  1.0    2  NaN
1    3  4.0    5  NaN
2    6  7.0    8  NaN
0    9  NaN   10  11.0
1   12  NaN   13  14.0
2   15  NaN   16  17.0
```

As with concatenation on `axis=1`, the index labels in the rows are copied without consideration of the creation of duplicates, and the column labels are joined in a manner which ensures no duplicate column name is included in the result.

Ignoring the index labels

If you would like to ensure that the resulting index does not have duplicates and preserves all the rows, you can use the `ignore_index=True` parameter. This essentially returns the same result, except with the new `Int64Index`:

```
In [20]: # remove duplicates in the result index by ignoring the
# index labels in the source DataFrame objects
df1.append(df2, ignore_index=True)

Out[20]:   a    b    c    d
0    0  1.0    2  NaN
1    3  4.0    5  NaN
2    6  7.0    8  NaN
3    9  NaN   10  11.0
4   12  NaN   13  14.0
5   15  NaN   16  17.0
```

 *This operation also functions on concatenations.*

Merging and joining data

Pandas allows the merging of pandas objects with database-like join operations, using the `pd.merge()` function and the `.merge()` method of a `DataFrame` object. A merge combines the data of two pandas objects by finding matching values in one or more columns or row indexes. It then returns a new object that represents a combination of the data from both, based on relational-database-like join semantics applied to those values.

Merges are useful as they allow us to model a single `DataFrame` for each type of data (one of the rules of having tidy data), but to be able to relate data in different `DataFrame` objects using values existing in both sets of data.

Merging data from multiple pandas objects

A practical example of merges would be that of looking up customer names from orders. To demonstrate this in pandas, we will use the following two `DataFrame` objects. One represents a list of customer details and the other represents the orders made by the customers and what day the order was made. They will be related to each other using the `CustomerID` columns in each.

```
In [22]: # and these are the orders made by our customers
# they are related to customers by CustomerID
orders = {'CustomerID': [10, 11, 10],
          'OrderDate': [date(2014, 12, 1),
                        date(2014, 12, 1),
                        date(2014, 12, 1)]}
orders = pd.DataFrame(orders)
orders
```

```
Out[22]:   CustomerID      OrderDate
0            10  2014-12-01
1            11  2014-12-01
2            10  2014-12-01
```

Now suppose we would like to ship the orders to the customers. We would need to merge the `orders` data with the `customers` detail data to determine the address for each order. This can be easily performed with the following statement:

```
In [23]: # merge customers and orders so we can ship the items
customers.merge(orders)
```

```
Out[23]:      Address  CustomerID      Name      OrderDate
0  Address for Mike           10    Mike  2014-12-01
1  Address for Mike           10    Mike  2014-12-01
2  Address for Marcia         11  Marcia  2014-12-01
```

Pandas has done something magical for us here by being able to accomplish this with such a simple piece of code. It has realized that our `customers` and `orders` objects, both have a column named `CustomerID`, and with this knowledge. It uses common values found in that column of both `DataFrame` objects to relate the data in both and form the merged data based on inner join semantics.

To be even more detailed on what occurs, what pandas specifically does is the following:

1. It determines the columns in both `customers` and `orders` with common labels. These columns are treated as the keys to perform the join.
2. It creates a new `DataFrame`, whose columns are the labels from the keys identified in step 1, followed by all the non-key labels from both the objects.
3. It matches values in the key columns of both `DataFrame` objects.
4. It then creates a row in the result for each set of matching labels.

5. It then copies the data from those matching rows from each source object into that respective row and columns of the result.
6. It assigns a new `Int64Index` to the result.

The join in a merge can use values from multiple columns. To demonstrate, the following creates two `DataFrame` objects and performs the merge using values in the `key1` and `key2` columns of both the objects:

```
In [26]: # demonstrate merge without specifying columns to merge
# this will implicitly merge on all common columns
left.merge(right)
```

```
Out[26]:   key1  key2  lval1  rval1
0      a      x      0      6
1      c      z      2      8
```

This merge identifies that the `key1` and `key2` columns are common in both the `DataFrame` objects. The matching tuples of values in both `DataFrame` objects for these columns are (`a`, `x`) and (`c`, `z`), and therefore, this results in two rows of values.

To explicitly specify which column is used to relate the objects, you can use the `on` parameter. The following demonstrates this by performing a merge using only the values in the `key1` column of both the `DataFrame` objects:

```
In [27]: # demonstrate merge using an explicit column
# on needs the value to be in both DataFrame objects
left.merge(right, on='key1')
```

```
Out[27]:   key1  key2_x  lval1  key2_y  rval1
0      a      x      0      x      6
1      b      y      1      a      7
2      c      z      2      z      8
```

Comparing this result to the previous example, the result now has three rows, as there are matching `a`, `b`, and `c` values in that single column of both objects.

The `on` parameter can also be given a list of column names. The following reverts to using both the `key1` and `key2` columns, which results in an identical result to the previous example where those two columns were implicitly identified by pandas:

```
In [28]: # merge explicitly using two columns
left.merge(right, on=['key1', 'key2'])
```

```
Out[28]:   key1  key2  lval1  rval1
0      a      x      0      6
1      c      z      2      8
```

The columns specified with `on` need to exist in both the `DataFrame` objects. If you would like to merge based on columns with different names in each object, you can use the `left_on` and `right_on` parameters, passing the name or names of the columns to each respective parameter.

To perform a merge with the labels of the row indexes of the two `DataFrame` objects, you can use the `left_index=True` and `right_index=True` parameters (both need to be specified):

```
In [29]: # join on the row indices of both matrices
pd.merge(left, right, left_index=True, right_index=True)
```



```
Out[29]:   key1_x  key2_x  lval1  key1_y  key2_y  rval1
1          b        y      1        a        x      6
2          c        z      2        b        a      7
```

This has identified that the index labels in common are `1` and `2`, so the resulting `DataFrame` has two rows with these values and labels in the index. Pandas then creates a column in the result for every column in both the objects, and then copies the values.

As both `DataFrame` objects had a column with identical name `key`, the columns in the result have the `_x` and `_y` suffixes appended to them to identify the `DataFrame` object they originated from. `_x` is for left and `_y` is for right. You can specify these suffixes using the `suffixes` parameter and passing a two-item sequence.

Specifying the join semantics of a merge operation

The default type of join performed by `pd.merge()` is an **inner join**. To use another join method, specify the join type using the `how` parameter of the `pd.merge()` function (or the `.merge()` method). The valid options are:

- `inner`: This is the intersection of keys from both `DataFrame` objects
- `outer`: This is the union of keys from both `DataFrame` objects
- `left`: This only uses keys from the left `DataFrame`
- `right`: This only uses keys from the right `DataFrame`

As we have seen, an inner join is the default and it returns a merge of the data from both `DataFrame` objects only where the values match.

An outer join, in contrast, returns both the merge of the matched rows and the unmatched values from both the left and right `DataFrame` objects, but with `NaN` filled in the unmatched portion. The following code demonstrates an outer join:

```
In [30]: # outer join, merges all matched data,
# and fills unmatched items with NaN
left.merge(right, how='outer')
```

```
Out[30]:   key1  key2  lval1  rval1
          0      a      x    0.0    6.0
          1      b      y    1.0    NaN
          2      c      z    2.0    8.0
          3      b      a    NaN    7.0
```

A left join will return the merge of the rows that satisfy the join of the values in the specified columns, and also returns the unmatched rows from only `left`:

```
In [31]: # left join, merges all matched data, and only fills unmatched
# items from the left dataframe with NaN filled for the
# unmatched items in the result
# rows with labels 0 and 2
# match on key1 and key2 the row with label 1 is from left

left.merge(right, how='left')
```

```
Out[31]:   key1  key2  lval1  rval1
          0      a      x    0    6.0
          1      b      y    1    NaN
          2      c      z    2    8.0
```

A right join will return the merge of the rows that satisfy the join of the values in the specified columns, and also returns the unmatched rows from only `right`:

```
In [32]: # right join, merges all matched data, and only fills unmatched
# item from the right with NaN filled for the unmatched items
# in the result
# rows with labels 0 and 2 match on key1 and key2
# the row with label 1 is from right
left.merge(right, how='right')
```

```
Out[32]:   key1  key2  lval1  rval1
0      a      x    0.0      6
1      c      z    2.0      8
2      b      a    NaN      7
```

The pandas library also provides a `.join()` method that can be used to perform a join using the index labels of the two `DataFrame` objects (instead of values in columns). Note that if the columns in the two `DataFrame` objects do not have unique column names, you must specify suffixes using the `lsuffix` and `rsuffix` parameters (automatic suffixing is not performed, as with `merge`). The following code demonstrates both the join and the specification of suffixes:

```
In [33]: # join left with right (default method is outer)
# and since these DataFrame objects have duplicate column names
# we just specify lsuffix and rsuffix
left.join(right, lsuffix='_left', rsuffix='_right')
```

```
Out[33]:   key1_left  key2_left  lval1  key1_right  key2_right  rval1
0            a          x      0           NaN          NaN      NaN
1            b          y      1           a          x      6.0
2            c          z      2           b          a      7.0
```

The default type of join performed is an outer join. Note that this differs from the default of the `.merge()` method, which defaults to inner. To change to an inner join, specify `how='inner'`, as is demonstrated in the following example:

```
In [34]: # join left with right with an inner join
left.join(right, lsuffix='_left', rsuffix='_right', how='inner')
```

```
Out[34]:   key1_left  key2_left  lval1  key1_right  key2_right  rval1
1            b          y      1           a          x      6
2            c          z      2           b          a      7
```

Note that this is roughly equivalent to the earlier result from `out [29]`, except the result has columns with slightly different names.

It is also possible to perform right and left joins, but they lead to results similar to the previous examples, so they will be omitted for brevity.

Pivoting data to and from value and indexes

Data is often stored in a stacked format, which is also referred to as record format. This is common in databases, .csv files, and Excel spreadsheets. In a stacked format, the data is often not normalized and has repeated values in many columns, or values that should logically exist in other tables (violating another concept of tidy data).

Take the following data which represents a stream of data from an accelerometer on a

```
In [35]: # read in accelerometer data
sensor_readings = pd.read_csv("data/accel.csv")
sensor_readings
```

```
Out[35]:    interval  axis   reading
0            0      X     0.0
1            0      Y     0.5
2            0      Z     1.0
3            1      X     0.1
4            1      Y     0.4
..          ...
7            2      Y     0.3
8            2      Z     0.8
9            3      X     0.3
10           3      Y     0.2
11           3      Z     0.7

[12 rows x 3 columns]
```

An issue with this data as it is organized is: how does one go about determining the readings for a specific axis? This can be naively done with Boolean selections:

```
In [36]: # extract X-axis readings
sensor_readings[sensor_readings['axis'] == 'X']
```

```
Out[36]:    interval  axis   reading
0            0      X     0.0
3            1      X     0.1
6            2      X     0.2
9            3      X     0.3
```

An issue here is, what if you want to know the values for all axes at a given time and not just the x axis. To do this, you can perform a selection for each value of the axis, but that is repetitive code and does not handle the scenario of new axis values being inserted into `DataFrame` without a change to the code.

A better representation would be where columns represent the unique variable values. To convert to this form, use the `DataFrame` objects' `.pivot()` function:

```
In [37]: # pivot the data. Interval becomes the index, the columns are
# the current axes values, and use the readings as values
sensor_readings.pivot(index='interval',
                       columns='axis',
                       values='reading')
```

```
Out[37]: axis      X      Y      Z
interval
0          0.0    0.5    1.0
1          0.1    0.4    0.9
2          0.2    0.3    0.8
3          0.3    0.2    0.7
```

This has taken all the distinct values from the axis column and pivoted them into columns on the new DataFrame, while filling in values for the new columns from the appropriate rows and columns of the original DataFrame. This new DataFrame demonstrates that it is now very easy to identify the x, y, and z sensor readings at each time interval.

Stacking and unstacking

Similar to the pivot function are the `.stack()` and `.unstack()` methods. The process of stacking pivots a level of column labels to the row index. Unstacking performs the opposite, that is, pivoting a level of the row index into the column index.

One of the differences between stacking/unstacking and performing a pivot is that unlike pivots, the stack and unstack functions are able to pivot specific levels of a hierarchical index. Also, where a pivot retains the same number of levels on an index, a stack and unstack always increases the levels on the index of one of the axes (columns for unstacking and rows for stacking) and decrease the levels on the other axis.

Stacking using non-hierarchical indexes

To demonstrate stacking, we will look at several examples using a `DataFrame` object with non-hierarchical indexes. We will begin our examples using the following `DataFrame`:

```
In [38]: # simple DataFrame with one column
df = pd.DataFrame({'a': [1, 2]}, index={'one', 'two'})
df
```

```
Out[38]:      a
one    1
two    2
```

Stacking will move one level of the column's index into a new level of the row's index. As our `DataFrame` only has one level, this collapses a `DataFrame` object into a `Series` object with a hierarchical row index:

```
In [39]: # push the column to another level of the index
# the result is a Series where values are looked up through
# a multi-index
stacked1 = df.stack()
stacked1
```

```
Out[39]: one    a    1
          two   a    2
          dtype: int64
```

To access values, we now need to pass a tuple to the indexer of the `Series` object, which does the lookup with just the index:

```
In [40]: # lookup one / a using just the index via a tuple
stacked1[('one', 'a')]
```

```
Out[40]: 1
```

If the `DataFrame` object contains multiple columns, then all the columns are moved to the same additional level of the new `Series` object:

```
In [42]: # push the two columns into a single level of the index
stacked2 = df.stack()
stacked2
```

```
Out[42]: one    a    1
          b    3
          two   a    2
          b    4
          dtype: int64
```

Values for what would have previously been different columns, can now still be accessed using the tuple syntax with the index:

```
In [43]: # lookup value with index of one / b  
stacked2[('one', 'b')]
```

```
Out[43]: 3
```

Unstacking will perform a similar operation in the opposite direction, by moving a level of the row index into a level of the column's axis. We will examine this process in the next section as unstacking generally assumes that the index being unstacked is hierarchical.

Unstacking using hierarchical indexes

To demonstrate unstacking with hierarchical indexes, we will revisit the sensor data we saw earlier in the chapter. However, we will add an additional column to the measurement data that represents readings for multiple users and copy data for two users. The following sets up this data:

```
Out[44]:      reading
   who    interval axis
   Mike      0       X    0.0
              Y    0.5
              Z    1.0
             1       X    0.1
              Y    0.4
...
Mikael     2       Y   30.0
              Z   80.0
             3       X   30.0
              Y   20.0
              Z   70.0

[24 rows x 1 columns]
```

With this organization in the data, we can do such things as examine all the readings for a specific person using only the index:

```
In [45]: # lookup user data for Mike using just the index
multi_user_sensor_data.loc['Mike']
```

```
Out[45]:      reading
   interval axis
   0       X    0.0
              Y    0.5
              Z    1.0
   1       X    0.1
              Y    0.4
...
   2       Y    0.3
              Z    0.8
   3       X    0.3
              Y    0.2
              Z    0.7

[12 rows x 1 columns]
```

We can also get all the readings of all axes and for all users at interval 1 using `.xs()`:

```
In [46]: # readings for all users and axes at interval 1  
multi_user_sensor_data.xs(1, level='interval')
```

```
Out[46]:      reading  
    who     axis  
    Mike     X      0.1  
              Y      0.4  
              Z      0.9  
    Mikael   X     10.0  
              Y     40.0  
              Z    90.0
```

Unstacking will move the last level of the row index into a new level of the column index, resulting in columns having `MultiIndex`. The following demonstrates the last level of this unstacking (the `axis` level of the index):

```
In [47]: # unstack the who level  
multi_user_sensor_data.unstack()
```

```
Out[47]:      reading  
    axis          X      Y      Z  
    who   interval  
    Mikael 0      0.0  50.0 100.0  
          1      10.0 40.0  90.0  
          2      20.0 30.0  80.0  
          3      30.0 20.0  70.0  
    Mike   0      0.0  0.5  1.0  
          1      0.1  0.4  0.9  
          2      0.2  0.3  0.8  
          3      0.3  0.2  0.7
```

To unstack a different level, use the `level` parameter. The following code unstacks the first level (`level=0`):

```
In [48]: # unstack at level=0  
multi_user_sensor_data.unstack(level=0)
```

```
Out[48]:      reading  
    who          Mikael Mike  
interval axis  
0      X      0.0  0.0  
        Y      50.0  0.5  
        Z     100.0  1.0  
1      X      10.0  0.1  
        Y      40.0  0.4  
...  
2      Y      30.0  0.3  
        Z      80.0  0.8  
3      X      30.0  0.3  
        Y      20.0  0.2  
        Z      70.0  0.7  
  
[ 12 rows x 2 columns]
```

Multiple levels can be unstacked simultaneously by passing a list of the levels to `.unstack()`. Additionally, if the levels are named, they can be specified by name instead of location. The following unstacks the `who` and `axis` levels by name:

```
In [49]: # unstack who and axis levels  
unstacked = multi_user_sensor_data.unstack(['who', 'axis'])  
unstacked
```

```
Out[49]:      reading  
    who          Mike          Mikael  
    axis         X         Y         Z         X         Y         Z  
interval  
0            0.0  0.5  1.0      0.0  50.0  100.0  
1            0.1  0.4  0.9     10.0  40.0   90.0  
2            0.2  0.3  0.8     20.0  30.0   80.0  
3            0.3  0.2  0.7     30.0  20.0   70.0
```

To be thorough, we can restack this data. The following code will stack the `who` level of the column back into the row index:

```
In [50]: # and we can of course stack what we have unstacked  
# this re-stacks who  
unstacked.stack(level='who')
```

```
Out[50]:
```

		reading		
axis		X	Y	Z
interval	who			
0	Mikael	0.0	50.0	100.0
	Mike	0.0	0.5	1.0
1	Mikael	10.0	40.0	90.0
	Mike	0.1	0.4	0.9
2	Mikael	20.0	30.0	80.0
	Mike	0.2	0.3	0.8
3	Mikael	30.0	20.0	70.0
	Mike	0.3	0.2	0.7

There are a couple of things worth pointing out about this result. First, stacking and unstacking always move the levels into the last levels of the other index. Note that the `who` level is now the last level of the row index, but it started out earlier as the first level. This would have ramifications on the code used to access elements via that index, as it has changed to another level. If you want to put a level back into another position, you will need to reorganize the indexes with other means than stacking and unstacking.

Second, with all this moving around of data, stacking and unstacking (as well as pivoting) do not lose any information. They simply change the means by which it is organized and accessed.

Melting data to and from long and wide format

Melting is a type of un-pivoting, and is often referred to as changing a `DataFrame` object from **wide format** to **long format**. This format is common in various statistical analyses, and data you read may be already provided in a melted form. Or you may need to pass data in this format to other code that expects this organization.

Technically, melting is the process of reshaping a `DataFrame` object into a format where two or more columns, referred to as `variable` and `value`, are created by un-pivoting column labels in the `variable` column, and then moving the data from these columns into the appropriate location in the `value` column. All other columns are then made into identifier columns that assist in describing the data.

The concept of melting is often best understood using a simple example. In this example, we start with a `DataFrame` object that represents measurements of two variables, each represented with its own column, `Height` and `Weight`, and one additional column representing the person and specified with the `Name` column:

```
In [51]: # we will demonstrate melting with this DataFrame
data = pd.DataFrame({'Name' : ['Mike', 'Mikael'],
                     'Height' : [6.1, 6.0],
                     'Weight' : [220, 185]})

data
```



```
Out[51]:    Height      Name  Weight
0      6.1     Mike     220
1      6.0   Mikael     185
```

The following melts this `DataFrame`, using the `Name` column as the identifier column, and the `Height` and `Weight` columns as measured variables. The `Name` column remains, with the `Height` and `Weight` columns un-pivoted into the `variable` column. Then the values from these two columns are rearranged into the `value` column, and ensured to align with the appropriate combination values of `Name` and `variable` that would have existed in the original data:

```
In [52]: # melt it, use Name as the id's,
# Height and Weight columns as the variables
pd.melt(data,
         id_vars=['Name'],
         value_vars=['Height', 'Weight'])
```



```
Out[52]:    Name  variable  value
0     Mike    Height    6.1
1  Mikael    Height    6.0
2     Mike   Weight  220.0
3  Mikael   Weight  185.0
```

The data is now restructured, so that it is easy to extract the value for any combination of `variable` and `Name`. Additionally, in this format it is easier to add a new variable and measurement, as the data can simply be

added as a new row instead of requiring a change of structure to `DataFrame` by adding a new column.

Performance benefits of stacked data

Finally, we will examine why we would want to stack data. It can be shown that stacked data is more efficient than using lookup through a single level index and then a column lookup, or even compared to an `.iloc` lookup that specifies the row and column by location. The following demonstrates this:

```
In [53]: # stacked scalar access can be a lot faster than
# column access

# time the different methods
import timeit
t = timeit.Timer("stacked1[('one', 'a')]",
                 "from __main__ import stacked1, df")
r1 = timeit.timeit(lambda: stacked1.loc[('one', 'a')],
                   number=10000)
r2 = timeit.timeit(lambda: df.loc['one']['a'],
                   number=10000)
r3 = timeit.timeit(lambda: df.iloc[1, 0],
                   number=10000)

# and the results are... Yes, it's the fastest of the three
r1, r2, r3
```

Out[53]: (0.5042991369991796, 1.2535194699958083, 0.1478830999985803)

This can have extreme benefits on an application performance if it is required to repeatedly access a large number of scalar values out of a `DataFrame`.

Summary

In this chapter, we examined several techniques of combining and reshaping data in one or more `DataFrame` objects. We started the chapter by examining how to combine data from multiple pandas objects. Then, we examined how to concatenate multiple `DataFrame` objects, both along the row and column axes. From this, we then examined how pandas can be used to perform database-like joins and merges of data based on values in multiple `DataFrame` objects.

We then examined how to reshape data in `DataFrame` using pivots, stacking, and melting. Through this, we saw how each of these processes provides several variations on how to move data around by changing the shape of the indexes, and by moving data in and out of indexes. This showed us how to organize data in formats that are efficient for lookup from other forms that may be more of convenience for the producer of the data.

In the next chapter, we will learn about grouping, and aggregate analysis of data in those groups, which allows us to derive results based upon like-values in the data.

Data Aggregation

Data aggregation is the process of grouping data based on some meaningful categories of the information. Analysis is then performed on each of the groups to report one or more summary statistics for each. This summarization in this sense is a general term in that *summarization* can literally be a summation (such as total number of units sold) or statistical calculation such as a mean or standard deviation.

This chapter will examine the facilities of pandas to perform data aggregation. This includes a powerful split-apply-combine pattern for grouping, performing group-level transformations and analyses, and reporting the results from every group within a summary pandas object. Within this framework, we will examine several techniques of grouping data, applying functions on a group level, and being able to filter data in or out of the analysis.

Specifically, in this chapter we will cover:

- An overview of the split, apply, and combine pattern for data analysis
- Grouping by a single column's values
- Accessing the results of a pandas grouping
- Grouping using the values in multiple columns
- Grouping using index levels
- Applying aggregation functions to grouped data
- An overview of transforming data
- Practical examples of transformation: filling with means and z-scores
- Using filtering to selectively remove groups of data
- Discretization and binning

Configuring pandas

The examples in this chapter use the following imports and configuration statements:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set formattign options
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

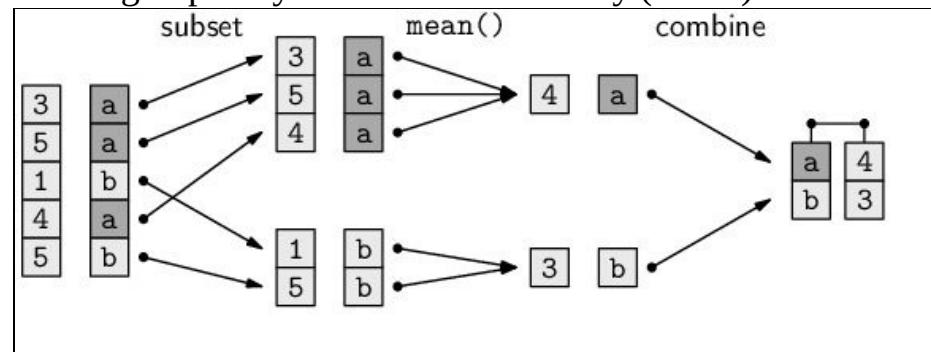
# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```


The split, apply, and combine (SAC) pattern

Many data analysis problems utilize a pattern of processing data referred to as **split-apply-combine**. In this pattern, three steps are taken to analyze data:

- A dataset is split into smaller pieces based on certain criteria
- Each of these pieces are operated upon independently
- All the results are then combined back and presented as a single unit

The following diagram demonstrates a simple split-apply-combine process to calculate the mean of values grouped by a character-based key (a or b):



The data is then split by the index label into two groups (one each for a and b). The mean of the values in each group is calculated. The resulting values from the group are then combined into a single pandas object, which is indexed by the label representing each group.

Splitting in pandas is performed using the `.groupby()` method of a `Series` or `DataFrame`. This method is given one or more index label and/or column name; they will group the data based on the associated values.

Once the data is split, it is possible to perform one or more of the following categories of operations on each group:

- **Aggregation:** Calculate a summary statistic such as group means or counts of the items in each group
- **Transformation:** Perform group- or item-specific calculations
- **Filtration:** Remove entire groups of data based on a group-level calculation

The final stage, combine, is performed automatically by pandas, which collects the results of the apply stage and constructs a single merged result.



For more information on split-apply-combine, there is a paper from the Journal of Statistical Software titled "The Split-Apply-Combine Strategy for Data Analysis". This paper goes into more details of the pattern, and although it utilizes R in its examples, it is still a valuable read for someone learning pandas. You can get this paper at <http://www.jstacs.org/v40/i01/paper>.

Data for the examples

The examples in this chapter will utilize a dataset that represents measurements of several device sensors. The data consists of readings on the **X**, **Y**, and **Z** axis of both an accelerometer and orientation sensor:

```
In [2]: # load the sensors data  
sensor_data = pd.read_csv("data/sensors.csv")  
sensor_data[:5]
```

```
Out[2]:   interval  sensor  axis  reading  
0           0    accel     Z      0.0  
1           0    accel     Y      0.5  
2           0    accel     X      1.0  
3           1    accel     Z      0.1  
4           1    accel     Y      0.4
```


Splitting data

Our examination of splitting data within a pandas object will be broken into several steps. In the first, we will look at creating a grouping based on columns, and then examine the properties of the grouping that is created. We will then examine accessing various properties and results of the grouping to learn several properties of the groups that were created. Then we will examine grouping using index labels instead of content in columns.

Grouping by a single column's values

The sensor data consists of three categorical variables (`sensor`, `interval`, and `axis`) and one continuous variable (`reading`). It is possible to group by any single categorical variable by passing its name to `.groupby()`. The following code groups the sensor data by the values in the `sensor` column:

```
In [3]: # group this data by the sensor column / variable  
# returns a DataFrameGroupBy object  
grouped_by_sensor = sensor_data.groupby('sensor')  
grouped_by_sensor
```

```
Out[3]: <pandas.core.groupby.DataFrameGroupBy object at 0x1086282b0>
```

The result of `.groupby()` on a `DataFrame` is a subclass of a `GroupBy` object, either a `DataFrameGroupBy` for a `DataFrame` or a `SeriesGroupBy` for a `Series`. This object represents an interim description of the grouping that will eventually be performed. This object helps pandas to first validate the grouping relative to the data prior to execution. This can help in optimization and identification of errors, and gives you a point where you can check certain properties prior to what could be an expensive computational process.

This interim object has many useful properties. The `.ngroups` property will retrieve the number of groups that will be formed in the result:

```
In [4]: # get the number of groups that this will create  
grouped_by_sensor.ngroups
```

```
Out[4]: 2
```

The `.groups` property will return a Python dictionary whose keys represent the names of each group (if multiple columns are specified, it is a tuple). The values in the dictionary are an array of the index labels contained within each respective group:

```
In [5]: # what are the groups that were found?  
grouped_by_sensor.groups
```

```
Out[5]: {'accel': Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], dtype='int64'),  
         'orientation': Int64Index([12, 13, 14, 15, 16, 17, 18, 19, 20, 21,  
                                     22,  
                                     23],  
                                    dtype='int64')}
```


Accessing the results of a grouping

The `grouped` variable can be thought of as a collection of named groups and can be used to examine the contents of the groupings. Let's examine these groupings using the following function:

```
In [6]: # a helper function to print the contents of the groups
def print_groups (group_object):
    # loop over all groups, printing the group name
    # and group details
    for name, group in group_object:
        print name
        print (group[:5])
```

This function will loop through each of the groups and print its name and the first five rows:

```
In [7]: # examine the content of the groups we created
print_groups(grouped_by_sensor)
```

```
accel
  interval sensor axis  reading
0          0    accel    Z      0.0
1          0    accel    Y      0.5
2          0    accel    X      1.0
3          1    accel    Z      0.1
4          1    accel    Y      0.4
orientation
  interval      sensor axis  reading
12         0 orientation    Z      0.0
13         0 orientation    Y      0.1
14         0 orientation    X      0.0
15         1 orientation    Z      0.0
16         1 orientation    Y      0.2
```

An examination of these results gives us some insights into how pandas has performed the split. A group has been created for each distinct value in the `sensors` column, and the group is named after the value. Each group then contains a `DataFrame` object consisting of the rows where the sensor value matched the name of the group.

The `.size()` method returns a summary of the size of all the groups:

```
In [8]: # get how many items are in each group
grouped_by_sensor.size()
```

```
Out[8]: sensor
accel           12
orientation     12
dtype: int64
```

The `.count()` method returns the number of items in each column of every group:

```
In [9]: # get the count of items in each column of each group  
grouped_by_sensor.count()
```

```
Out[9]:      interval  axis  reading  
sensor  
accel           12     12      12  
orientation     12     12      12
```

Any specific group can be retrieved using the `.get_group()` method. The following code retrieves the `accel` group:

```
In [10]: # get the data in one specific group  
grouped_by_sensor.get_group('accel')[:5]
```

```
Out[10]:      interval  sensor  axis  reading  
0            0    accel    Z      0.0  
1            0    accel    Y      0.5  
2            0    accel    X      1.0  
3            1    accel    Z      0.1  
4            1    accel    Y      0.4
```

The `.head()` and `.tail()` methods can be used to return the specified number of items in each group. This code retrieves the first three rows in each group:

```
In [11]: # get the first three items in each group  
grouped_by_sensor.head(3)
```

```
Out[11]:      interval      sensor  axis  reading  
0            0    accel    Z      0.0  
1            0    accel    Y      0.5  
2            0    accel    X      1.0  
12           0  orientation    Z      0.0  
13           0  orientation    Y      0.1  
14           0  orientation    X      0.0
```

The `.nth()` method will return the n -th item in each group. The following code demonstrates how to use this to retrieve the second row of each group:

```
In [12]: # get the 2nd item in each group  
grouped_by_sensor.nth(1)
```

```
Out[12]:      axis  interval  reading  
sensor  
accel        Y          0      0.5  
orientation   Y          0      0.1
```

The `.describe()` method can be used to return descriptive statistics for each group:

```
In [13]: # get descriptive statistics for each group  
grouped_by_sensor.describe()
```

```
Out[13]:
```

	interval	count	mean	std	...	reading	\
sensor					...		
accel	12.0	1.5	1.167748	...		0.35	0.725
orientation	12.0	1.5	1.167748	...		0.10	0.225

	max
sensor	
accel	1.0
orientation	0.4


```
[2 rows x 16 columns]
```

Groups are sorted by their group name in ascending order. If you want to prevent sorting during grouping, use the `sort=False` option.

Grouping using multiple columns

Grouping can also be performed on multiple columns by passing a list of column names. The following code groups the data by both the `sensor` and `axis` columns:

```
In [14]: # group by both sensor and axis values
mcg = sensor_data.groupby(['sensor', 'axis'])
print_groups(mcg)

('accel', 'X')
    interval  sensor  axis  reading
2           0  accel    X      1.0
5           1  accel    X      0.9
8           2  accel    X      0.8
11          3  accel    X      0.7

('accel', 'Y')
    interval  sensor  axis  reading
1           0  accel    Y      0.5
4           1  accel    Y      0.4
7           2  accel    Y      0.3
10          3  accel    Y      0.2

('accel', 'Z')
    interval  sensor  axis  reading
0           0  accel    Z      0.0
3           1  accel    Z      0.1
6           2  accel    Z      0.2
9           3  accel    Z      0.3

('orientation', 'X')
    interval      sensor  axis  reading
14          0  orientation  X      0.0
17          1  orientation  X      0.1
20          2  orientation  X      0.2
23          3  orientation  X      0.3

('orientation', 'Y')
    interval      sensor  axis  reading
13          0  orientation  Y      0.1
16          1  orientation  Y      0.2
19          2  orientation  Y      0.3
22          3  orientation  Y      0.4

('orientation', 'Z')
    interval      sensor  axis  reading
12          0  orientation  Z      0.0
15          1  orientation  Z      0.0
18          2  orientation  Z      0.0
21          3  orientation  Z      0.0
```

Since multiple columns were specified, the name of each group is now a tuple representing each distinct combination of values from both `sensor` and `axis`.

Grouping using index levels

It is possible to group by using the values in the index instead of the columns. The sensor data is well suited for a hierarchical index, which can be used to demonstrate this concept. Let's set up a form of this data with a hierarchical index consisting of the `sensor` and `axis` columns:

```
In [15]: # make a copy of the data and reindex the copy
mi = sensor_data.copy()
mi = mi.set_index(['sensor', 'axis'])
mi

Out[15]:
      interval  reading
sensor    axis
accel      Z        0    0.0
           Y        0    0.5
           X        0    1.0
           Z        1    0.1
           Y        1    0.4
...
...
orientation Y        2    0.3
           X        2    0.2
           Z        3    0.0
           Y        3    0.4
           X        3    0.3

[24 rows x 2 columns]
```

Grouping can now be performed using the various levels of the hierarchical index. This code will group by index level 0 (the sensor names):

```
In [16]: # group by the first level of the index
print_groups(mi.groupby(level=0))

accel
      interval  reading
sensor axis
accel  Z        0    0.0
       Y        0    0.5
       X        0    1.0
       Z        1    0.1
       Y        1    0.4
orientation
      interval  reading
sensor axis
orientation  Z        0    0.0
             Y        0    0.1
             X        0    0.0
             Z        1    0.0
             Y        1    0.2
```

Grouping by multiple levels can be performed by passing the levels in a list. And if `MultiIndex` has names specified for the levels, then these names can be used instead of integers. The following code demonstrates grouping by `sensor` and `axis`:

```
In [17]: # group by multiple levels of the index
print_groups(mi.groupby(level=['sensor', 'axis']))

('accel', 'X')
    interval  reading
sensor axis
accel X      0     1.0
        X      1     0.9
        X      2     0.8
        X      3     0.7
('accel', 'Y')
    interval  reading
sensor axis
accel Y      0     0.5
        Y      1     0.4
        Y      2     0.3
        Y      3     0.2
('accel', 'Z')
    interval  reading
sensor axis
accel Z      0     0.0
        Z      1     0.1
        Z      2     0.2
        Z      3     0.3
('orientation', 'X')
    interval  reading
sensor axis
orientation X  0     0.0
        X      1     0.1
        X      2     0.2
        X      3     0.3
('orientation', 'Y')
    interval  reading
sensor axis
orientation Y  0     0.1
        Y      1     0.2
        Y      2     0.3
        Y      3     0.4
('orientation', 'Z')
    interval  reading
sensor axis
orientation Z  0     0.0
        Z      1     0.0
        Z      2     0.0
        Z      3     0.0
```


Applying aggregate functions, transforms, and filters

The apply step allows for three distinct operations on each group of data:

- Applying an aggregation function
- Performing a transformation
- Filtering an entire group from the results

Let's examine each of these operations.

Applying aggregation functions to groups

An aggregation function can be applied to each group using the `.aggregate()` (or in short, `.agg()`) method of the `GroupBy` object. The parameter of `.agg()` is a reference to a function that will be applied to each group. In the case of a `DataFrame`, this function will be applied to each column of data within the group.

The following example demonstrates calculating the mean of the values for each `sensor` and `axis`:

```
In [18]: # calculate the mean for each sensor/axis
sensor_axis_grouping = mi.groupby(level=['sensor', 'axis'])
sensor_axis_grouping.agg(np.mean)
```

```
Out[18]:
```

		interval	reading
sensor	axis		
accel	X	1.5	0.85
	Y	1.5	0.35
	Z	1.5	0.15
orientation	X	1.5	0.15
	Y	1.5	0.25
	Z	1.5	0.00

As `.agg()` will apply the method to every column in each group, pandas also calculates the mean of the interval values (which may not be of much practical interest).

The result of the aggregation will have an identically structured index as the original data. The `as_index=False` can be used to create a numeric index and shift the level of the original index into columns:

```
In [19]: # do not create an index matching the original object
sensor_data.groupby(['sensor', 'axis'], as_index=False).agg(np.mean)
```

```
Out[19]:
```

	sensor	axis	interval	reading
0	accel	X	1.5	0.85
1	accel	Y	1.5	0.35
2	accel	Z	1.5	0.15
3	orientation	X	1.5	0.15
4	orientation	Y	1.5	0.25
5	orientation	Z	1.5	0.00

Many aggregation functions are built in directly to the `GroupBy` object to save you some typing. Specifically, these functions are (prefixed by `gb.`): `gb.agg` `gb.boxplot` `gb.cummin` `gb.describe` `gb.filter` `gb.get_group` `gb.height` `gb.last` `gb.median` `gb.ngroups` `gb.plot` `gb.rank` `gb.std` `gb.transform` `gb.aggregate` `gb.count` `gb.cumprod` `gb.dtype` `gb.first` `gb.groups` `gb.hist` `gb.max` `gb.min` `gb.nth` `gb.prod` `gb.resample` `gb.sum` `gb.var` `gb.apply` `gb.cummax` `gb.cumsum` `gb.fillna` `gb.gender` `gb.head` `gb.indices` `gb.mean` `gb.name` `gb.ohlc` `gb.quantile` `gb.size` `gb.tail` `gb.weight`

As a demonstration, the following code also calculates the mean for each `sensor` and `axis` combination:

```
In [20]: # can simply apply the agg function to the group by object  
sensor_axis_grouping.mean()
```

```
Out[20]:
```

		interval	reading
sensor	axis		
accel	X	1.5	0.85
	Y	1.5	0.35
	Z	1.5	0.15
orientation	X	1.5	0.15
	Y	1.5	0.25
	Z	1.5	0.00

It is also possible to apply multiple aggregation functions in the same statement by passing the functions in a list.

```
In [21]: # apply multiple aggregation functions at once  
sensor_axis_grouping.agg([np.sum, np.std])
```

```
Out[21]:
```

		interval	reading		
sensor	axis	sum	std	sum	std
accel	X	6	1.290994	3.4	0.129099
	Y	6	1.290994	1.4	0.129099
	Z	6	1.290994	0.6	0.129099
orientation	X	6	1.290994	0.6	0.129099
	Y	6	1.290994	1.0	0.129099
	Z	6	1.290994	0.0	0.000000

A different function can be applied to each column by passing a Python dictionary to `.agg()`. The keys of the dictionary represent the column name that the function is to be applied to, and the value of each dictionary entry is the function. The following code demonstrates this technique by calculating the mean of the `reading` column and returning the length of the group in place of the `interval` value:

```
In [22]: # apply a different function to each column  
sensor_axis_grouping.agg({'interval' : len,  
                           'reading' : np.mean})
```

```
Out[22]:
```

		interval	reading
sensor	axis		
accel	X	4	0.85
	Y	4	0.35
	Z	4	0.15
orientation	X	4	0.15
	Y	4	0.25
	Z	4	0.00

Aggregation can also be performed on specific columns using the `[]` operator on the `GroupBy` object. This code calculates the mean of only the `reading` column:

```
In [23]: # calculate the mean of the reading column  
sensor_axis_grouping['reading'].mean()
```

```
Out[23]: sensor      axis  
accel      X      0.85  
           Y      0.35  
           Z      0.15  
orientation X      0.15  
           Y      0.25  
           Z      0.00  
Name: reading, dtype: float64
```


Transforming groups of data

A `GroupBy` object provides a `.transform()` method that applies a function to the all values in the `DataFrame` in each group. We will examine the general transformation process and then look at two real-world examples.

The general process of transformation

The `.transform()` method of a `GroupBy` object applies a function to every value in the DataFrame and returns another DataFrame that has the following characteristics:

- It is indexed identically to the concatenation of the indexes in all the groups
- The number of rows is equal to the sum of the number of rows in all the groups
- It consists of non-grouped columns to which pandas has successfully applied the given function (some columns can be dropped)

To demonstrate transformation in action, let's start with the following DataFrame:

```
In [24]: # a DataFrame to use for examples
transform_data = pd.DataFrame({ 'Label': ['A', 'C', 'B', 'A', 'C'],
                                'Values': [0, 1, 2, 3, 4],
                                'Values2': [5, 6, 7, 8, 9],
                                'Other': ['foo', 'bar', 'baz',
                                          'fiz', 'buz']},
                               index = list('VWXYZ'))
transform_data
```

```
Out[24]:   Label Other  Values  Values2
V      A    foo      0       5
W      C    bar      1       6
X      B    baz      2       7
Y      A    fiz      3       8
Z      C    buz      4       9
```

Now let's group by the `Label` column:

```
In [25]: # group by label
grouped_by_label = df.groupby('Label')
print_groups(grouped_by_label)
```

```
A
  Label Other  Values  Values2
V      A    foo      0       5
Y      A    fiz      3       8
B
  Label Other  Values  Values2
X      B    baz      2       7
C
  Label Other  Values  Values2
W      C    bar      1       6
Z      C    buz      4       9
```

The following code performs a transformation that applies a function to add 10 to each value:

```
In [26]: # add ten to all values in all columns  
grouped_by_label.transform(lambda x: x + 10)
```

```
Out[26]:    Values  Values2  
V        10      15  
W        11      16  
X        12      17  
Y        13      18  
Z        14      19
```

pandas attempts to apply the function to all the columns, but since `Label` and other columns have string values, the transformation function fails (it will throw an exception). Because of that failure, those two columns are omitted from the result.

The result is also non-grouped, as the grouping structure is removed from the result of a transformation. The resulting object will have an index that matches the original `DataFrame` object's index, in this case, v, w, x, y and z.

Filling missing values with the mean of the group

A common transformation in statistical analysis with grouped data is to replace missing data within each group with the mean of the non-`NaN` values in the group. To demonstrate this, the following code creates a `DataFrame` with a `Label` column with two values (`A` and `B`) and a `values` column containing a series of integers, but with one value replaced with `NaN`. The data is then grouped by the `Label` column:

```
In [27]: # data to demonstrate replacement on NaN
df = pd.DataFrame({ 'Label': list("ABABAB"),
                     'Values': [10, 20, 11, np.nan, 12, 22]})
grouped = df.groupby('Label')
print_grouped(grouped)
```

```
A
  Label  Values
0      A    10.0
2      A    11.0
4      A    12.0
B
  Label  Values
1      B    20.0
3      B     NaN
5      B    22.0
```

The mean of each group can be calculated using `.mean()`:

```
In [28]: # calculate the mean of the two groups
grouped.mean()
```

```
Out[28]:      Values
Label
A        11.0
B        21.0
```

Now suppose that we need the `B` group to have all `NaN` values filled in because others using this data code may have difficulties handling the `NaN` value. This can be performed succinctly with the following:

```
In [29]: # use transform to fill the NaNs with the mean of the group
filled_NaNs = grouped.transform(lambda x: x.fillna(x.mean()))
filled_NaNs
```

```
Out[29]:      Values
0        10.0
1        20.0
2        11.0
3        21.0
4        12.0
5        22.0
```


Calculating normalized z-scores with a transformation

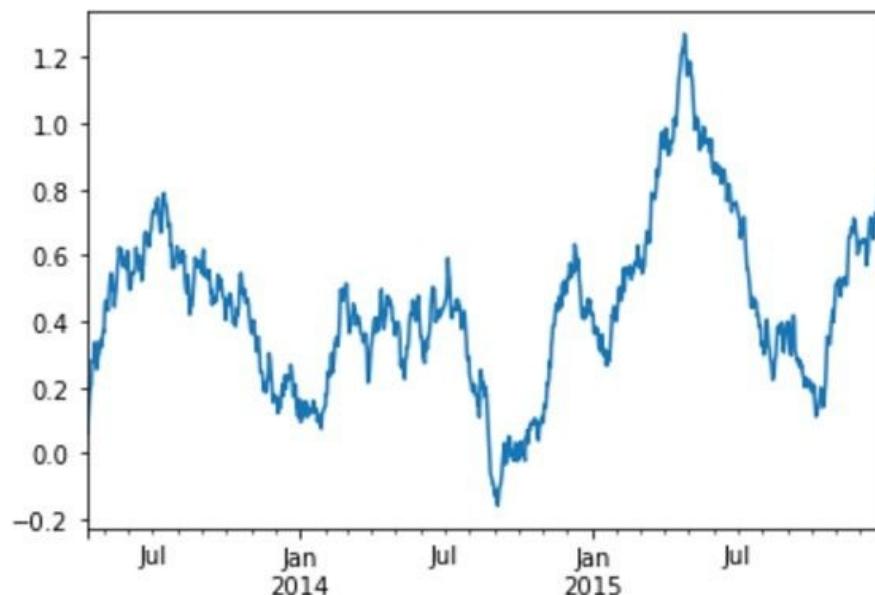
Another common example of transformation is creating normalized z-scores on groups of data. To demonstrate this, we will take a randomly generated series of values using a normal distribution with mean of 0.5 and a standard deviation of 2. The data is indexed by day and a rolling mean is calculated on a 100-day window to generate samples of the mean:

```
In [31]: # generate a rolling mean time series
np.random.seed(123456)
data = pd.Series(np.random.normal(0.5, 2, 365*3),
                 pd.date_range('2013-01-01', periods=365*3))
rolling = data.rolling(
    window=100,
    min_periods=100,
    center=False).mean().dropna()
rolling[:5]
```

```
Out[31]: 2013-04-10    0.073603
2013-04-11    0.057313
2013-04-12    0.089255
2013-04-13    0.133248
2013-04-14    0.175876
Freq: D, dtype: float64
```

The rolling means have the following appearance:

```
In [32]: # visualize the series
rolling.plot();
```



At this point, we would like to standardize the rolling means for each calendar year. The following code

groups the data by year and reports the existing mean and standard deviation of each group:

```
In [33]: # calculate mean and std by year
group_key = lambda x: x.year
groups = rolling.groupby(group_key)
groups.agg([np.mean, np.std])
```

```
Out[33]:      mean      std
2013  0.454233  0.171988
2014  0.286502  0.182040
2015  0.599447  0.275786
```

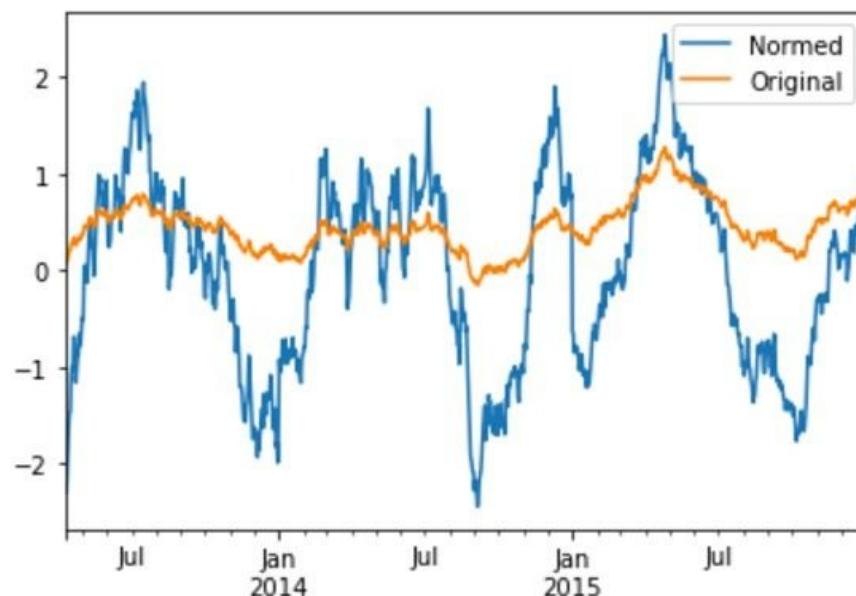
To perform the standardization, this code defines a z-score function, applies it as a transform to each group, and reports on the new mean and standard deviation:

```
In [34]: # normalize to the z-score
zscore = lambda x: (x - x.mean()) / x.std()
normed = rolling.groupby(groupkey).transform(zscore)
normed.groupby(groupkey).agg([np.mean, np.std])
```

```
Out[34]:      mean  std
2013 -3.172066e-17  1.0
2014 -1.881296e-15  1.0
2015 -1.492261e-15  1.0
```

And we can also compare the original and transformed data:

```
In [35]: # plot original vs normalize
compared = pd.DataFrame({ 'Original': rolling,
                           'Normed': normed })
compared.plot();
```



Filtering groups from aggregation

Groups of data can be selectively dropped from processing using `.filter()`. This method is supplied a function that can be used to make group-level decisions on whether the entire group is included in the result after the combination. The function should return `True` if the group is to be included in the result and `False` to exclude it.

We will examine several scenarios using the following data:

```
In [36]: # data for our examples
df = pd.DataFrame({'Label': list('AABCCC'),
                   'Values': [1, 2, 3, 4, np.nan, 8]})
df
```

```
Out[36]:   Label  Values
0      A    1.0
1      A    2.0
2      B    3.0
3      C    4.0
4      C    NaN
5      C    8.0
```

The first demonstration will drop groups that do not have a minimum number of items. Specifically, they will be dropped if they only have one item or less:

```
In [37]: # drop groups with one or fewer non-NaN values
f = lambda x: x.Values.count() > 1
df.groupby('Label').filter(f)
```

```
Out[37]:   Label  Values
0      A    1.0
1      A    2.0
3      C    4.0
4      C    NaN
5      C    8.0
```

The following example will omit groups that have any `NaN` values:

```
In [38]: # drop any groups with NaN values
f = lambda x: x.Values.isnull().sum() == 0
df.groupby('Label').filter(f)
```

```
Out[38]:   Label  Values
0      A    1.0
1      A    2.0
2      B    3.0
```

The next example will only select groups that have a mean that is greater than 2.0, the mean of the entire data set (basically, this selects groups of data that have an exceptional behavior as compared to the whole):

```
In [39]: # select groups with a mean of 2.0 or greater
grouped = df.groupby('Label')
group_mean = grouped.mean().mean()
f = lambda x: abs(x.Values.mean() - group_mean) > 2.0
df.groupby('Label').filter(f)
```

```
Out[39]:   Label  Values
3      C    4.0
4      C    NaN
5      C    8.0
```

Summary

In this chapter, we examined various techniques for grouping and analyzing groups of data with pandas. An introduction to the split-apply-combine pattern was given along with an overview of how this pattern is implemented in pandas. Then we learned how to split data into groups based on data in both columns and in the levels of an index. We then examined how to process data within each group using aggregation functions and transformations. We finished with a quick examination of how to filter groups of data based on their contents.

In the next chapter, we will dive deep into one of the most powerful and robust capabilities of pandas - modeling of time series data.

Time-Series Modelling

A **time-series** is a measurement of one or more variables over a period of time and at a specific interval. Once a time-series is captured, analysis is often performed to identify patterns in the series, in essence, determining what is happening as time goes by. This ability to analyze time-series data is essential in the modern world, be it in order to analyze financial information or to monitor exercise on a wearable device and match your exercises to goals and diet.

pandas provides extensive abilities for modeling time-series data. In this chapter, we will examine many of these capabilities, including:

- Creating time series with specific frequencies
- Representation of dates, time, and intervals
- Representing a point in time with a Timestamp
- Using a Timedelta to represent a time interval
- Indexing using DatetimeIndex
- Creating time-series with specific frequencies
- Representing data intervals with date offsets
- Anchoring periods to specific days of the week, month, quarter, or year
- Modelling an interval of time with a Period
- Indexing using the PeriodIndex
- Handling holidays with calendars
- Normalizing timestamps using time zones
- Shifting and lagging a time-series
- Performing frequency conversion on a time-series
- Up and down resampling of a time-series
- Performing rolling window operations on a time-series

Setting up the IPython notebook

To utilize the examples in this chapter, we will need to include the following imports and settings:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set formattign options
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```


Representation of dates, time, and intervals

To begin to understand time-series data we need to first examine how pandas represents dates, time, and intervals of time. pandas provides extensive built-in facilities to represent these concepts as the representations of these concepts are not implemented by Python or NumPy robustly enough to handle the many concepts needed to process time-series data.

Some of the additional capabilities include being able to transform data across different frequencies and to apply different calendars to take into account things such as business days and holidays in financial calculations.

The datetime, day, and time objects

The `datetime` object is part of the `datetime` library and not a part of pandas. This class can be utilized to construct objects representing several common patterns such as a fixed point in time using a date and time, or simply a day without a time component, or a time without a date component.

The `datetime` objects do not have the accuracy needed for much of the mathematics involved in extensive calculations on time-series data. However, they are commonly used to initialize pandas objects, with pandas converting them into pandas timestamp objects behind the scenes. Therefore, they are still worth a brief mention here as they will be used frequently during initialization.

A `datetime` object can be initialized using a minimum of three parameters representing year, month, and day:

```
In [2]: # datetime object for Dec 15 2014  
        datetime(2014, 12, 15)  
  
Out[2]: datetime.datetime(2014, 12, 15, 0, 0)
```

The result has defaulted the hour and minute values to `0`. The hour and minute components can also be specified with two more values to the constructor. The following creates a `datetime` object that also specifies 5:30 pm:

```
In [3]: # specific date and also with a time of 5:30 pm  
        datetime(2014, 12, 15, 17, 30)  
  
Out[3]: datetime.datetime(2014, 12, 15, 17, 30)
```

The current date and time can be determined using the `datetime.now()` function which retrieves the local date and time:

```
In [4]: # get the local "now" (date and time)  
        # can take a timezone, but that's not demonstrated here  
        datetime.now()  
  
Out[4]: datetime.datetime(2017, 6, 21, 10, 13, 59, 303740)
```

A `datetime.date` object represents a specific day (it has no time component). It can be created by passing a `datetime` object to the constructor of `datetime`:

```
In [5]: # a date without time can be represented  
        # by creating a date using a datetime object  
        datetime.date(datetime(2014, 12, 15))  
  
Out[5]: datetime.date(2014, 12, 15)
```

And you can retrieve the current local date using the following:

```
In [6]: # get just the current date  
        datetime.now().date()  
  
Out[6]: datetime.date(2017, 6, 21)
```

A `time` without a date component can be created using the `datetime.time` object and by passing a `datetime`

object to its constructor:

```
In [7]: # get just a time from a datetime  
datetime.time(datetime(2014, 12, 15, 17, 30))  
  
Out[7]: datetime.time(17, 30)
```

The current local time can be retrieved using the following:

```
In [8]: # get the current local time  
datetime.now().time()  
  
Out[8]: datetime.time(10, 13, 59, 525234)
```


Representing a point in time with a Timestamp

The representation of date and time in pandas is performed using the `pandas.tslib.Timestamp` class. A pandas `Timestamp` is based on the `datetime64` dtype and has higher precision than the Python `datetime` object. In pandas, `Timestamp` objects are generally interchangeable with `datetime` objects, so you can typically use them wherever you may use `datetime` objects.

You can create a `Timestamp` object using `pd.Timestamp` (a shortcut for `pandas.tslib.Timestamp`) and by passing a string representing a date, time, or date and time:

```
In [9]: # a timestamp representing a specific date
pd.Timestamp('2014-12-15')

Out[9]: Timestamp('2014-12-15 00:00:00')
```

A time element can also be specified:

```
In [10]: # a timestamp with both date and time
pd.Timestamp('2014-12-15 17:30')

Out[10]: Timestamp('2014-12-15 17:30:00')
```

A `Timestamp` can also be created using just a time, which will default to also assigning the current local date:

```
In [11]: # timestamp with just a time
# which adds in the current local date
pd.Timestamp('17:30')

Out[11]: Timestamp('2017-06-21 17:30:00')
```

The following demonstrates how to retrieve the current date and time using `Timestamp`:

```
In [12]: # get the current date and time (now)
pd.Timestamp("now")

Out[12]: Timestamp('2017-06-21 10:13:59.788432')
```

As a pandas user you will not normally create `Timestamp` objects directly. Many of the pandas functions that use dates and times will allow you to pass in a `datetime` object or a text representation of a date/time and the functions will perform the conversion internally.

Using a Timedelta to represent a time interval

To represent a difference in time we will use the pandas `Timedelta` object. These are common as results of determining the duration between two dates or to calculate the date at a specific interval of time from another date and/or time.

To demonstrate a `Timedelta`, the following uses a `timedelta` object to calculate a one-day increase in the time from the specified date:

```
In [13]: # what is one day from 2014-11-30?  
today = datetime(2014, 11, 30)  
tomorrow = today + pd.Timedelta(days=1)  
tomorrow  
  
Out[13]: datetime.datetime(2014, 12, 1, 0, 0)
```

The following demonstrates how to calculate how many days there are between two dates:

```
In [14]: # how many days between these two dates?  
date1 = datetime(2014, 12, 2)  
date2 = datetime(2014, 11, 28)  
date1 - date2  
  
Out[14]: datetime.timedelta(4)
```


Introducing time-series data

pandas excels in manipulating time-series data. This is very likely because of its origins in processing financial information. These abilities have been continuously refined over all of its versions to progressively increase its capabilities for time-series manipulation.

Indexing using DatetimeIndex

The core of the time-series functionality in pandas revolves around the use of specialized indexes that represent measurements of data at one or more timestamps. These indexes in pandas are referred to as `DatetimeIndex` objects. These are incredibly powerful objects, and they allow us to automatically align data based on dates and times.

There are several ways to create `DatetimeIndex` objects in pandas. The following creates a `DateTimedelta` by passing a list of `datetime` objects to a `Series`:

```
In [15]: # create a very simple time-series with two index labels
          # and random values
dates = [datetime(2014, 8, 1), datetime(2014, 8, 2)]
ts = pd.Series(np.random.randn(2), dates)
ts

Out[15]: 2014-08-01    -1.067196
2014-08-02     0.106145
dtype: float64
```

This `Series` has taken the `datetime` objects and constructed a `DatetimeIndex` from the date values. Each value of that index is a `Timestamp` object.

The following verifies the type of the index and the types of the labels in the index:

```
In [16]: # what is the type of the index?
type(ts.index)

Out[16]: pandas.core.indexes.datetimes.DatetimeIndex

In [17]: # and we can see it is a collection of timestamps
type(ts.index[0])

Out[17]: pandas._libs.tslib.Timestamp
```

It is not required that you pass `datetime` objects in the list to create a time-series. The `Series` object is smart enough to recognize that a string represents `datetime` and does the conversion for you. The following is equivalent to the previous example:

```
In [18]: # create from just a list of dates as strings!
np.random.seed(123456)
dates = ['2014-08-01', '2014-08-02']
ts = pd.Series(np.random.randn(2), dates)
ts

Out[18]: 2014-08-01    0.469112
2014-08-02   -0.282863
dtype: float64
```

pandas provides a utility function in `pd.to_datetime()` which takes a sequence of similar- or mixed-type objects which pandas attempts to convert into `Timestamp` objects and those into a `DatetimeIndex`. If an object in the sequence cannot be converted, then pandas will create a `NAT` value which means not-a-time:

```
In [19]: # convert a sequence of objects to a DatetimeIndex
dti = pd.to_datetime(['Aug 1, 2014',
                     '2014-08-02',
                     '2014.8.3',
                     None])
for l in dti: print (l)

2014-08-01 00:00:00
2014-08-02 00:00:00
2014-08-03 00:00:00
NaT
```

Be careful, as the `pd.to_datetime()` function will throw an exception if it cannot convert a value to a `Timestamp`:

```
In [20]: # this is a list of objects, not timestamps...
# Throws an error in 0.20.1
# pd.to_datetime(['Aug 1, 2014', 'foo'])
```

To force the function to convert to dates instead of throwing an exception, you can use the `errors="coerce"` parameter. Values that cannot be converted will then assign `NaT` in the resulting index:

```
In [21]: # force the conversion, NaT for items that dont work
pd.to_datetime(['Aug 1, 2014', 'foo'], errors="coerce")

Out[21]: DatetimeIndex(['2014-08-01', 'NaT'], dtype='datetime64[ns]', freq=None)
```

A range of timestamps with a specific frequency can be easily created using the `pd.date_range()` function. The following creates a `Series` object from `DatetimeIndex` of 10 consecutive days:

```
In [22]: # create a range of dates starting at a specific date
# and for a specific number of days, creating a Series
np.random.seed(123456)
periods = pd.date_range('8/1/2014', periods=10)
date_series = pd.Series(np.random.randn(10), index=periods)
date_series

Out[22]: 2014-08-01    0.469112
2014-08-02   -0.282863
2014-08-03   -1.509059
2014-08-04   -1.135632
2014-08-05    1.212112
2014-08-06   -0.173215
2014-08-07    0.119209
2014-08-08   -1.044236
2014-08-09   -0.861849
2014-08-10   -2.104569
Freq: D, dtype: float64
```

A `DatetimeIndex` can be used for various index operations such as data alignment, selection, and slicing. The following demonstrates slicing based by position:

```
In [23]: # slice by location
subset = date_series[3:7]
subset

Out[23]: 2014-08-04   -1.135632
2014-08-05    1.212112
2014-08-06   -0.173215
2014-08-07    0.119209
Freq: D, dtype: float64
```

To demonstrate alignment, we will use the following `Series` created with the index of the subset we just created:

```
In [24]: # a Series to demonstrate alignment
s2 = pd.Series([10, 100, 1000, 10000], subset.index)
s2

Out[24]: 2014-08-04      10
2014-08-05     100
2014-08-06    1000
2014-08-07  10000
Freq: D, dtype: int64
```

When we add `s2` and `date_series`, alignment will be performed, returning `NaN` where items do not align. The value at each index label will be the sum of the values found at the same labels:

```
In [25]: # demonstrate alignment by date on a subset of items
date_series + s2

Out[25]: 2014-08-01      NaN
2014-08-02      NaN
2014-08-03      NaN
2014-08-04    8.864368
2014-08-05  101.212112
2014-08-06  999.826785
2014-08-07 10000.119209
2014-08-08      NaN
2014-08-09      NaN
2014-08-10      NaN
Freq: D, dtype: float64
```

Items in a `Series` with a `DatetimeIndex` can be retrieved using a string representing a date instead of having to specify a `datetime` object:

```
In [26]: # lookup item by a string representing a date
date_series['2014-08-05']

Out[26]: 1.2121120250208506
```

`DatetimeIndex` can also be sliced using a string representing dates:

```
In [27]: # slice between two dates specified by string representing dates
date_series['2014-08-05':'2014-08-07']

Out[27]: 2014-08-05    1.212112
2014-08-06   -0.173215
2014-08-07    0.119209
Freq: D, dtype: float64
```

Another convenient feature of pandas is that `DatetimeIndex` can be sliced using partial date specifications. As an example, the following code creates a `Series` object with dates spanning two years, and then selects only those items of the year 2013:

```
In [28]: # a two year range of daily data in a Series
# only select those in 2013
s3 = pd.Series(0, pd.date_range('2013-01-01', '2014-12-31'))
s3['2013']

Out[28]: 2013-01-01    0
2013-01-02    0
2013-01-03    0
2013-01-04    0
2013-01-05    0
..
2013-12-27    0
2013-12-28    0
2013-12-29    0
2013-12-30    0
2013-12-31    0
Freq: D, Length: 365, dtype: int64
```

It is also possible to select items in a specific year and month. The following selects the items in August 2014:

```
In [29]: # 31 items for May 2014
s3['2014-05']

Out[29]: 2014-05-01    0
          2014-05-02    0
          2014-05-03    0
          2014-05-04    0
          2014-05-05    0
          ..
          2014-05-27    0
          2014-05-28    0
          2014-05-29    0
          2014-05-30    0
          2014-05-31    0
Freq: D, Length: 31, dtype: int64
```

This also works with slices. The following returns items in August and September, 2014:

```
In [30]: # items between two months
s3['2014-08':'2014-09']

Out[30]: 2014-08-01    0
          2014-08-02    0
          2014-08-03    0
          2014-08-04    0
          2014-08-05    0
          ..
          2014-09-26    0
          2014-09-27    0
          2014-09-28    0
          2014-09-29    0
          2014-09-30    0
Freq: D, Length: 61, dtype: int64
```


Creating time-series with specific frequencies

Time-series data can be created on intervals other than daily frequency. Different frequencies can be generated with `pd.date_range()` by utilizing the `freq` parameter. This parameter defaults to a value of 'D' which represents daily frequency.

To demonstrate alternative frequencies, the following creates a `DatetimeIndex` with 1-minute intervals by specifying `freq='T'`:

```
In [31]: # generate a Series at one minute intervals
np.random.seed(123456)
bymin = pd.Series(np.random.randn(24*60*90),
                  pd.date_range('2014-08-01',
                                '2014-10-29 23:59',
                                freq='T'))
bymin[:5]

Out[31]: 2014-08-01 00:00:00    0.469112
2014-08-01 00:01:00   -0.282863
2014-08-01 00:02:00   -1.509059
2014-08-01 00:03:00   -1.135632
2014-08-01 00:04:00    1.212112
Freq: T, dtype: float64
```

This time-series allows us to slice at a finer resolution. The following slices at the minute level:

```
In [32]: # slice down to the minute
bymin['2014-08-01 00:02':'2014-08-01 00:07']

Out[32]: 2014-08-01 00:02:00   -1.509059
2014-08-01 00:03:00   -1.135632
2014-08-01 00:04:00    1.212112
2014-08-01 00:05:00   -0.173215
2014-08-01 00:06:00    0.119209
2014-08-01 00:07:00   -1.044236
Freq: T, dtype: float64
```

The following table lists the possible frequency values:

Alias	Description
B	Business day frequency
C	Custom business day frequency
D	Calendar day frequency (the default)
W	Weekly frequency
M	Month end frequency

BM	Business month end frequency
CBM	Custom business month end frequency
MS	Month start frequency
BMS	Business month start frequency
CBMS	Custom business month start frequency
Q	Quarter end frequency
BQ	Business quarter frequency
QS	Quarter start frequency
BQS	Business quarter start frequency
A	Year-end frequency
BA	Business year-end frequency
AS	Year start frequency
BAS	Business year start frequency
H	Hourly frequency
T	Minute-by-minute frequency
S	Second-by-second frequency

L	Milliseconds
U	Microseconds

You can use the '`B`' frequency to create a time-series that uses only business days:

We can see that two days were skipped as they were on the weekend.

A range can be created starting at a particular date and time, with a specific frequency, and for a specific number of periods using the `periods` parameter. The following creates a 5-item `DatetimeIndex` starting at 2014-08-01 12:10:01 and at 1-second intervals:

Calculating new dates using offsets

Frequencies in pandas are represented using date offsets. We have touched on this concept at the beginning of the chapter when discussing `Timedelta` objects. pandas extends the capabilities of these using the concept of `DateOffset` objects. They are objects which represent knowledge of how to integrate time offsets and frequencies relative to `DatetimeIndex` objects.

Representing data intervals with date offsets

`DatetimeIndex` objects are created at various frequencies by using passing frequency strings such as '`M`', '`W`', and '`BM`' using the `freq` parameter of `pd.date_range()`. Under the hood, these frequency strings are translated into an instance of the pandas `DateOffset` object.

A `DateOffset` represents a regular frequency increment. Specific date offset logic, such as "month", "business day", or "hour", is represented in pandas with various subclasses of `DateOffset`. A `DateOffset` provides pandas with the intelligence to be able to determine how to calculate a specific interval of time from a reference date and time. This provides the user of pandas much greater flexibility in representing date/time offsets than just using a fixed numeric interval

A useful and practical example is calculating the next day of business. This is not simply determined by adding one day to a `datetime`. If the date represents a Friday, the next business day in the US financial market is not Saturday but Monday. And in some cases, one business day from a Friday may actually be Tuesday if Monday is a holiday. pandas gives us all the tools required to handle these types of tricky scenarios.

Let's examine this in action by generating a date range using '`B`' as the frequency:

```
In [35]: # get all business days between and inclusive of these two dates
dti = pd.date_range('2014-08-29', '2014-09-05', freq='B')
dti.values

Out[35]: array(['2014-08-29T00:00:00.000000000', '2014-09-01T00:00:00.000000000',
   '2014-09-02T00:00:00.000000000', '2014-09-03T00:00:00.000000000',
   '2014-09-04T00:00:00.000000000', '2014-09-05T00:00:00.000000000'],
  dtype='datetime64[ns]')
```

This time-series has omitted `2014-08-30` and `2014-08-31` as they are Saturday and Sunday and not considered a business day.

A `DatetimeIndex` has a `.freq` property that represents the frequency of the timestamps in the index:

```
In [36]: # check the frequency is BusinessDay
dti.freq

Out[36]: <BusinessDay>
```

Notice that pandas has created an instance of the `BusinessDay` class to represent the `DateOffset` unit of this index. As mentioned earlier, pandas represents different date offsets with a subclass of the `DateOffset` class. The following are the various built-in date offset classes that are provided by pandas:

Class	Description
<code>DateOffset</code>	Generic offset defaults to one calendar day

BDay	Business day
CDay	Custom business day
Week	One week, optionally anchored on a day of the week
WeekofMonth	The x-th day of the y-th week of each month
LastWeekofMonth	The x-th day of the last week of each month
MonthEnd	Calendar month end
MonthBegin	Calendar month start
BMonthEnd	Business month end
BMonthBegin	Business month start
CBMonthEnd	Custom business month end
CBMonthBegin	Custom business month start
QuarterEnd	Quarter end
QuarterBegin	Quarter start
BQuarterEnd	Business quarter end
BQuarterBegin	Business quarter start
FYS253Quarter	Retail (52-53 week) quarter

YearEnd	Calendar year end
YearBegin	Calendar year start
BYearEnd	Business quarter end
BYearBegin	Business quarter start
FYS253	Retail (52-53 week) year
Hour	One hour
Minute	One minute
Second	One second
Milli	One millisecond
Micro	One microsecond

pandas uses this strategy of using `DateOffset` and its specializations to codify logic to calculate the next day. This makes using these objects very flexible as well as powerful. `DateOffset` objects can be used in various scenarios:

- They can be added or subtracted to obtain a shifted date
- They can be multiplied by an integer (positive or negative) so that the increment will be applied multiple times
- They have `rollforward` and `rollback` methods to move a date forward or backward to the next or previous "offset date"

`DateOffset` objects can be created by passing them a `datetime` object that represents a fixed duration of time or using a number of keyword arguments. Keyword arguments fall into two general categories. The first category is keywords that represent absolute dates: year, month, day, hour, minute, second, and microsecond. The second category represents relative durations and can be negative values: years, months, weeks, day, hours, minutes, seconds, and microseconds.

The following creates a 1-day offset and adds it to `datetime`:

```
In [37]: # calculate a one day offset from 2014-8-29
d = datetime(2014, 8, 29)
do = pd.DateOffset(days = 1)
d + do

Out[37]: Timestamp('2014-08-30 00:00:00')
```

The following calculates the next business day from a given date:

```
In [38]: # import the data offset types
from pandas.tseries.offsets import *
# calculate one business day from 2014-8-31
d + BusinessDay()

Out[38]: Timestamp('2014-09-01 00:00:00')
```

Multiple units of a specific `DateOffset` can be represented via multiplication:

```
In [39]: # determine 2 business days from 2014-8-29
d + 2 * BusinessDay()

Out[39]: Timestamp('2014-09-02 00:00:00')
```

The following demonstrates using a `BMonthEnd` object to calculate the last business day of a month from a given date (in this case, 2014-09-02):

```
In [40]: # what is the next business month end
# from a specific date?
d + BMonthEnd()

Out[40]: Timestamp('2014-09-30 00:00:00')
```

The following uses the `BMonthEnd` objects `.rollforward()` method to calculate the next month end:

```
In [41]: # calculate the next month end by
# rolling forward from a specific date
BMonthEnd().rollforward(datetime(2014, 9, 15))

Out[41]: Timestamp('2014-09-30 00:00:00')
```

Several of the offset classes can be parameterized to provide finer control of the offset behavior. As an example, the following calculates the date of the Tuesday (`weekday = 1`) in the week prior to 2014-08-31:

```
In [42]: # calculate the date of the Tuesday previous
# to a specified date
d - Week(weekday = 1)

Out[42]: Timestamp('2014-08-26 00:00:00')
```


Anchored offsets

Anchored offsets are frequencies that represent a given frequency and begin at a specific point such as a specific day of the week, month, or year. Anchored offsets use a specific shorthand nomenclature. As an example, the following strings specify a specific day of the week:

Alias	Description
W-SUN	Weekly on Sunday (same as 'W')
W-MON	Weekly on Monday
W-TUE	Weekly on Tuesday
W-WED	Weekly on Wednesday
W-THU	Weekly on Thursday
W-FRI	Weekly on Friday
W-SAT	Weekly on Saturday

As an example, the following generates an index that consists of the dates of all Wednesdays between the two specified dates:

```
In [43]: # calculate all wednesdays between 2014-06-01
# and 2014-08-31
wednesdays = pd.date_range('2014-06-01',
                           '2014-07-31', freq="W-WED")
wednesdays.values

Out[43]: array(['2014-06-04T00:00:00.000000000', '2014-06-11T00:00:00.000000000',
               '2014-06-18T00:00:00.000000000', '2014-06-25T00:00:00.000000000',
               '2014-07-02T00:00:00.000000000', '2014-07-09T00:00:00.000000000',
               '2014-07-16T00:00:00.000000000', '2014-07-23T00:00:00.000000000',
               '2014-07-30T00:00:00.000000000'], dtype='datetime64[ns]')
```

Anchored offsets can also be created using the annual and quarterly frequencies. These frequency anchors are of the general form [B][A|Q][S]-[MON], where B (business days) and s (start of period instead end) are

optional, `A` is for annual or `Q` for quarterly, and `MON` is the three-digit abbreviation for the month (JAN, FEB, ...).

To demonstrate, the following generates the business dates for quarter end in the year 2014 with the year anchored at the end of June:

```
In [44]: # what are all of the business quarterly end
# dates in 2014?
qends = pd.date_range('2014-01-01', '2014-12-31',
                      freq='BQS-JUN')
qends.values

Out[44]: array(['2014-03-03T00:00:00.000000000', '2014-06-02T00:00:00.000000
000',
               '2014-09-01T00:00:00.000000000', '2014-12-01T00:00:00.000000
000'], dtype='datetime64[ns]')
```


Representing durations of time using Period

Many useful analysis operations on time-series data require that events within a specific time interval be analyzed. A simple example would be to determine how many financial transactions occurred in a specific period.

These types of analyses can be performed using `Timestamp` and `DateOffset`, where the bounds are calculated and then items filtered based on these bounds. However, this becomes cumbersome when you need to deal with events that must be grouped into multiple periods of time, as you start to need to manage sets of the `Timestamp` and `DateOffset` objects.

To facilitate these types of data organization and calculations, pandas makes intervals of time a formal construct using the `Period` class. pandas also formalizes series of `Period` objects using `PeriodIndex`, which provides capabilities of aligning data items based on the indexes associated period objects.

Modelling an interval of time with a Period

pandas formalizes the concept of an interval of time using a `Period` object. A `Period` allows you to specify durations based on frequencies such as daily, weekly, monthly, annually, quarterly, and so on, and it will provide a specific start and end `Timestamp` representing the specific bounded interval of time.

A `Period` is created using a timestamp and a frequency, where the timestamp represents the anchor used as a point of reference and the frequency is the duration of time. To demonstrate, the following creates a period representing one month and which is anchored in August 2014:

```
In [45]: # create a period representing a month of time
# starting in August 2014
aug2014 = pd.Period('2014-08', freq='M')
aug2014
```

```
Out[45]: Period('2014-08', 'M')
```

A `Period` has `start_time` and `end_time` properties that inform us about the derived start and end times:

```
In [46]: # examine the start and end times of this period
aug2014.start_time, aug2014.end_time
```

```
Out[46]: (Timestamp('2014-08-01 00:00:00'), Timestamp('2014-08-31 23:59:59.999999'))
```

As we specified a period that is August 2014, pandas determines the anchor (`start_time`) and then calculates `end_time` based on the specified frequency. In this case, it calculates one month from `start_time` and returns the last time unit prior to that value.

Mathematical operations are overloaded on `Period` to calculate another `Period` based on the given value. The following creates a new `Period` object based on variable `aug2014` which is shifted by `1` unit of its represented frequency (which is one month):

```
In [47]: # calculate the period that is one frequency
# unit of the aug2014 period further along in time
# This happens to be September 2014
sep2014 = aug2014 + 1
sep2014
```

```
Out[47]: Period('2014-09', 'M')
```

The concept of the shift is very important and powerful. The addition of `1` to this `Period` object informs it to shift in time one positive unit of whatever frequency is represented by the object. In this case, it shifts the period one month forward to September 2014.

If we examine the start and end times represented in the `sep2014` variable, we see that pandas has gone through the effort of determining the correct dates representing the entirety of September 2014:

```
In [48]: sep2014.start_time, sep2014.end_time
```

```
Out[48]: (Timestamp('2014-09-01 00:00:00'), Timestamp('2014-09-30 23:59:59.999999'))
```

Note that `Period` had the intelligence to know that September is 30 days and not 31. This is part of the intelligence behind the `Period` object that saves us a lot of coding, helping us to solve many difficult date

management problems.

Indexing using the PeriodIndex

A series of `Period` objects can be combined into a special form of pandas index known as `PeriodIndex`. A `PeriodIndex` index is useful for being able to associate data to specific intervals of time and with being able to slice and perform analysis on the events in each interval.

The following creates a `PeriodIndex` consisting of 1-month intervals for the year 2013:

```
In [49]: # create a period index representing all monthly boundaries in 2013
mp2013 = pd.period_range('1/1/2013', '12/31/2013', freq='M')
mp2013

Out[49]: PeriodIndex(['2013-01', '2013-02', '2013-03', '2013-04',
 '2013-05', '2013-06', '2013-07', '2013-08',
 '2013-09', '2013-10', '2013-11', '2013-12'],
 dtype='period[M]', freq='M')
```

`PeriodIndex` differs from `DatetimeIndex` in that the index labels are `Period` objects. The following prints the start and end times for all the `Period` objects in the index:

```
In [50]: # loop through all period objects in the index
# printing start and end time for each
for p in mp2013:
    print ("{} {} {}".format(p.start_time, p.end_time))

2013-01-01 00:00:00 2013-01-31 23:59:59.999999999
2013-02-01 00:00:00 2013-02-28 23:59:59.999999999
2013-03-01 00:00:00 2013-03-31 23:59:59.999999999
2013-04-01 00:00:00 2013-04-30 23:59:59.999999999
2013-05-01 00:00:00 2013-05-31 23:59:59.999999999
2013-06-01 00:00:00 2013-06-30 23:59:59.999999999
2013-07-01 00:00:00 2013-07-31 23:59:59.999999999
2013-08-01 00:00:00 2013-08-31 23:59:59.999999999
2013-09-01 00:00:00 2013-09-30 23:59:59.999999999
2013-10-01 00:00:00 2013-10-31 23:59:59.999999999
2013-11-01 00:00:00 2013-11-30 23:59:59.999999999
2013-12-01 00:00:00 2013-12-31 23:59:59.999999999
```

pandas has determined the start and end of each month while taking into account the actual number of days in each specific month.

Using `PeriodIndex`, we can construct a `Series` object using it as the index and associate a value to each `Period` in the index:

```
In [51]: # create a Series with a PeriodIndex
np.random.seed(123456)
ps = pd.Series(np.random.randn(12), mp2013)
ps[:5]

Out[51]: 2013-01    0.469112
2013-02   -0.282863
2013-03   -1.509059
2013-04   -1.135632
2013-05    1.212112
Freq: M, dtype: float64
```

We now have a time-series where the value at a specific index label represents a measurement that spans a period of time. An example of a series like this would be the average value of a security in a given month instead of at a specific time. This becomes very useful when we perform the resampling of the time series to another frequency.

Like `DatetimeIndex`, `PeriodIndex` can be used to index values using `Period`, a string representing a period or

partial period specification. To demonstrate, we will create another series similar to the previous one but spanning two years, 2013 and 2014:

```
In [52]: # create a Series with a PeriodIndex and which
# represents all calendar month periods in 2013 and 2014
np.random.seed(123456)
ps = pd.Series(np.random.randn(24),
               pd.period_range('1/1/2013',
                               '12/31/2014', freq='M'))
ps
```

```
Out[52]: 2013-01    0.469112
2013-02   -0.282863
2013-03   -1.509059
2013-04   -1.135632
2013-05    1.212112
...
2014-08   -1.087401
2014-09   -0.673690
2014-10    0.113648
2014-11   -1.478427
2014-12    0.524988
Freq: M, Length: 24, dtype: float64
```

Individual values can be selected using the specific index label using either a `Period` object or a string representing a period. The following demonstrates how to use a string representation:

```
In [53]: # get value for period represented with 2014-06
ps['2014-06']
```

```
Out[53]: 0.567020349793672
```

Partial specifications can also be used, such as the following which retrieves all values just for periods in 2014:

```
In [54]: # get values for all periods in 2014
ps['2014']
```

```
Out[54]: 2014-01    0.721555
2014-02   -0.706771
2014-03   -1.039575
2014-04    0.271860
2014-05   -0.424972
...
2014-08   -1.087401
2014-09   -0.673690
2014-10    0.113648
2014-11   -1.478427
2014-12    0.524988
Freq: M, Length: 12, dtype: float64
```

A `PeriodIndex` can also be sliced. The following retrieves all values for periods between (and inclusive of) March and June 2014:

```
In [55]: # all values between (and including) March and June 2014
ps['2014-03':'2014-06']
```

```
Out[55]: 2014-03   -1.039575
2014-04    0.271860
2014-05   -0.424972
2014-06    0.567020
Freq: M, dtype: float64
```


Handling holidays using calendars

Earlier, when we calculated the next business day from August 29, 2014, we were told by pandas that this date is September 1, 2014. This is actually not correct in the United States: September 1, 2014 is a US federal holiday and banks and exchanges are closed on this day. The reason for this is that pandas uses a specific default calendar when calculating the next business day, and this default pandas calendar does not include September 1, 2014 as a holiday.

The solution to this issue is to either create a custom calendar (which we will not get into the details of), or use the one custom calendar provided by pandas for just this situation, `usFederalHolidayCalendar`. This custom calendar can then be passed to a `CustomBusinessDay` object that will be used instead of a `BusinessDay` object. The calculation using this `CustomBusinessDay` object will then use the new calendar and take into account the US federal holidays.

The following demonstrates the creation of a `usFederalCalendar` object and how to use it to report the days that it considers holidays:

```
In [56]: # demonstrate using the US federal holiday calendar
# first need to import it
from pandas.tseries.holiday import *
# create it and show what it considers holidays
cal = USFederalHolidayCalendar()
for d in cal.holidays(start='2014-01-01', end='2014-12-31'):
    print (d)
```

```
2014-01-01 00:00:00
2014-01-20 00:00:00
2014-02-17 00:00:00
2014-05-26 00:00:00
2014-07-04 00:00:00
2014-09-01 00:00:00
2014-10-13 00:00:00
2014-11-11 00:00:00
2014-11-27 00:00:00
2014-12-25 00:00:00
```

This calendar can then be used to calculate the next business day from August 29, 2014:

```
In [57]: # create CustomBusinessDay object based on the federal calendar
cbd = CustomBusinessDay(holidays=cal.holidays())

# now calc next business day from 2014-8-29
datetime(2014, 8, 29) + cbd
```

```
Out[57]: Timestamp('2014-09-02 00:00:00')
```

The resulting calculation now takes into account Labor Day not being a business day and returns the correct date of 2014-09-02.

Normalizing timestamps using time zones

Time zone management can be one of the most complicated issues to deal with when working with time-series data. Data is often collected in different systems across the globe using local time, and at some point, it will require coordination with data collected in other time zones.

Fortunately, pandas provides rich support for working with timestamps in different time zones. Under the covers, pandas utilizes the `pytz` and `dateutil` libraries to manage the time zone operations. The `dateutil` support is new as of pandas 0.14.1 and currently only supported for fixed offset and `tzfile` zones. The default library used by pandas is `pytz`, with support for `dateutil` provided for compatibility with other applications.

pandas objects that are time zone-aware support a `.tz` property. By default, pandas objects that are time zone-aware do not utilize a `timezone` object for purposes of efficiency. The following gets the current time and demonstrates that there is no time zone information by default:

```
In [58]: # get the current local time and demonstrate there is no
# timezone info by default
now = pd.Timestamp('now')
now, now.tz is None
```



```
Out[58]: (Timestamp('2017-06-29 12:32:23.256410'), True)
```

 This demonstrates that pandas treats `Timestamp("now")` as UTC by default but without time zone data. This is a good default, but be aware of this. In general, I find that if you are ever collecting data based on the time that will be stored for later access, or collected from multiple data sources, it is best to always localize to UTC.

Likewise, `DatetimeIndex` and its `Timestamp` objects will not have associated time zone information by default:

```
In [59]: # default DatetimeIndex and its Timestamps do not have
# time zone information
rng = pd.date_range('3/6/2012 00:00', periods=15, freq='D')
rng.tz is None, rng[0].tz is None
```



```
Out[59]: (True, True)
```

A list of common time zone names can be retrieved as shown in the following example. If you do a lot with time zone data, these will become very familiar:

```
In [60]: # import common timezones from pytz
from pytz import common_timezones
# report the first 5
common_timezones[:5]
```



```
Out[60]: ['Africa/Abidjan',
'Africa/Accra',
'Africa/Addis_Ababa',
'Africa/Algiers',
'Africa/Asmara']
```

The local UTC time can be found using the following which utilizes the `.tz_localize()` method of `Timestamp`

```
In [61]: # get now, and now localized to UTC
now = Timestamp("now")
local_now = now.tz_localize('UTC')
now, local_now
```



```
Out[61]: (Timestamp('2017-06-29 12:32:23.407610'),
Timestamp('2017-06-29 12:32:23.407610+0000', tz='UTC'))
```

and by passing the `UTC` value:

Any `Timestamp` can be localized to a specific time zone by passing the time zone name to `.tz_localize()`:

```
In [62]: # localize a timestamp to US/Mountain time zone
tstamp = Timestamp('2014-08-01 12:00:00', tz='US/Mountain')
tstamp

Out[62]: Timestamp('2014-08-01 12:00:00-0600', tz='US/Mountain')
```

A `DatetimeIndex` can be created with a specific time zone by using the `tz` parameter of the `pd.date_range()`

```
In [63]: # create a DatetimeIndex using a timezone
rng = pd.date_range('3/6/2012 00:00:00',
                     periods=10, freq='D', tz='US/Mountain')
rng.tz, rng[0].tz

Out[63]: (<DstTzInfo 'US/Mountain' LMT-1 day, 17:00:00 STD>,
           <DstTzInfo 'US/Mountain' MST-1 day, 17:00:00 STD>)
```

method:

It is also possible to construct other time zones explicitly. This model can give you more control over which time zone is used in `.tz_localize()`. The following creates two different `timezone` objects and localizes

```
In [64]: # show use of timezone objects
# need to reference pytz
import pytz
# create an object for two different timezones
mountain_tz = pytz.timezone("US/Mountain")
eastern_tz = pytz.timezone("US/Eastern")
# apply each to 'now'
mountain_tz.localize(now), eastern_tz.localize(now)

Out[64]: (Timestamp('2017-06-29 12:32:23.407610-0600', tz='US/Mountain'),
          Timestamp('2017-06-29 12:32:23.407610-0400', tz='US/Eastern'))
```

a `Timestamp` to each:

Operations on multiple time-series objects will be aligned by `Timestamp` in their index by taking into account the time zone information. To demonstrate this, we will use the following which creates two `Series` objects using the two `DatetimeIndex` objects, each with the same start, periods, and frequency but using

```
In [65]: # create two Series, same start, same periods, same frequencies,
# each with a different timezone
s_mountain = Series(np.arange(0, 5),
                     index=pd.date_range('2014-08-01',
                                         periods=5, freq="H",
                                         tz='US/Mountain'))
s_eastern = Series(np.arange(0, 5),
                   index=pd.date_range('2014-08-01',
                                       periods=5, freq="H",
                                       tz='US/Eastern'))
s_mountain

Out[65]: 2014-08-01 00:00:00-06:00    0
          2014-08-01 01:00:00-06:00    1
          2014-08-01 02:00:00-06:00    2
          2014-08-01 03:00:00-06:00    3
          2014-08-01 04:00:00-06:00    4
          Freq: H, dtype: int64
```

different time zones:

```
In [66]: s_eastern

Out[66]: 2014-08-01 00:00:00-04:00    0
          2014-08-01 01:00:00-04:00    1
          2014-08-01 02:00:00-04:00    2
          2014-08-01 03:00:00-04:00    3
          2014-08-01 04:00:00-04:00    4
          Freq: H, dtype: int64
```

The following demonstrates the alignment of these two `Series` objects by time zone by adding the two together:

```
In [67]: # add the two Series. This only results in three items being aligned
s_eastern + s_mountain
```

Out[67]:	2014-08-01 04:00:00+00:00	NaN
	2014-08-01 05:00:00+00:00	NaN
	2014-08-01 06:00:00+00:00	2.0
	2014-08-01 07:00:00+00:00	4.0
	2014-08-01 08:00:00+00:00	6.0
	2014-08-01 09:00:00+00:00	NaN
	2014-08-01 10:00:00+00:00	NaN
	Freq: H, dtype: float64	

Once a time zone is assigned to an object, that object can be converted to another time zone using the

```
In [68]: # convert s1 from US/Eastern to US/Pacific
s_pacific = s_eastern.tz_convert("US/Pacific")
s_pacific
```

Out[68]:	2014-07-31 21:00:00-07:00	0
	2014-07-31 22:00:00-07:00	1
	2014-07-31 23:00:00-07:00	2
	2014-08-01 00:00:00-07:00	3
	2014-08-01 01:00:00-07:00	4
	Freq: H, dtype: int64	

.tz.convert() method:

Now if we add s_pacific to s_mountain, the alignment will force the same result:

```
In [69]: # this will be the same result as s_eastern + s_mountain
# as the timezones still get aligned to be the same
s_mountain + s_pacific
```

Out[69]:	2014-08-01 04:00:00+00:00	NaN
	2014-08-01 05:00:00+00:00	NaN
	2014-08-01 06:00:00+00:00	2.0
	2014-08-01 07:00:00+00:00	4.0
	2014-08-01 08:00:00+00:00	6.0
	2014-08-01 09:00:00+00:00	NaN
	2014-08-01 10:00:00+00:00	NaN
	Freq: H, dtype: float64	

Manipulating time-series data

We will now examine several common operations that are performed on time-series data. These operations entail realigning data, changing the frequency of the samples and their values, and calculating aggregate results on continuously moving subsets of the data to determine the behavior of the values in the data as time changes.

Shifting and lagging

A common operation on time-series data is to shift the values backward and forward in time. The pandas method for this is `.shift()` which will shift values in a `Series` or `DataFrame` a specified number of units of the frequency in the index.

To demonstrate shifting, we will use the following `series`. This `series` has five values, is indexed by date starting at `2014-08-01`, and uses a daily frequency:

```
In [70]: # create a Series to work with
np.random.seed(123456)
ts = Series([1, 2, 2.5, 1.5, 0.5],
            pd.date_range('2014-08-01', periods=5))
ts

Out[70]: 2014-08-01    1.0
          2014-08-02    2.0
          2014-08-03    2.5
          2014-08-04    1.5
          2014-08-05    0.5
          Freq: D, dtype: float64
```

The following shifts the values forward by 1 day:

```
In [71]: # shift forward one day
ts.shift(1)

Out[71]: 2014-08-01    NaN
          2014-08-02    1.0
          2014-08-03    2.0
          2014-08-04    2.5
          2014-08-05    1.5
          Freq: D, dtype: float64
```

pandas has moved the values forward one unit of the index's frequency, which is one day. The index itself remains unchanged. There was no replacement data for `2014-08-01` so it is filled with `NaN`.

A lag is a shift in a negative direction. The following lags the `series` by 2 days:

```
In [72]: # lag two days
ts.shift(-2)

Out[72]: 2014-08-01    2.5
          2014-08-02    1.5
          2014-08-03    0.5
          2014-08-04    NaN
          2014-08-05    NaN
          Freq: D, dtype: float64
```

Index labels `2014-08-04` and `2014-08-05` now have `NaN` values as there were no items to replace.

A common calculation that is performed using a shift is to calculate the percentage daily change in values. This can be performed by dividing a `series` object by its values shifted by 1:

```
In [73]: # calculate daily percentage change
ts / ts.shift(1)

Out[73]: 2014-08-01      NaN
          2014-08-02    2.000000
          2014-08-03    1.250000
          2014-08-04    0.600000
          2014-08-05    0.333333
          Freq: D, dtype: float64
```

Shifts can be performed on different frequencies than that in the index. When this is performed, the index will be modified and the values remain the same. As an example, the following shifts the series forward by one business day:

```
In [74]: # shift forward one business day
ts.shift(1, freq="B")
```



```
Out[74]: 2014-08-04    1.0
          2014-08-04    2.0
          2014-08-04    2.5
          2014-08-05    1.5
          2014-08-06    0.5
          dtype: float64
```

As another example, the following shifts forward by 5 hours:

```
In [75]: # shift forward five hours
ts.tshift(5, freq="H")
```



```
Out[75]: 2014-08-01 05:00:00    1.0
          2014-08-02 05:00:00    2.0
          2014-08-03 05:00:00    2.5
          2014-08-04 05:00:00    1.5
          2014-08-05 05:00:00    0.5
          Freq: D, dtype: float64
```

A time-series can also be shifted using `DateOffset`. The following code shifts the time series forward by 0.5 minutes:

```
In [76]: # shift using a DateOffset
ts.shift(1, DateOffset(minutes=0.5))
```



```
Out[76]: 2014-08-01 00:00:30    1.0
          2014-08-02 00:00:30    2.0
          2014-08-03 00:00:30    2.5
          2014-08-04 00:00:30    1.5
          2014-08-05 00:00:30    0.5
          Freq: D, dtype: float64
```

There is an alternative form of shifting provided by the `.tshift()` method. This method shifts the index labels by the specified units and a frequency specified by the `freq` parameter (which is required). The following code demonstrates this approach by adjusting the index by -1 hour:

```
In [77]: # shift just the index values
ts.tshift(-1, freq='H')
```



```
Out[77]: 2014-07-31 23:00:00    1.0
          2014-08-01 23:00:00    2.0
          2014-08-02 23:00:00    2.5
          2014-08-03 23:00:00    1.5
          2014-08-04 23:00:00    0.5
          Freq: D, dtype: float64
```


Performing frequency conversion on a time-series

Frequency data can be converted in pandas using the `.asfreq()` method of a time-series object. When converting frequency, a new `Series` object with a new `DatetimeIndex` object will be created. The `DatetimeIndex` of the new `Series` object starts at the first `Timestamp` of the original and progresses at the given frequency until the last `Timestamp` of the original. Values will then be aligned into the new `Series`.

To demonstrate, we will use the following time series of consecutive incremental integers mapped into each hour of each day for *August 2014*:

```
In [78]: # create a Series of incremental values
# index by hour through all of August 2014
periods = 31 * 24
hourly = Series(np.arange(0, periods),
                pd.date_range('08-01-2014', freq="2H",
                periods = periods))
hourly[:5]

Out[78]: 2014-08-01 00:00:00    0
2014-08-01 02:00:00    1
2014-08-01 04:00:00    2
2014-08-01 06:00:00    3
2014-08-01 08:00:00    4
Freq: 2H, dtype: int64
```

The following converts this time series to a daily frequency using `.asfreq('D')`:

```
In [79]: # convert to daily frequency
# many items will be dropped due to alignment
daily = hourly.asfreq('D')
daily[:5]

Out[79]: 2014-08-01    0
2014-08-02    12
2014-08-03    24
2014-08-04    36
2014-08-05    48
Freq: D, dtype: int64
```

As data was aligned to the new daily time series from the hourly time series, only values matching the exact days were copied.

If we convert this result back to an hourly frequency, we will see that many of the values are `NaN`:

```
In [80]: # convert back to hourly. Results in many NaNs
# as the new index has many labels that do not
# align from the source
daily.asfreq('H')

Out[80]: 2014-08-01 00:00:00    0.0
2014-08-01 01:00:00    NaN
2014-08-01 02:00:00    NaN
2014-08-01 03:00:00    NaN
2014-08-01 04:00:00    NaN
...
2014-09-30 20:00:00    NaN
2014-09-30 21:00:00    NaN
2014-09-30 22:00:00    NaN
2014-09-30 23:00:00    NaN
2014-10-01 00:00:00    732.0
Freq: H, Length: 1465, dtype: float64
```

The new index has `Timestamp` objects at hourly intervals, so only the timestamps at exact days align with the

daily time series, resulting in 670 `NaN` values.

This default behavior can be changed using the `method` parameter of the `.asfreq()` method. This value can be used for forward fill, reverse fill, or to pad the `NaN` values.

The `ffill` method will forward fill the last known value (pad also does the same):

```
In [81]: # forward fill values
daily.asfreq('H', method='ffill')

Out[81]: 2014-08-01 00:00:00      0
          2014-08-01 01:00:00      0
          2014-08-01 02:00:00      0
          2014-08-01 03:00:00      0
          2014-08-01 04:00:00      0
          ...
          2014-09-30 20:00:00    720
          2014-09-30 21:00:00    720
          2014-09-30 22:00:00    720
          2014-09-30 23:00:00    720
          2014-10-01 00:00:00    732
Freq: H, Length: 1465, dtype: int64
```

The `bfill` method will back fill values from the next known value:

```
In [82]: daily.asfreq('H', method='bfill')

Out[82]: 2014-08-01 00:00:00      0
          2014-08-01 01:00:00     12
          2014-08-01 02:00:00     12
          2014-08-01 03:00:00     12
          2014-08-01 04:00:00     12
          ...
          2014-09-30 20:00:00    732
          2014-09-30 21:00:00    732
          2014-09-30 22:00:00    732
          2014-09-30 23:00:00    732
          2014-10-01 00:00:00    732
Freq: H, Length: 1465, dtype: int64
```


Up and down resampling of a time-series

Frequency conversion provides a basic way to convert the index in a time series to another frequency. Data in the new time series is aligned with the old data and can result in many `NaN` values. This can be partially solved using a fill method, but that is limited in its capabilities to fill with appropriate information.

Resampling differs in that it does not perform a pure alignment. The values placed in the new series can use the same forward and reverse fill options, but they can also be specified using other pandas-provided algorithms or with your own functions.

To demonstrate resampling, we will use the following time series, which represents a random walk of values over a 5-day period:

```
In [83]: # calculate a random walk five days long at one second intervals
# this many items will be needed
count = 24 * 60 * 60 * 5
# create a series of values
np.random.seed(123456)
values = np.random.randn(count)
ws = pd.Series(values)
# calculate the walk
walk = ws.cumsum()
# patch the index
walk.index = pd.date_range('2014-08-01', periods=count, freq="S")
walk
```

```
Out[83]: 2014-08-01 00:00:00    0.469112
2014-08-01 00:00:01    0.186249
2014-08-01 00:00:02   -1.322810
2014-08-01 00:00:03   -2.458442
2014-08-01 00:00:04   -1.246330
...
2014-08-05 23:59:55   456.529763
2014-08-05 23:59:56   456.052131
2014-08-05 23:59:57   455.202981
2014-08-05 23:59:58   454.947362
2014-08-05 23:59:59   456.191430
Freq: S, Length: 432000, dtype: float64
```

Resampling in pandas is accomplished using the `.resample()` method and by passing it a new frequency. To demonstrate this, the following resamples the by-the-second data to by-the-minute. This is a downsampling, as the result has a lower frequency and results in less values:

```
In [84]: # resample to minute intervals
walk.resample("1Min").mean()
```

```
Out[84]: 2014-08-01 00:00:00    -8.718220
2014-08-01 00:01:00    -15.239213
2014-08-01 00:02:00    -9.179315
2014-08-01 00:03:00    -8.338307
2014-08-01 00:04:00    -8.129554
...
2014-08-05 23:55:00    453.773467
2014-08-05 23:56:00    450.857039
2014-08-05 23:57:00    450.078149
2014-08-05 23:58:00    444.637806
2014-08-05 23:59:00    453.837417
Freq: T, Length: 7200, dtype: float64
```

Notice that the first value is -8.718220 whereas the original data had a value of 0.469112. A frequency conversion would have left this value at -8.718220. This is because a resampling does not copy data through alignment. A resampling will actually split the data into buckets of data based on new periods and

then apply a particular operation to the data in each bucket, in this case calculating the mean of the bucket. This can be verified with the following which slices the first minute of data from the walk and calculates its mean:

```
In [85]: # calculate the mean of the first minute of the walk  
walk['2014-08-01 00:00'].mean()  
  
Out[85]: -8.718220052832644
```

In downsampling, as the existing data is put into buckets based on the new intervals, there can often be a question of what values are on each end of the bucket. As an example, should the first interval in the previous resampling be from 2014-08-01 00:00:00 through 2014-08-01 23:59:59, or should it end at 2014-08-04 00:00:00 but start at 2014-08-03 23:59:59?

The default is the former, and it is referred to as a left close. The other scenario that excludes the left value and includes the right is a right close and can be performed by using the `closed='right'` parameter. The following demonstrates this, and notice the slight difference in the intervals and values that result:

```
In [86]: # use a right close  
walk.resample("1Min", closed='right').mean()  
  
Out[86]:  
2014-07-31 23:59:00      0.469112  
2014-08-01 00:00:00     -8.907477  
2014-08-01 00:01:00    -15.280685  
2014-08-01 00:02:00    -9.083865  
2014-08-01 00:03:00    -8.285550  
...  
2014-08-05 23:55:00    453.726168  
2014-08-05 23:56:00    450.849039  
2014-08-05 23:57:00    450.039159  
2014-08-05 23:58:00    444.631719  
2014-08-05 23:59:00    453.955377  
Freq: T, Length: 7201, dtype: float64
```

The decision about whether to use a right or left close is really up to you and your data modeling, but pandas gives you the option for either.

Calculating the mean of each bucket is only one option. The following demonstrates taking the first value in each bucket:

```
In [87]: # resample to 1 minute  
walk.resample("1Min").first()  
  
Out[87]:  
2014-08-01 00:00:00      0.469112  
2014-08-01 00:01:00     -10.886314  
2014-08-01 00:02:00     -13.374656  
2014-08-01 00:03:00     -7.647693  
2014-08-01 00:04:00     -4.482292  
...  
2014-08-05 23:55:00    452.900335  
2014-08-05 23:56:00    450.062374  
2014-08-05 23:57:00    449.582419  
2014-08-05 23:58:00    447.243014  
2014-08-05 23:59:00    446.877810  
Freq: T, Length: 7200, dtype: float64
```

To demonstrate upsampling, we will resample the walk to minutes and then back into seconds:

```
In [88]: # resample to 1 minute intervals, then back to 1 sec
bymin = walk.resample("1Min").mean()
bymin.resample('S').mean()

Out[88]: 2014-08-01 00:00:00      -8.718220
2014-08-01 00:00:01          NaN
2014-08-01 00:00:02          NaN
2014-08-01 00:00:03          NaN
2014-08-01 00:00:04          NaN
...
2014-08-05 23:58:56          NaN
2014-08-05 23:58:57          NaN
2014-08-05 23:58:58          NaN
2014-08-05 23:58:59          NaN
2014-08-05 23:59:00      453.837417
Freq: S, Length: 431941, dtype: float64
```

The upsampling created the index values for the second-by-second data but inserted `NaN` values by default. This default behavior can be modified using the `fill_method` parameter. We saw this when changing frequency with the options of forward and backward filling. These are also available with resampling. The following demonstrates how to use the forward fill:

```
In [89]: # resample to 1 second intervals using forward fill
bymin.resample("S").bfill()

Out[89]: 2014-08-01 00:00:00      -8.718220
2014-08-01 00:00:01     -15.239213
2014-08-01 00:00:02     -15.239213
2014-08-01 00:00:03     -15.239213
2014-08-01 00:00:04     -15.239213
...
2014-08-05 23:58:56     453.837417
2014-08-05 23:58:57     453.837417
2014-08-05 23:58:58     453.837417
2014-08-05 23:58:59     453.837417
2014-08-05 23:59:00     453.837417
Freq: S, Length: 431941, dtype: float64
```

It is also possible to interpolate the missing values using the `.interpolate()` method on the result. This will calculate a linear interpolation between the values existing in the result for all of the `NaN` values created during the resampling:

```
In [90]: # demonstrate interpolating the NaN values
interpolated = bymin.resample("S").interpolate()
interpolated

Out[90]: 2014-08-01 00:00:00      -8.718220
2014-08-01 00:00:01     -8.826903
2014-08-01 00:00:02     -8.935586
2014-08-01 00:00:03     -9.044270
2014-08-01 00:00:04     -9.152953
...
2014-08-05 23:58:56     453.224110
2014-08-05 23:58:57     453.377437
2014-08-05 23:58:58     453.530764
2014-08-05 23:58:59     453.684090
2014-08-05 23:59:00     453.837417
Freq: S, Length: 431941, dtype: float64
```

pandas also provides a very convenient resampling method referred to as open, high, low, and close, by using the `.ohlc()` method. The following example takes our second-by-second data and calculates hour-by-hour `ohlc` values:

```
In [91]: # show ohlc resampling
ohlc = walk.resample("H").ohlc()
ohlc
```

```
Out[91]:
```

	open	high	low	\
2014-08-01 00:00:00	0.469112	0.469112	-67.873166	
2014-08-01 01:00:00	-3.374321	23.793007	-56.585154	
2014-08-01 02:00:00	-54.276885	5.232441	-87.809456	
2014-08-01 03:00:00	0.260576	17.124638	-65.820652	
2014-08-01 04:00:00	-38.436581	3.537231	-109.805294	
...	
2014-08-05 19:00:00	437.652077	550.408942	430.549178	
2014-08-05 20:00:00	496.539759	510.371745	456.365565	
2014-08-05 21:00:00	476.025498	506.952877	425.472410	
2014-08-05 22:00:00	497.941355	506.599652	411.119919	
2014-08-05 23:00:00	443.017962	489.083657	426.062444	
	close			
2014-08-01 00:00:00	-2.922520			
2014-08-01 01:00:00	-55.101543			
2014-08-01 02:00:00	1.913276			
2014-08-01 03:00:00	-38.530620			
2014-08-01 04:00:00	-61.014553			
...	...			
2014-08-05 19:00:00	494.471788			
2014-08-05 20:00:00	476.505765			
2014-08-05 21:00:00	498.547578			
2014-08-05 22:00:00	443.925832			
2014-08-05 23:00:00	456.191430			

[120 rows x 4 columns]

Time-series moving-window operations

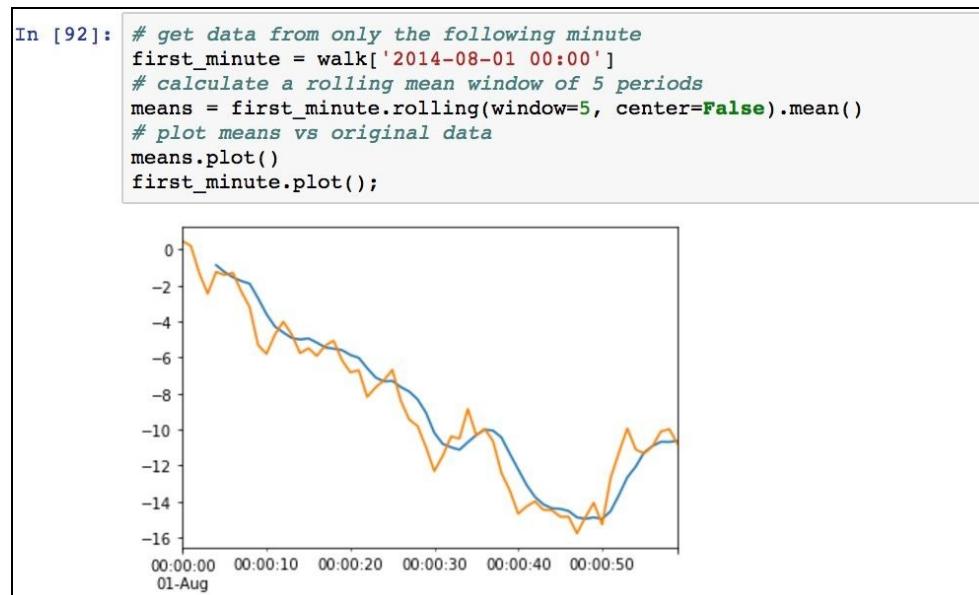
pandas provides a number of functions to compute moving (also known as rolling) statistics. In a rolling window, pandas computes the statistic on a window of data represented by a particular period of time. The window is then rolled along a certain interval, and the statistic is continually calculated on each window as long as the window fits within the dates of the time series.

pandas provides direct support for rolling windows by providing a `.rolling()` method on the Series and DataFrame objects. The resulting value from `.rolling()` can then have one of many different methods called which perform the calculation on each window. The following table shows a number of these methods:

Function	Description
<code>.rolling().mean()</code>	The mean of values in the window
<code>.rolling().std()</code>	The standard deviation of values in the window
<code>.rolling().var()</code>	The variance of values
<code>.rolling().min()</code>	The minimum of values in the window
<code>.rolling().max()</code>	The maximum of values in the window
<code>.rolling().cov()</code>	The covariance of values
<code>.rolling().quantile()</code>	Moving window score at percentile/sample quantile
<code>.rolling().corr()</code>	The correlation of values in the window
<code>.rolling().median()</code>	The median of values in the window
<code>.rolling().sum()</code>	The sum of values in the window
<code>.rolling().apply()</code>	The application of a user function to values in the window

.rolling().count()	The number of non-NaN values in a window
.rolling().skew()	The skewedness of the values in the window
.rolling().kurt()	The kurtosis of values in the window

As a practical example, a rolling mean is commonly used to smooth out short-term fluctuations and highlight longer-term trends in data and is used quite commonly in financial time-series analysis. To demonstrate, we will calculate a rolling mean with a window of 5 on the first minute of the random walk created earlier in the chapter.

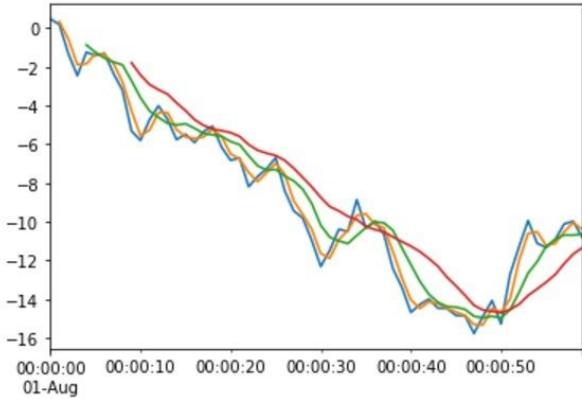


The generation of charts will be covered in more detail in Chapter 14, Visualization

It can be seen how `.rolling().mean()` provides a smoother representation of the underlying data. A larger window will create less variance, and smaller windows will create more (until the window size is 1, which will be identical to the original series).

The following demonstrates the rolling mean with windows of 2, 5, and 10 plotted against the original

```
In [93]: # demonstrate the difference between 2, 5 and
# 10 interval rolling windows
h1w = walk['2014-08-01 00:00']
means2 = h1w.rolling(window=2, center=False).mean()
means5 = h1w.rolling(window=5, center=False).mean()
means10 = h1w.rolling(window=10, center=False).mean()
h1w.plot()
means2.plot()
means5.plot()
means10.plot();
```



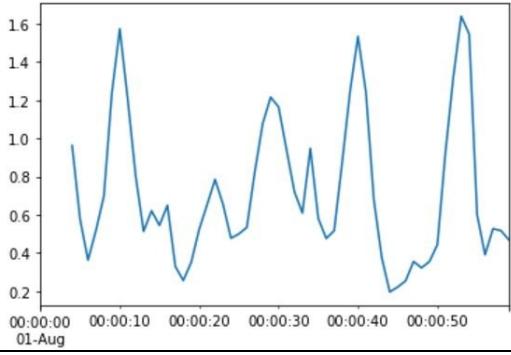
series:

Note that the larger the window, the more data is missing at the beginning of the curve. A window of size n requires n data points before the measure can be calculated and hence the gap in the beginning of the plot.

Any user defined function can be applied via a rolling window using the `.rolling().apply()` method. The supplied function will be passed an array of values in the window and should return a single value. Pandas will then combine the results from each window into a time-series.

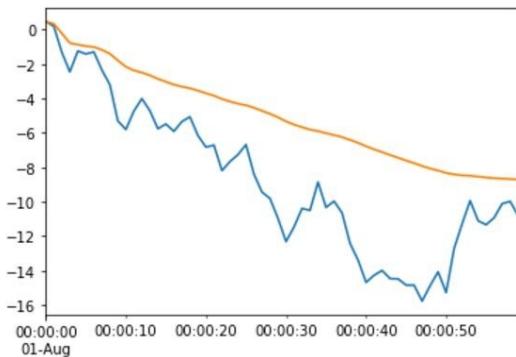
To demonstrate, the following code calculates the mean average deviation which gives you a feel of how far all values in the sample are from the overall mean:

```
In [94]: # calculate mean average deviation with window of 5 intervals
mean_abs_dev = lambda x: np.fabs(x - x.mean()).mean()
means = h1w.rolling(window=5, center=False).apply(mean_abs_dev)
means.plot();
```



An expanding window mean can be calculated using a slight variant of the use of the `pd.rolling_mean` function that repeatedly calculates the mean by always starting with the first value in the time series and for each iteration increases the window size by one. An expanding window mean will be more stable (less responsive) than a rolling window, because as the size of the window increases, the less the impact

```
In [95]: # calculate an expanding rolling mean
hlw.plot()
expanding = hlw.expanding(min_periods=1).mean()
expanding.plot();
```



of the next value will be:

Summary

In this chapter, we examined many of the ways to represent events that occur at specific points in time and how to model these values as they change over time. This involved learning many capabilities of pandas including date and time objects, representation of changes in time with intervals and periods, and performing several types of operations on time series data such as frequency conversion, resampling, and calculating rolling windows.

In the remaining two chapters of this book, we will leave the mechanics of pandas behind and look more into both visualization of data and applying pandas to analysis of financial data.

Visualization

One of the most important parts of data analysis is in creating a great visualization to immediately convey the underlying meaning in the data. Data visualization is effective, as we humans are visual creatures and have evolved to be able to discern meaning when information is laid out in a way that our brain can interpret almost immediately when the impulses from the retina hit the brain.

Over the years, there has been significant research that has resulted in many effective visualization techniques to convey specific patterns in data. These patterns have been implemented in visualization libraries, and pandas is designed to utilize these and make their use very simple.

This chapter will cover several of these techniques, primarily focusing on matplotlib, and many of the common visualizations. We will do this in three steps. The first introduces the general concepts of programming visualizations with pandas, emphasizing the process of creating time-series charts. During this, we will dive into the techniques of labeling axes and creating legends, colors, line styles, and markers.

The second step will focus on the many types of data visualizations commonly used in pandas and data analysis, including:

- Showing relative differences with bar plots
- Picturing distributions of data with histograms
- Depicting distributions of categorical data with box and whisker charts
- Demonstrating cumulative totals with area plots
- Relationships between two variables with scatter plots
- Estimates of distribution with the kernel density plot
- Correlations between multiple variables with the scatter plot matrix
- Strengths of relationships in multiple variables with heatmaps

The final step will examine creating composite plots by dividing plots into subparts to be able to render multiple plots within a single graphical canvas. This will help the viewer of the visualization to relate different sets of data at a glance.

Configuring pandas

The examples in this chapter will all be based on the following imports and default settings. One small difference in this chapter is the declaration of a `seedval` variable that is shared throughout the examples:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set formattign options
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline

# a common seed value for random number generation
seedval = 111111
```


Plotting basics with pandas

The pandas library itself performs does not perform data visualization. To perform this, pandas tightly integrates with other robust visualization libraries that are part of the Python ecosystem. The most common of these integrations is with `matplotlib`. This chapter will, therefore, focus its examples on `matplotlib`, but we will also point you to other possible libraries to try on your own. Two of these are worth mentioning.

Seaborn is another Python visualization library which is also based on `matplotlib`. It provides a high-level interface for rendering attractive statistical graphics. It has native support for NumPy and pandas data structures. The goal of Seaborn is to create `matplotlib` graphs that look a lot less scientific in nature. To learn about Seaborn, please visit the site at <http://seaborn.pydata.org/index.html>.

While both Seaborn and `matplotlib` are exceptional at rendering data, they both suffer from a lack of user interactivity. And although the renderings from both can be displayed into a browser through tools such as Jupyter, the renderings themselves are not created with DOM nor do they make use of any capabilities of the browser.

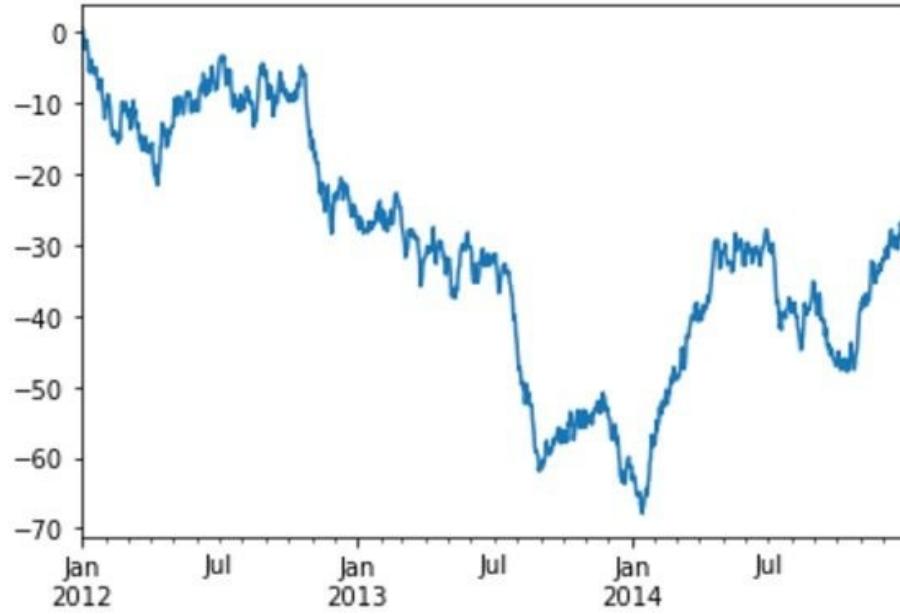
In order to facilitate rendering in the browser and to provide rich interactivity, several libraries have been created to integrate Python and pandas with `d3.js`. `d3.js` is a JavaScript library for manipulating documents and creating rich, interactive data visualizations. One of the most popular of these is `mpld3`. Unfortunately, at the time of writing this book, it does not work with Python 3.6 and could not be covered. But please still check out `d3.js` at <https://d3js.org> and `mpld3` at <http://mpld3.github.io/>.

Creating time-series charts

One of the most common data visualizations is of time-series data. Visualizing a time series in pandas is as simple as calling `.plot()` on a `DataFrame` or `Series` object that models a time series.

The following example demonstrates creating a time series that represents a random walk of values over time, akin to the movements in the price of a stock:

```
In [2]: # generate a random walk time-series
np.random.seed(seedval)
s = pd.Series(np.random.randn(1096),
              index=pd.date_range('2012-01-01',
                                   '2014-12-31'))
walk_ts = s.cumsum()
# this plots the walk - just that easy :)
walk_ts.plot();
```

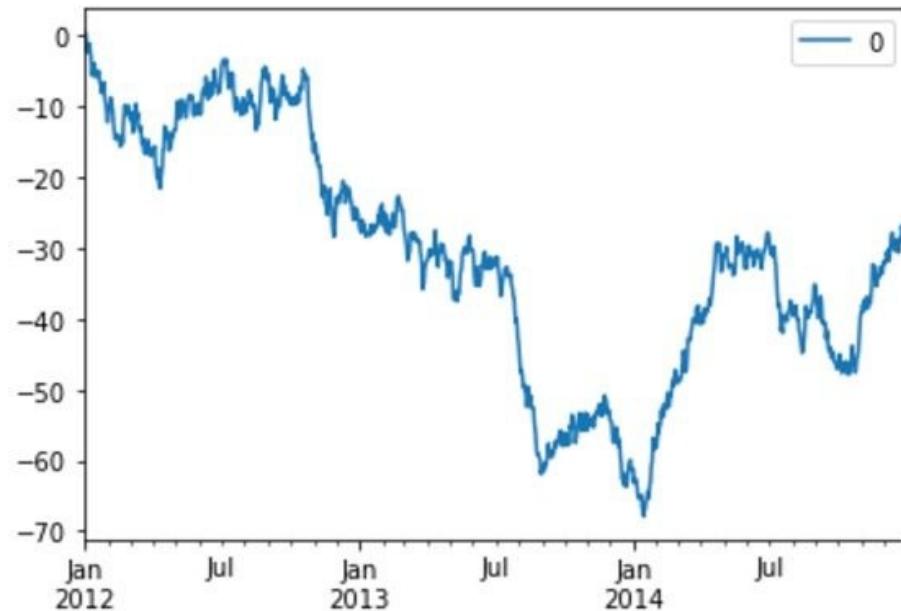


The `.plot()` method on pandas objects is a wrapper function around the `matplotlib` library's `plot()` function. It makes plots of pandas data very easy to create as its implementation is coded to know how to render many visualizations based on the underlying data. It handles many of the details such as selecting series, labeling, and axis generation.

In the previous example, `.plot()` determined that the `Series` contains dates for its index and therefore the `x`-axis should be formatted as dates. It also selects a default color for the data.

Plotting a `Series` of data gives a similar result as rendering a `DataFrame` with a single column. The following code shows this by producing the same graph with one small difference: it has added a legend to the graph. Charts will contain a legend by default when generated from a `DataFrame`.

```
In [3]: # a DataFrame with a single column will produce  
# the same plot as plotting the Series it is created from  
walk_df = pd.DataFrame(walk_ts)  
walk_df.plot();
```



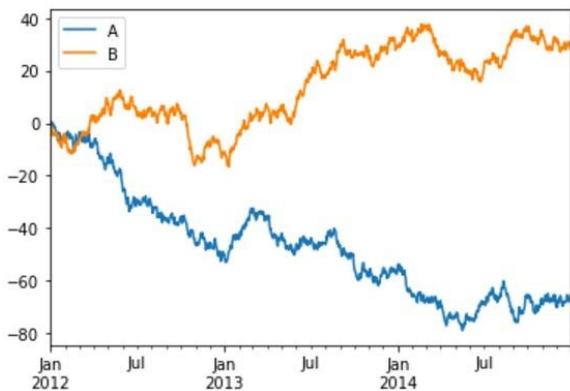
If a DataFrame contains more than one column, .plot() will add multiple items to the legend and pick a different color for each line:

```
In [4]: # generate two random walks, one in each of  
# two columns in a DataFrame  
np.random.seed(seedval)  
df = pd.DataFrame(np.random.randn(1096, 2),  
                  index=walk_ts.index, columns=list('AB'))  
walk_df = df.cumsum()  
walk_df.head()
```

```
Out[4]:
```

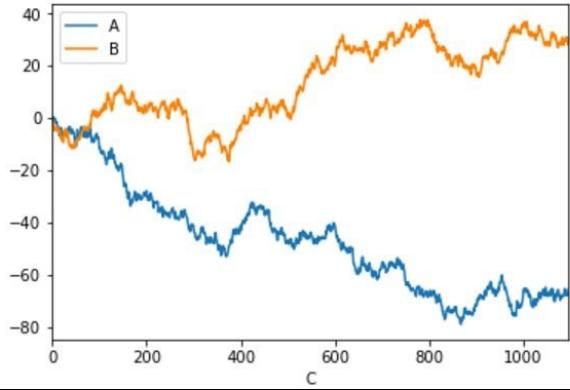
	A	B
2012-01-01	0.469112	-0.282863
2012-01-02	-1.039946	-1.418496
2012-01-03	0.172166	-1.591710
2012-01-04	0.291375	-2.635946
2012-01-05	-0.570474	-4.740516

```
In [5]: # plot the DataFrame, which will plot a line
# for each column, with a legend
walk_df.plot();
```



If you want to use a column of data in a `DataFrame` as the labels on the x-axis of the plot (instead of the index labels), use the `x` parameter to specify the name of a column representing the labels. Then use the `y` parameter to specify which columns are used as data:

```
In [6]: # copy the walk
df2 = walk_df.copy()
# add a column C which is 0 .. 1096
df2['C'] = pd.Series(np.arange(0, len(df2)), index=df2.index)
# instead of dates on the x-axis, use the 'C' column,
# which will label the axis with 0..1000
df2.plot(x='C', y=['A', 'B']);
```



Adorning and styling your time-series plot

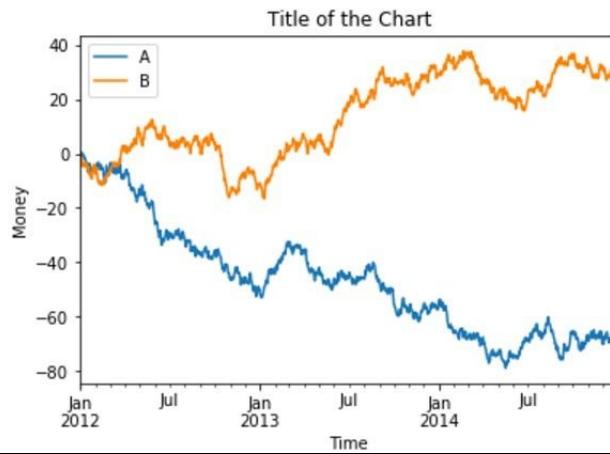
The built-in `.plot()` method has many options that you can use to change the content in the plot. Let's cover several of the common options used in many plots.

Adding a title and changing axes labels

The title of the chart can be set using the `title` parameter. The axes labels are set not with `.plot()` but directly by using the `plt.ylabel()` and `plt.xlabel()` functions after calling `.plot()`:

```
In [7]: # create a time-series chart with a title and specific
# x and y axis labels

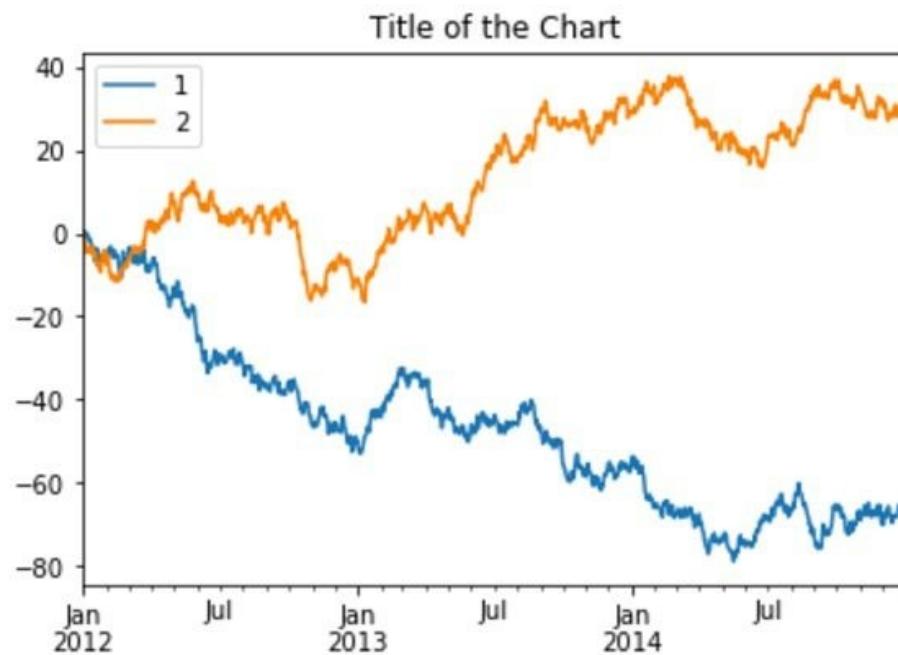
# title is set in the .plot() method as a parameter
walk_df.plot(title='Title of the Chart')
# explicitly set the x and y axes labels after the .plot()
plt.xlabel('Time')
plt.ylabel('Money');
```



Specifying the legend content and position

To change the text used for each data series in the legend (the default is the column name from `DataFrame`), capture the `ax` object returned from the `.plot()` method and use its `.legend()` method. This object is an `AxesSubplot` object and can be used to change various aspects of the plot before it is generated:

```
In [8]: # change the legend items to be different
# from the names of the columns in the DataFrame
ax = walk_df.plot(title='Title of the Chart')
# this sets the legend labels
ax.legend(['1', '2']);
```



The location of the legend can be set using the `loc` parameter of `.legend()`. By default, pandas sets the location to `'best'`, which tells `matplotlib` to examine the data and determine the best place it thinks to put the legend. However, you can also specify any of the following to position the legend more specifically (you can use either the string or the numeric code):

Text	Code
'best'	0
'upper right'	1
'upper left'	2

'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

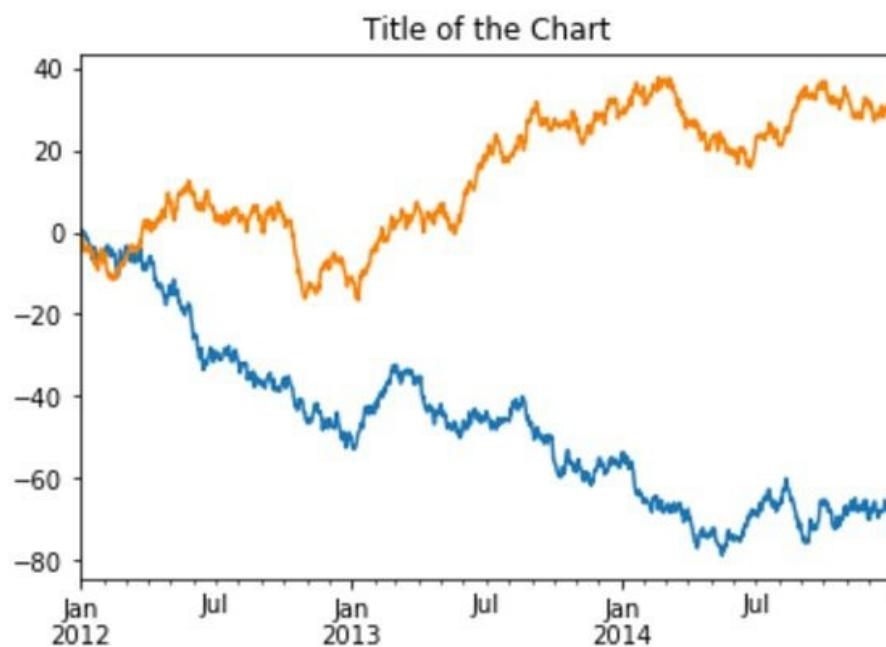
The following example demonstrates placing the legend in the upper-center portion of the graph:

```
In [9]: # change the position of the legend
ax = walk_df.plot(title='Title of the Chart')
# put the legend in the upper center of the chart
ax.legend(['1', '2'], loc='upper center');
```



Legends can be turned off by using the `legend=False`:

```
In [10]: # omit the legend by using legend=False  
walk_df.plot(title='Title of the Chart', legend=False);
```



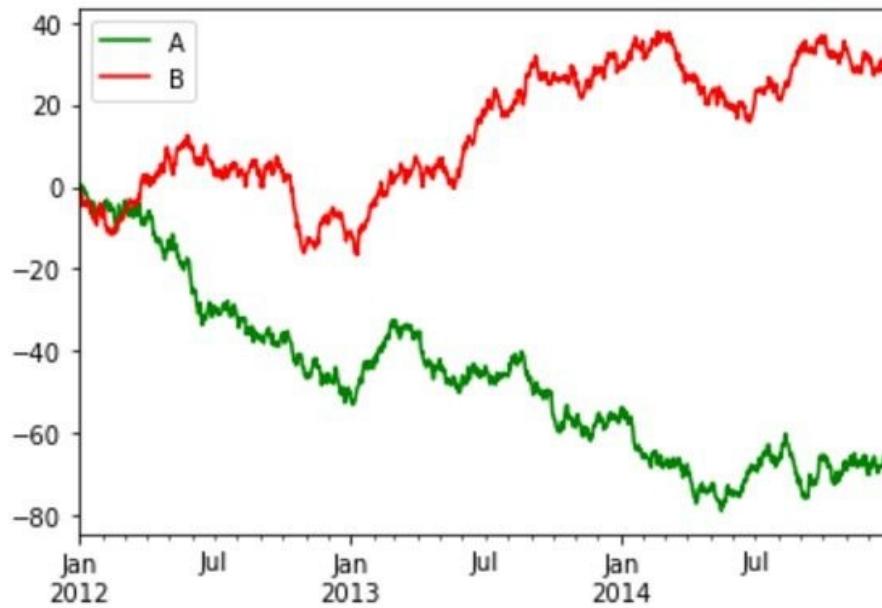
Specifying line colors, styles, thickness, and markers

pandas automatically sets the colors of each series on any chart. To specify your own colors, supply style codes to the `style` parameter of the `plot` function. pandas has a number of built-in single-character codes for colors, several of which are listed here:

- b: Blue
- g: Green
- r: Red
- c: Cyan
- m: Magenta
- y: Yellow
- k: Black
- w: White

It is also possible to specify the color using a hexadecimal RGB code in **#RRGGBB** format. The following example demonstrates both by setting the color of the first series to green using a single-digit code and the second series to red using the RGB hexadecimal code:

```
In [11]: # change the line colors on the plot
# use character code for the first line,
# hex RGB for the second
walk_df.plot(style=[ 'g', '#FF0000']);
```



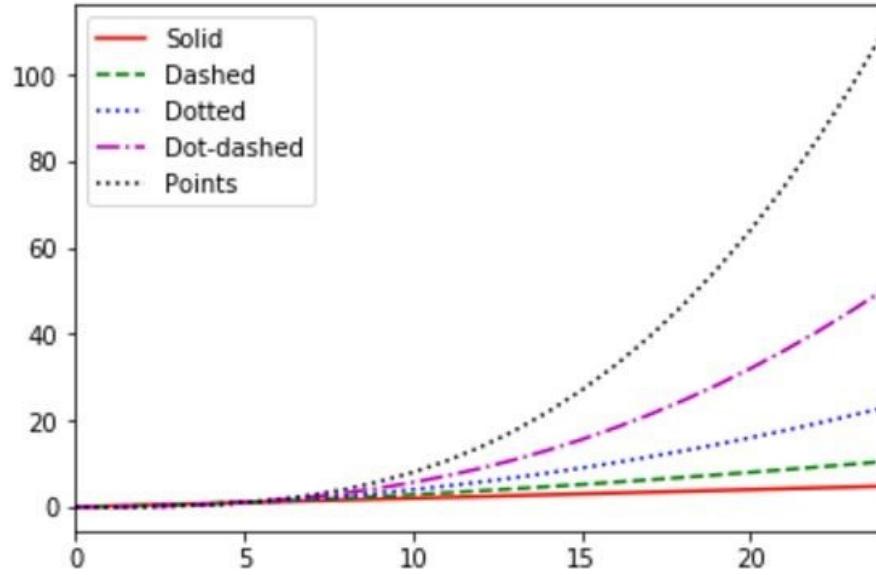
Line styles can be specified by using a line style code. These can be used in combination with the color style codes by immediately following the color code. Here are some examples of several useful line style codes:

- '-' = solid

- '--' = dashed
- ':' = dotted
- '-.' = dot-dashed
- '.' = points

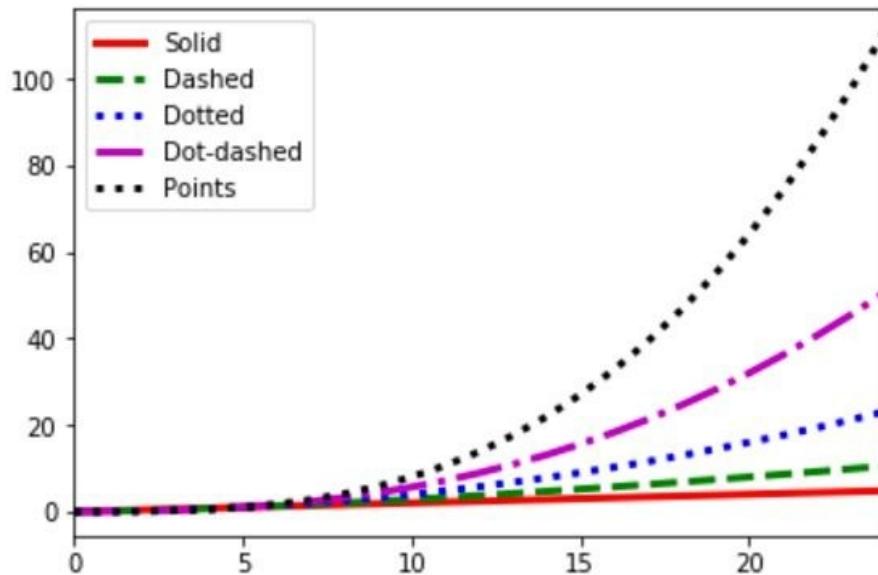
The following plot demonstrates these five line styles by drawing five data series, each with one of these styles:

```
In [12]: # show off different line styles
t = np.arange(0., 5., 0.2)
legend_labels = ['Solid', 'Dashed', 'Dotted',
                  'Dot-dashed', 'Points']
line_style = pd.DataFrame({0 : t,
                           1 : t**1.5,
                           2 : t**2.0,
                           3 : t**2.5,
                           4 : t**3.0})
# generate the plot, specifying color and line style for each line
ax = line_style.plot(style=['r-', 'g--', 'b:', 'm-.', 'k:'])
# set the legend
ax.legend(legend_labels, loc='upper left');
```



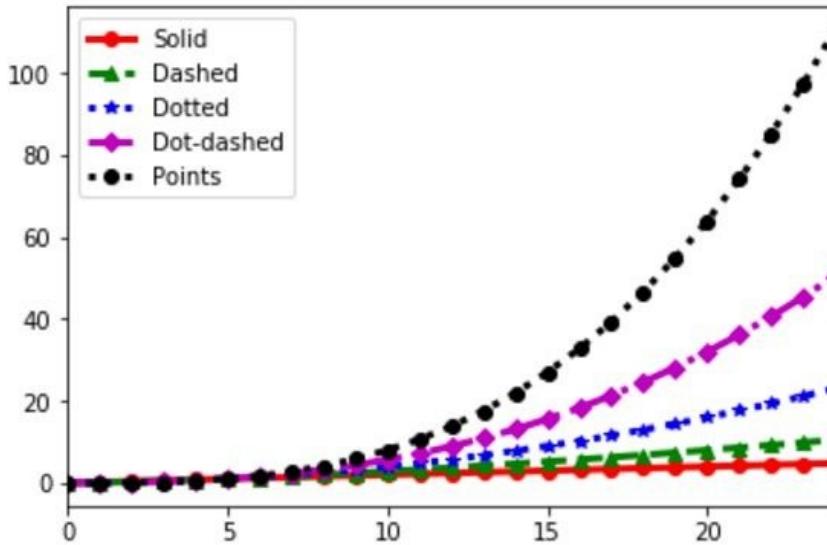
The thickness of a line can be specified by using the `lw` parameter . This can be passed a thickness for multiple lines by passing a list of widths, or a single width that is applied to all the lines. The following code redraws the graph with a line width of 3, making the lines a little more pronounced:

```
In [13]: # regenerate the plot, specifying color and line style  
# for each line and a line width of 3 for all lines  
ax = line_style.plot(style=['r-', 'g--', 'b:', 'm-.', 'k:'], lw=3)  
ax.legend(legend_labels, loc='upper left');
```



Markers on a line can be specified by using abbreviations in the style code. There are quite a few marker types provided and you can see them all at http://matplotlib.org/api/markers_api.html. We will examine five of them in the next chart by having each series use a different marker from the following: circles, stars, triangles, diamonds, and points:

```
In [14]: # redraw, adding markers to the lines  
ax = line_style.plot(style=['r-o', 'g--^', 'b-*',  
                           'm-.D', 'k:o'], lw=3)  
ax.legend(legend_labels, loc='upper left');
```



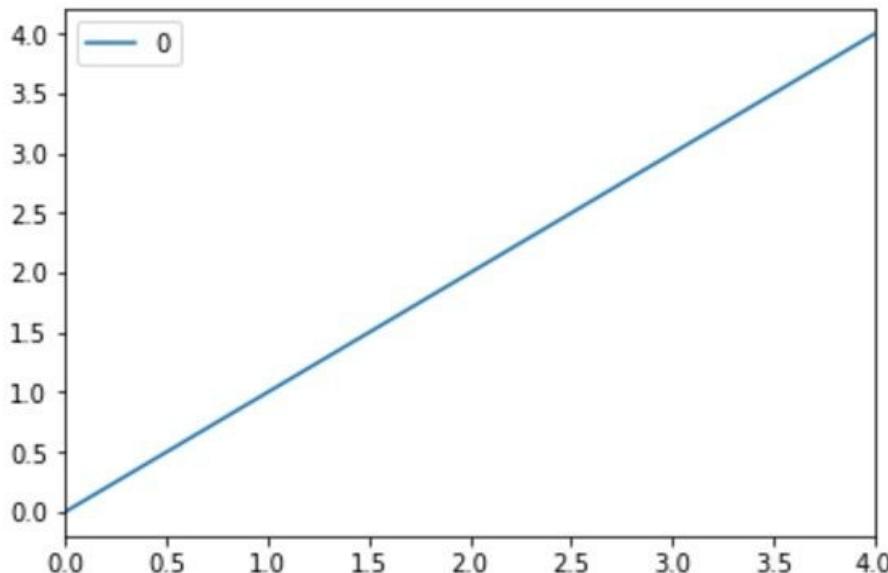
Specifying tick mark locations and tick labels

The location and rendering of tick marks can be customized using various functions. The following code walks you through several examples of controlling their values and rendering.

The values that pandas decides to use at label locations can be found using the `plt.xticks()` function. This function returns two values: the values at each tick and the objects representing the actual labels:

```
In [15]: # a simple plot to use to examine ticks
ticks_data = pd.DataFrame(np.arange(0,5))
ticks_data.plot()
ticks, labels = plt.xticks()
ticks
```

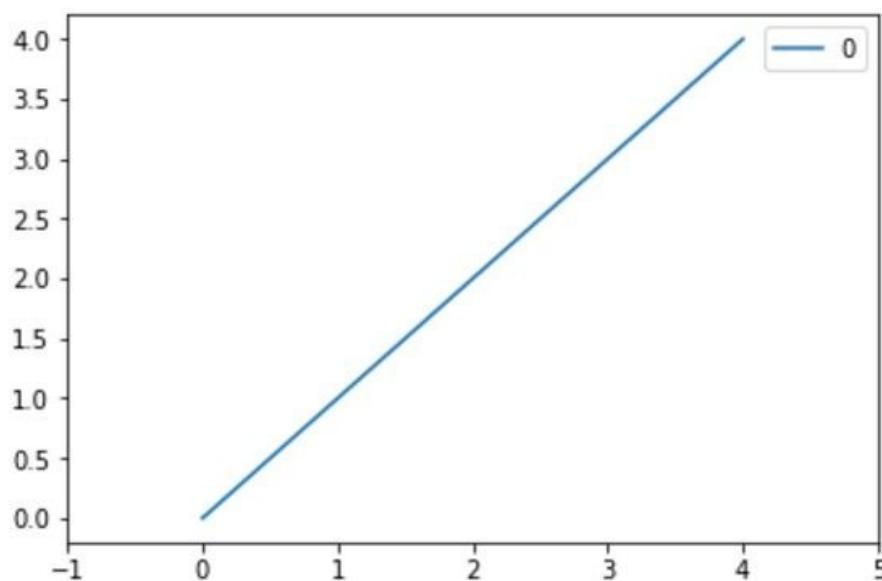
```
Out[15]: array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ])
```



This tick array contains the locations of the ticks in units of the actual data along the x-axis. In this case, pandas has decided that a range of 0 through 4 (the min and max) and an interval of 0.5 are appropriate.

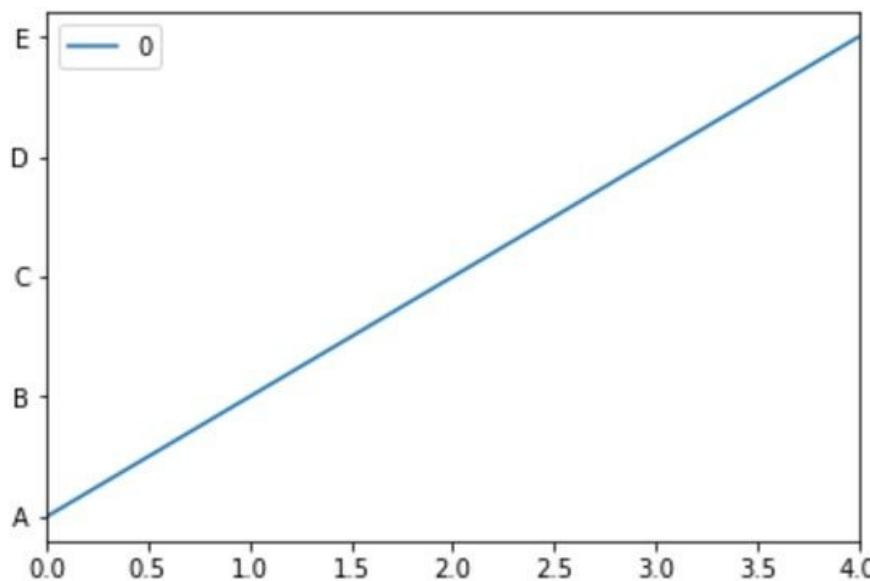
If you want to use other locations, then provide them by passing a list to `plt.xticks()`. The following code demonstrates this using even integers from -1 to 5. This set of values will change both the extents of the axis as well as remove non-integral labels:

```
In [16]: # resize x-axis to (-1, 5), and draw ticks  
# only at integer values  
ticks_data = pd.DataFrame(np.arange(0,5))  
ticks_data.plot()  
plt.xticks(np.arange(-1, 6));
```



The labels at each tick location can be specified by passing them as the second parameter. This is shown in the following sample by changing the y-axis ticks and labels to integral values and consecutive alpha characters:

```
In [17]: # rename y-axis tick labels to A, B, C, D, and E  
ticks_data = pd.DataFrame(np.arange(0,5))  
ticks_data.plot()  
plt.yticks(np.arange(0, 5), list("ABCDE"));
```



Formatting axes' tick date labels using formatters

Formatting of axes' labels whose underlying data type is `datetime` is done using **locators** and **formatters**. Locators control the position of the ticks and formatters control the formatting of the labels.

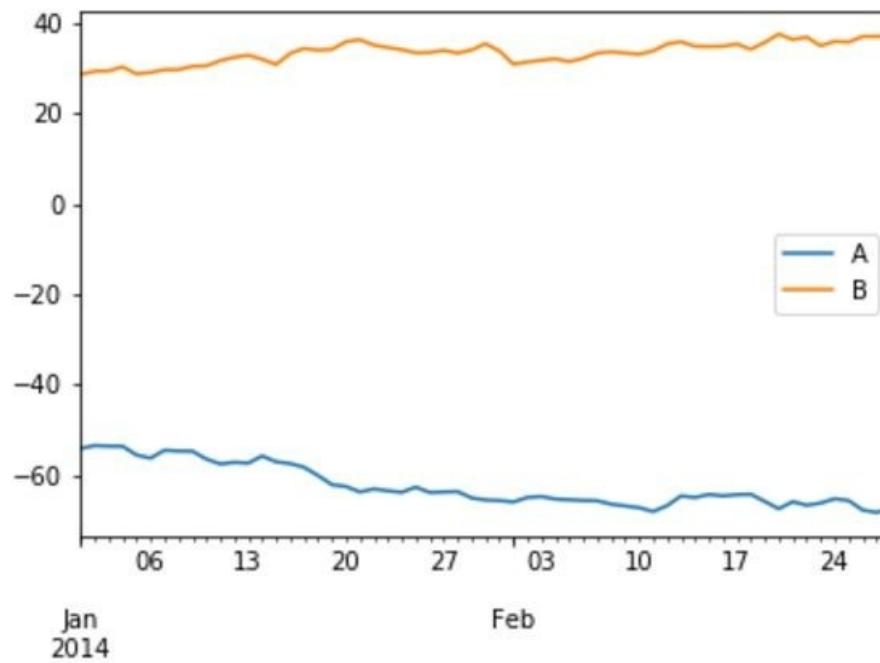
`matplotlib` provides several classes in `matplotlib.dates` to help facilitate the process:

- `MinuteLocator`, `HourLocator`, `DayLocator`, `WeekdayLocator`, `MonthLocator`, and `YearLocator`
- These are specific locators coded to determine where the ticks for each type of date field will be found on the axis
- `DateFormatter`
- This is a class that can be used to format date objects into labels on the axis

The default locator and formatter are `AutoDateLocator` and `AutoDateFormatter`. These can be changed by providing different object implementations.

To demonstrate, let's start with the following subset of random walk data from earlier examples; it represents only the data from January through February 2014. Plotting this gives us the following output:

```
In [18]: # plot January–February 2014 from the random walk
walk_df.loc['2014-01':'2014-02'].plot();
```



The labels on the *x*-axis of this plot have two series, the **minor** and the **major**. The minor labels in this plot contain the day of the month, and the major ones contain the year and month (the year only for the first month). We can set locators and formatters for each of the minor and major levels to change the values.

This will be demonstrated by changing the minor labels to start at Monday in each week and to contain the date and day of the week (right now, the chart uses weekly and only Friday's date-without the day name).

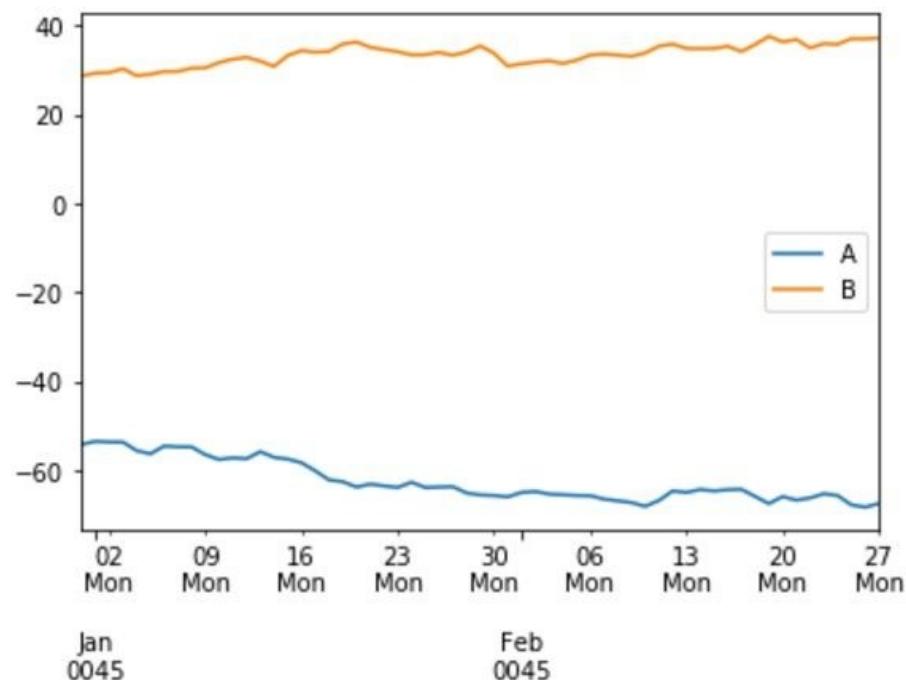
The major labels will use the monthly location and will always include both the month name and the year:

```
In [19]: # this import styles helps us type less
from matplotlib.dates import WeekdayLocator, \
DateFormatter, MonthLocator

# plot Jan-Feb 2014
ax = walk_df.loc['2014-01':'2014-02'].plot()

# do the minor labels
weekday_locator = WeekdayLocator(byweekday=(0), interval=1)
ax.xaxis.set_minor_locator(weekday_locator)
ax.xaxis.set_minor_formatter(DateFormatter("%d\n%a"))

# do the major labels
ax.xaxis.set_major_locator(MonthLocator())
ax.xaxis.set_major_formatter(DateFormatter('\n\n\n%b\n%Y'))
```



This is almost what we wanted. However, note that the year is being reported as 0045. To create a plot with custom-date-based labels, it is required to avoid the pandas `.plot()` and go all the way down to directly using `matplotlib`. Fortunately, this is not too hard. This snippet changes the code slightly and renders what we need in the desired format:

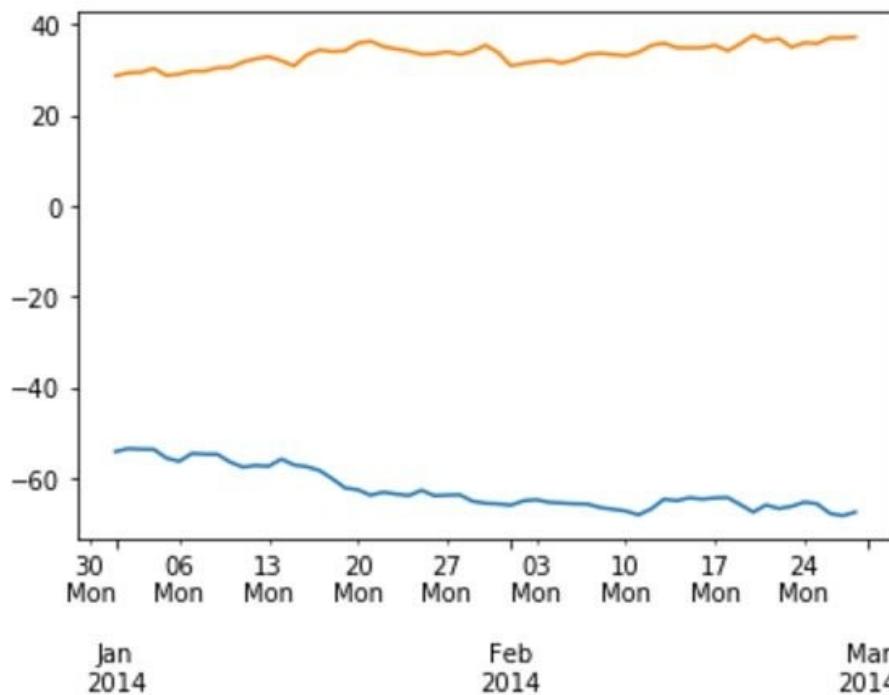
```
In [20]: # this gets around the pandas / matplotlib year issue
# need to reference the subset twice, so let's make a variable
walk_subset = walk_df['2014-01':'2014-02']

# this gets the plot so we can use it, we can ignore fig
fig, ax = plt.subplots()

# inform matplotlib that we will use the following as dates
# note we need to convert the index to a pydatetime series
ax.plot_date(walk_subset.index.to_pydatetime(), walk_subset, '-')

# do the minor labels
weekday_locator = WeekdayLocator(byweekday=(0), interval=1)
ax.xaxis.set_minor_locator(weekday_locator)
ax.xaxis.set_minor_formatter(DateFormatter('%d\n%a'))

# do the major labels
ax.xaxis.set_major_locator(MonthLocator())
ax.xaxis.set_major_formatter(DateFormatter('\n\n\n%b\n\n%Y'));
```



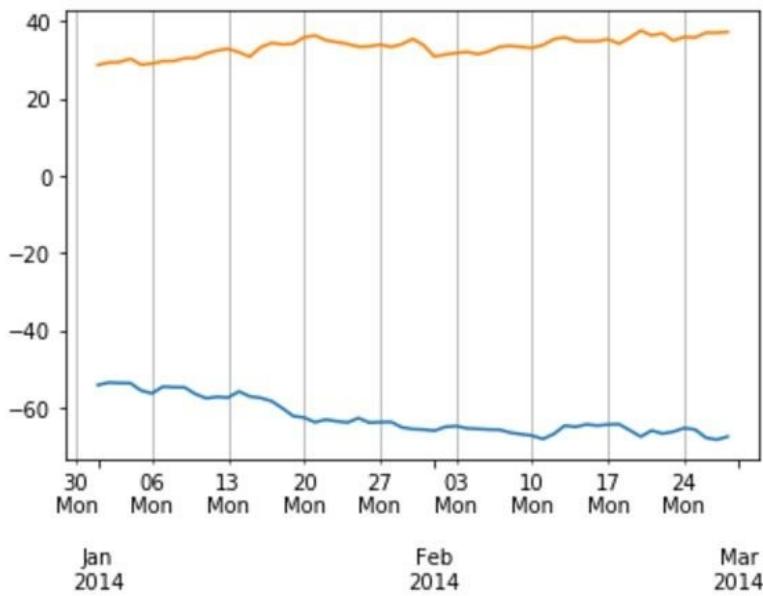
We can use the `.grid()` method of the `x-axis` object to add grid lines for the minor and major axes' ticks. The first parameter is `True` in order to render the lines or `False` in order to hide them. The second specifies the set of ticks. The following code replots this graph without the major grid lines but while rendering the minor lines:

```
In [21]: # this gets the plot so we can use it, we can ignore fig
fig, ax = plt.subplots()

# inform matplotlib that we will use the following as dates
# note we need to convert the index to a pydatetime series
ax.plot_date(walk_subset.index.to_pydatetime(), walk_subset, '-')

# do the minor labels
weekday_locator = WeekdayLocator(byweekday=(0), interval=1)
ax.xaxis.set_minor_locator(weekday_locator)
ax.xaxis.set_minor_formatter(DateFormatter('%d\n%a'))
ax.xaxis.grid(True, "minor") # turn on minor tick grid lines
ax.xaxis.grid(False, "major") # turn off major tick grid lines

# do the major labels
ax.xaxis.set_major_locator(MonthLocator())
ax.xaxis.set_major_formatter(DateFormatter('\n\n\n%b\n%Y'));
```



The final demonstration of formatting will use only the major labels, but on a weekly basis and using a `YYYY-MM-DD` format. However, since these would overlap, we will specify that they should be rotated to prevent the overlap. This rotation is specified using the `fig.autofmt_xdate()` function:

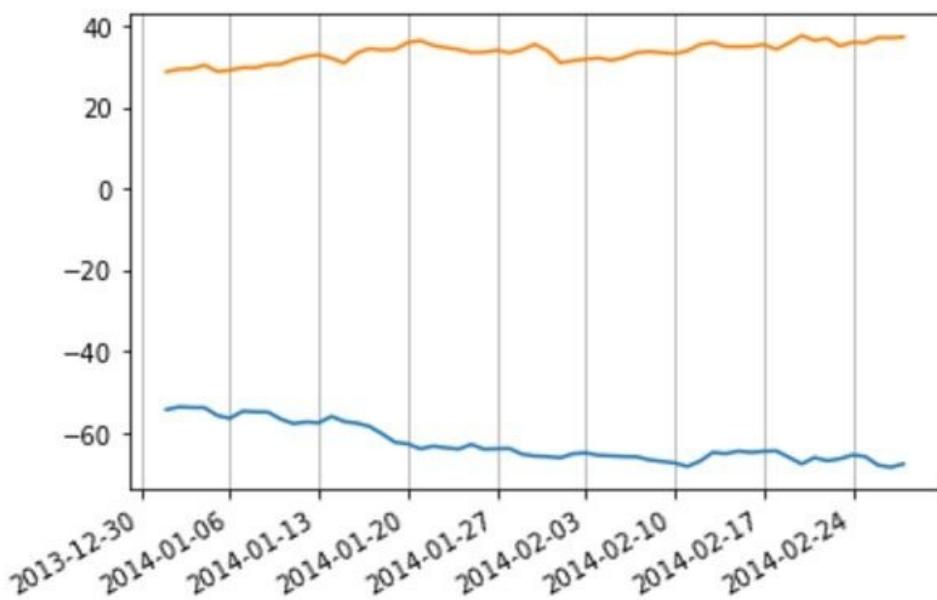
```
In [22]: # this gets the plot so we can use it, we can ignore fig
fig, ax = plt.subplots()

# inform matplotlib that we will use the following as dates
# note we need to convert the index to a pydatetime series
ax.plot_date(walk_subset.index.to_pydatetime(), walk_subset, '-')

ax.xaxis.grid(True, "major") # turn off major tick grid lines

# do the major labels
ax.xaxis.set_major_locator(weekdayLocator)
ax.xaxis.set_major_formatter(DateFormatter('%Y-%m-%d'));

# informs to rotate date labels
fig.autofmt_xdate();
```



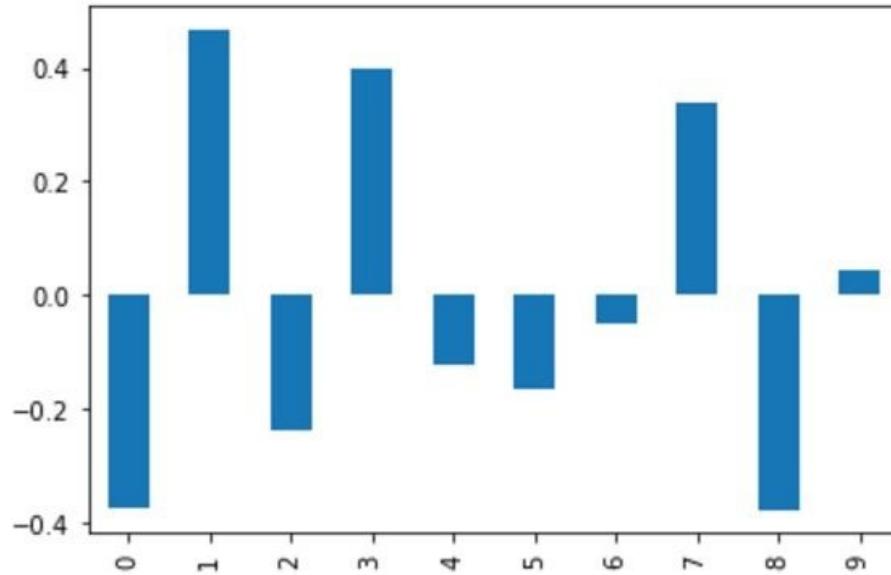
Common plots used in statistical analyses

Having learned how to create, lay out, and annotate time-series charts, we will now look at creating a variety that is useful in presenting statistical information.

Showing relative differences with bar plots

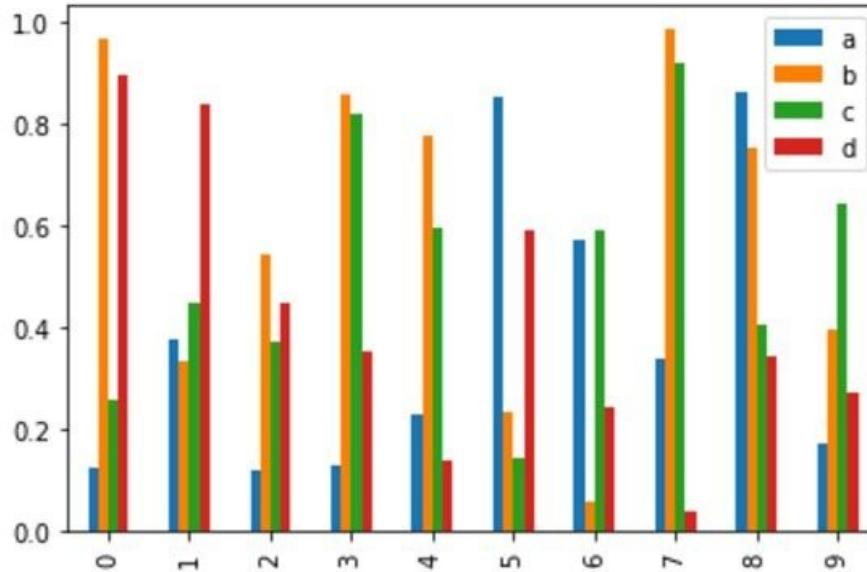
Bar plots are useful in visualizing the relative differences in values of non-time-series data. Bar plots can be created by using the `kind='bar'` parameter of the `.plot()`:

```
In [23]: # make a bar plot
# create a small series of 10 random values centered at 0.0
np.random.seed(seedval)
s = pd.Series(np.random.rand(10) - 0.5)
# plot the bar chart
s.plot(kind='bar');
```



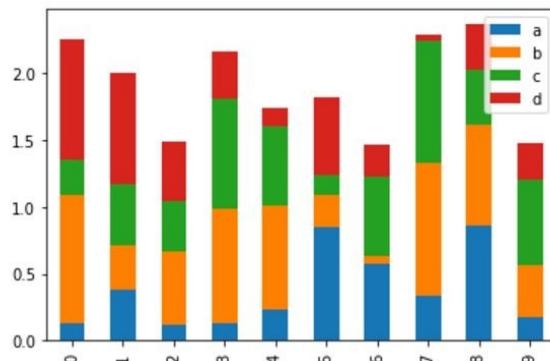
A multiple-series bar plot will be created to compare multiple values at each x-axis label:

```
In [24]: # draw a multiple series bar chart
# generate 4 columns of 10 random values
np.random.seed(seedval)
df2 = pd.DataFrame(np.random.rand(10, 4),
                   columns=[ 'a', 'b', 'c', 'd'])
# draw the multi-series bar chart
df2.plot(kind='bar');
```



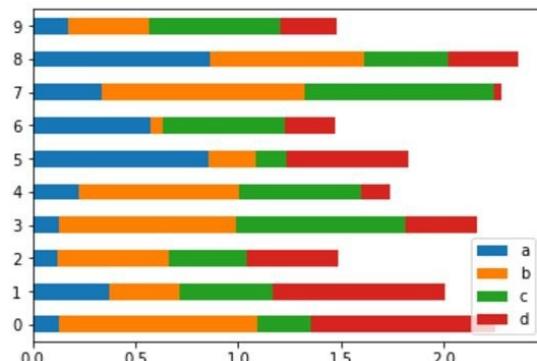
The `stacked=True` parameter can be used to stack the bars instead of being side by side:

```
In [25]: # horizontal stacked bar chart
df2.plot(kind='bar', stacked=True);
```



The orientation can be turned to horizontal using `kind='barh'`:

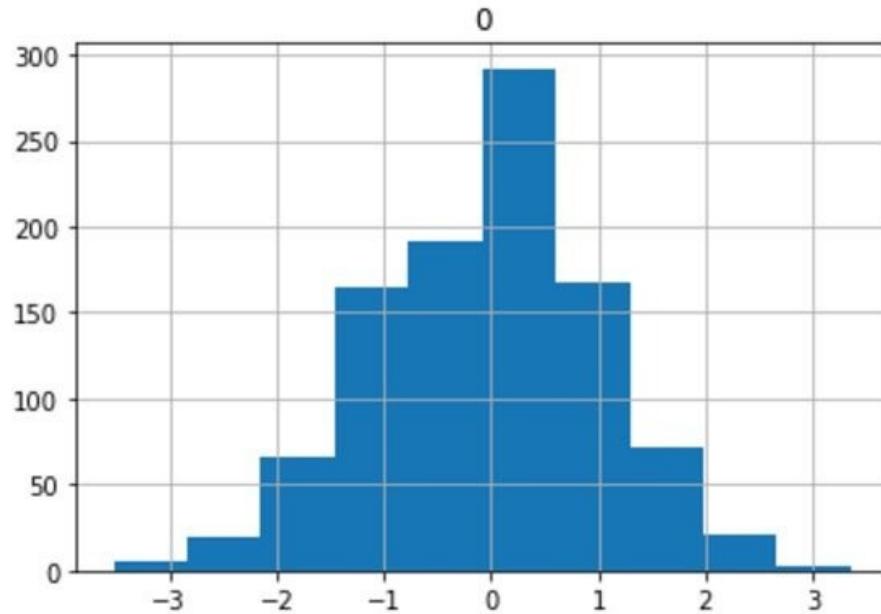
```
In [26]: # horizontal stacked bar chart
df2.plot(kind='barh', stacked=True);
```



Picturing distributions of data with histograms

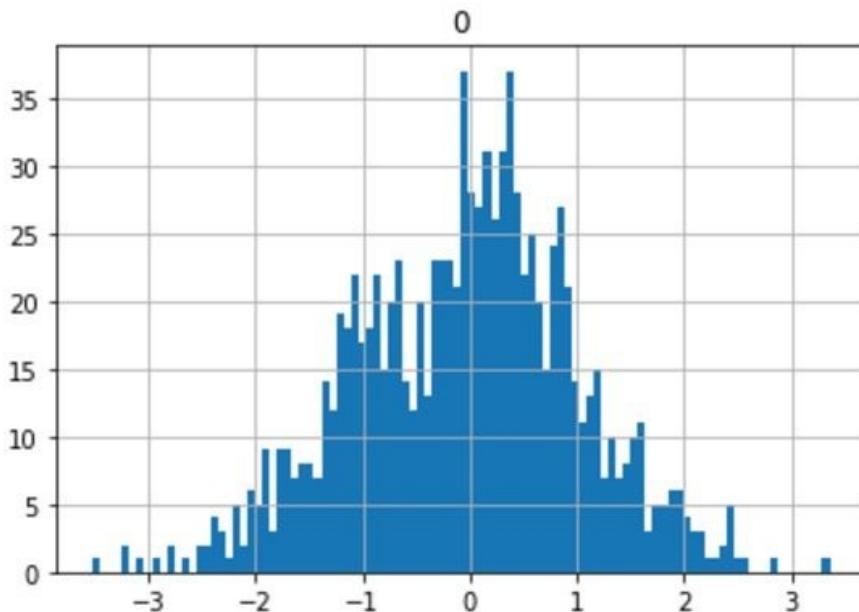
Histograms are useful for visualizing distributions of data. The following code generates a histogram based on 1000 random values in a normal distribution:

```
In [27]: # create a histogram
np.random.seed(seedval)
# 1000 random numbers
dfh = pd.DataFrame(np.random.randn(1000))
# draw the histogram
dfh.hist();
```



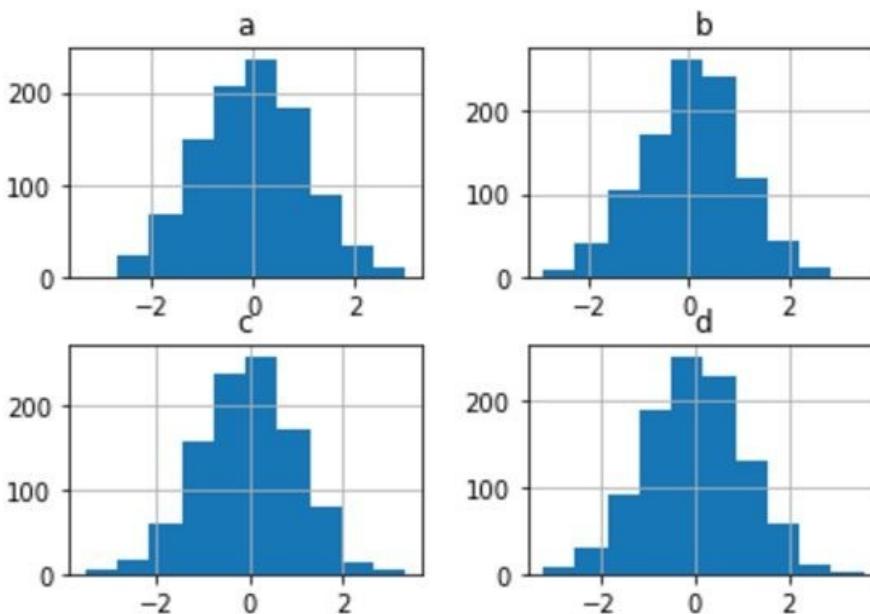
The resolution of a histogram can be controlled by specifying the number of bins to allocate to the graph. The default is 10, and increasing the number of bins gives finer details to the histogram. This code increases the number of bins to 100:

```
In [28]: # histogram again, but with more bins  
dfh.hist(bins = 100);
```



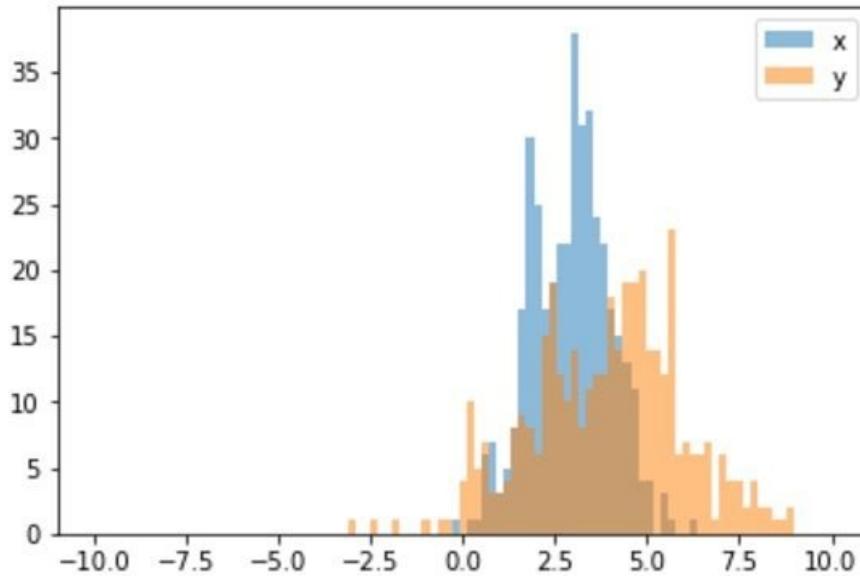
If the data has multiple series, the histogram function will automatically generate multiple histograms, one for each series:

```
In [29]: # generate multiple histogram plot  
# create data frame with 4 columns of 1000 random values  
np.random.seed(seedval)  
dfh = pd.DataFrame(np.random.randn(1000, 4),  
                   columns=[ 'a', 'b', 'c', 'd' ])  
# draw the chart. There are four columns so pandas draws  
# four histograms  
dfh.hist();
```



To overlay multiple histograms on the same graph (to give a quick visual difference of the distribution), call the `pyplot.hist()` function multiple times before `.show()`:

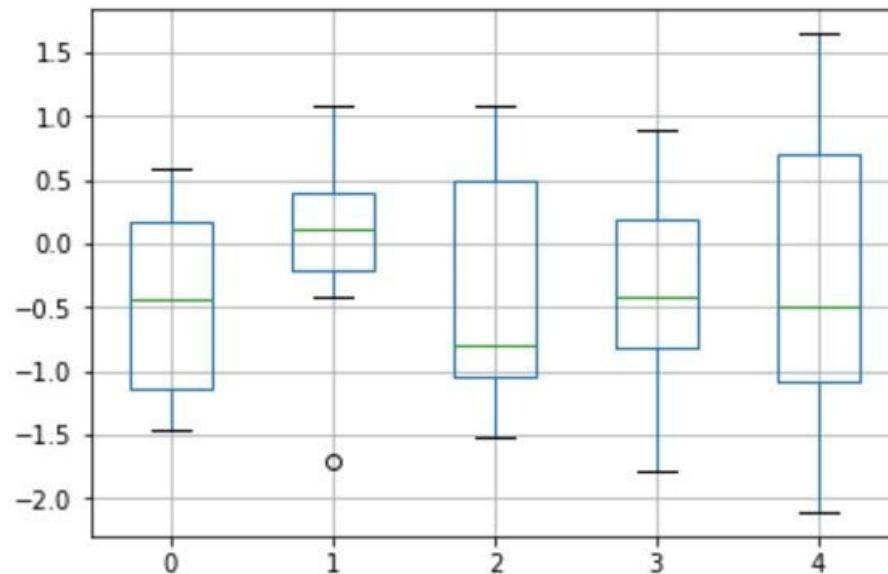
```
In [30]: # directly use pyplot to overlay multiple histograms  
# generate two distributions, each with a different  
# mean and standard deviation  
np.random.seed(seedval)  
x = [np.random.normal(3,1) for _ in range(400)]  
y = [np.random.normal(4,2) for _ in range(400)]  
  
# specify the bins (-10 to 10 with 100 bins)  
bins = np.linspace(-10, 10, 100)  
  
# generate plot x using plt.hist, 50% transparent  
plt.hist(x, bins, alpha=0.5, label='x')  
# generate plot y using plt.hist, 50% transparent  
plt.hist(y, bins, alpha=0.5, label='y')  
plt.legend(loc='upper right');
```



Depicting distributions of categorical data with box and whisker charts

Box plots come from descriptive statistics and are a useful way of depicting the distributions of categorical data using quartiles. Each box represents the values between the first and third quartiles of the data, with a line across the box at the median. Each whisker reaches out to demonstrate the extent to five interquartile ranges below and above the first and third quartiles:

```
In [31]: # create a box plot
# generate the series
np.random.seed(seedval)
dfb = pd.DataFrame(np.random.randn(10,5))
# generate the plot
dfb.boxplot(return_type='axes');
```

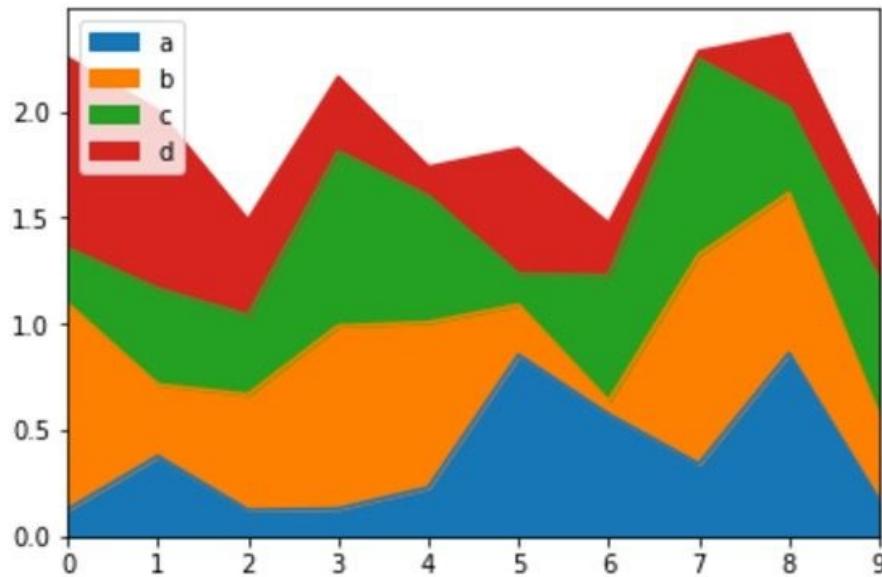


Demonstrating cumulative totals with area plots

Area plots are used to represent cumulative totals over time and to demonstrate the change in trends over time among related attributes. They can also be "stacked" to demonstrate representative totals across all variables.

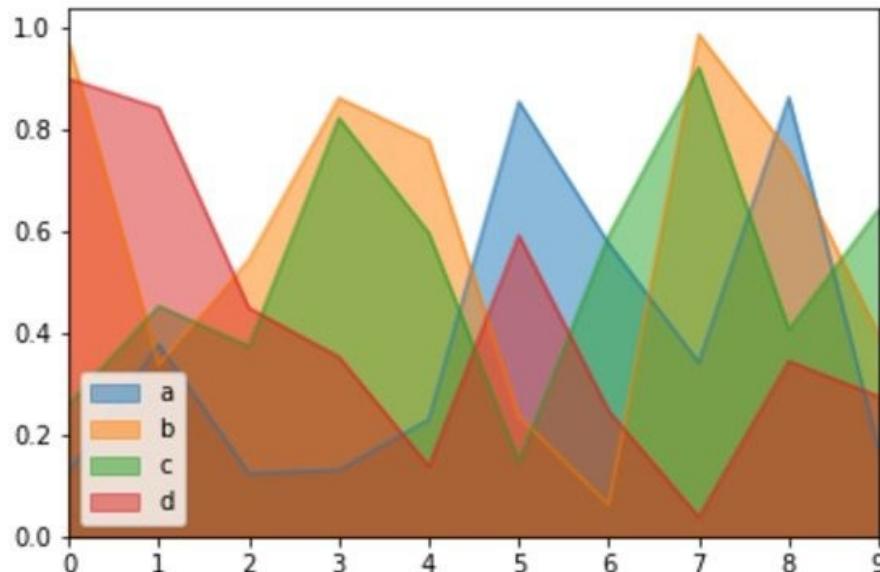
Area plots are generated by specifying `kind='area'`. A stacked area chart is the default:

```
In [32]: # create a stacked area plot
# generate a 4-column data frame of random data
np.random.seed(seedval)
dfa = pd.DataFrame(np.random.rand(10, 4),
                   columns=['a', 'b', 'c', 'd'])
# create the area plot
dfa.plot(kind='area');
```



Using `stacked=False` produces an unstacked area plot:

```
In [33]: # do not stack the area plot  
dfa.plot(kind='area', stacked=False);
```

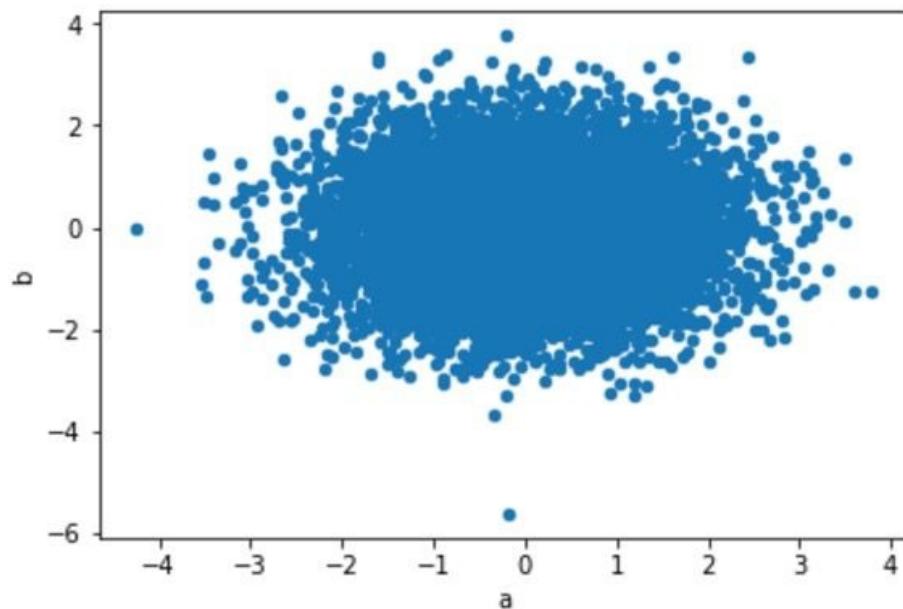


By default, unstacked plots have an alpha value of 0.5, so it is possible to see how multiple series of data overlap.

Relationships between two variables with scatter plots

A scatter plot displays the correlation between a pair of variables. A scatter plot can be created from `DataFrame` by using `.plot()` and specifying `kind='scatter'` as well as the x and y columns from the `DataFrame` source:

```
In [34]: # generate a scatter plot of two series of normally
# distributed random values
# we would expect this to cluster around 0,0
np.random.seed(111111)
sp_df = pd.DataFrame(np.random.randn(10000, 2),
                      columns=['a', 'b'])
sp_df.plot(kind='scatter', x='a', y='b');
```



More elaborate scatter plots can be created by dropping down into `matplotlib`. The following code demonstrates the use of Google stock data for year 2016, calculates delta in the closing price per day, and then renders close versus volume as bubbles of different sizes derived from the volume:

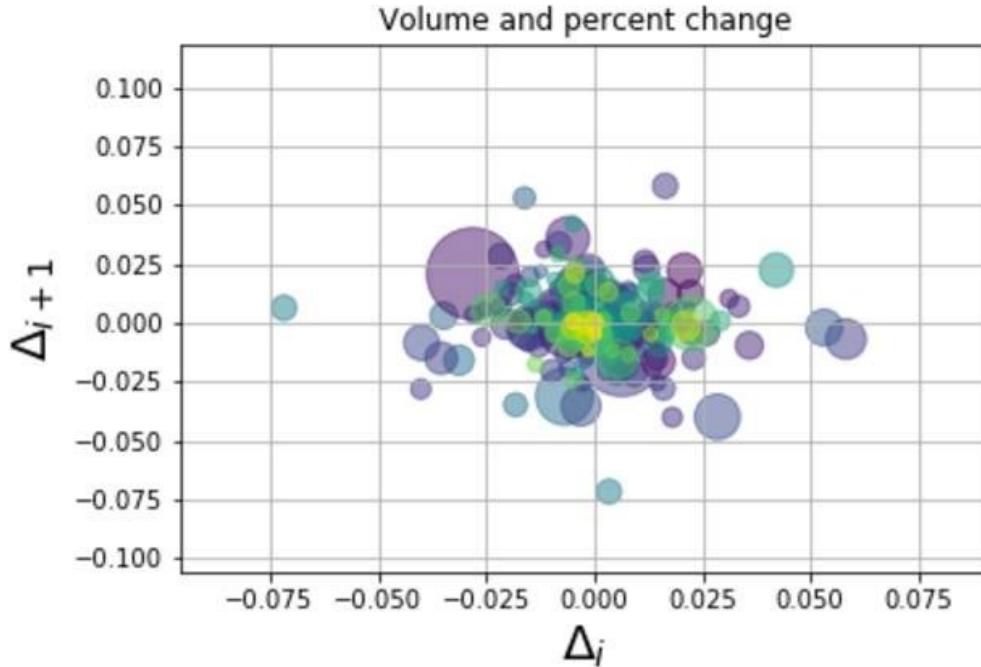
```
In [35]: import pandas_datareader as pdr
# get Google stock data from 1/1/2011 to 12/31/2011
start = datetime(2016, 1, 1)
end = datetime(2016, 12, 31)
stock_data = pdr.data.DataReader("MSFT", 'google', start, end)

# % change per day
delta = np.diff(stock_data["Close"])/stock_data["Close"][:-1]

# this calculates size of markers
volume = (15 * stock_data.Volume[:-2] / stock_data.Volume[0])**2
close = 0.003 * stock_data.Close[:-2] / 0.003 * stock_data.Open[:-2]

# generate scatter plot
fig, ax = plt.subplots()
ax.scatter(delta[:-1], delta[1:], c=close, s=volume, alpha=0.5)

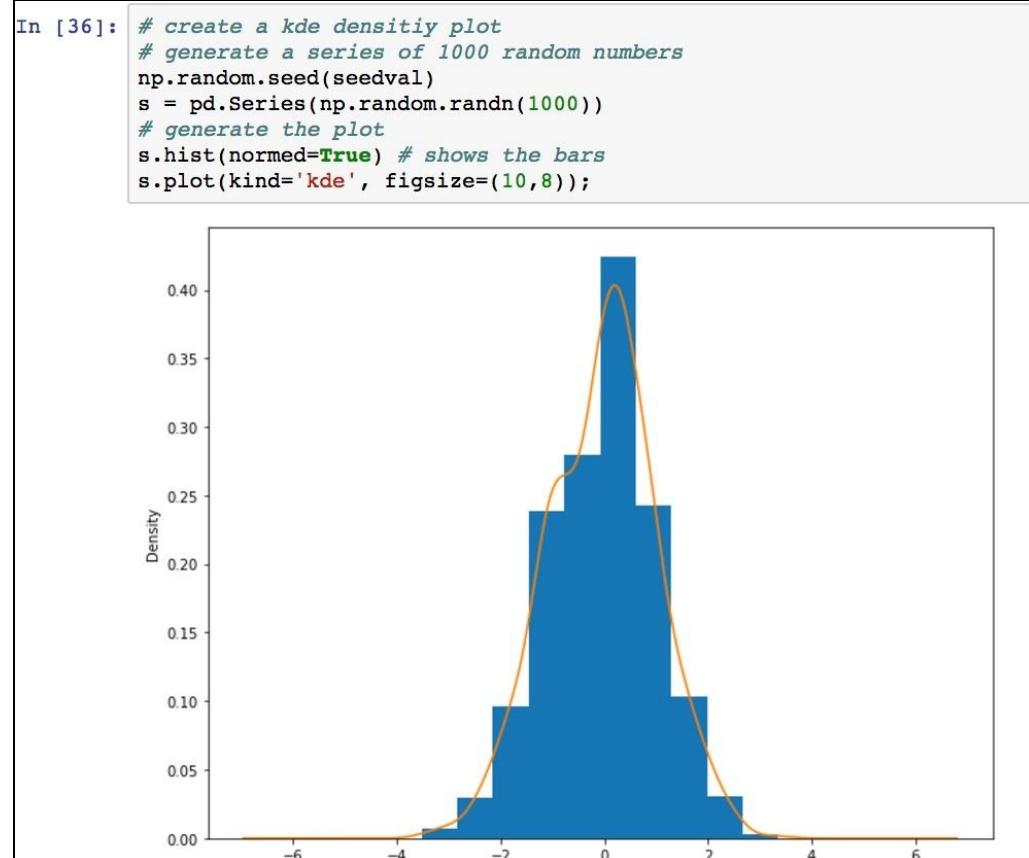
# add some labels and style
ax.set_xlabel(r'$\Delta_i$', fontsize=20)
ax.set_ylabel(r'$\Delta_{i+1}$', fontsize=20)
ax.set_title('Volume and percent change')
ax.grid(True);
```



Note the nomenclature for the x and y axes' labels, which creates a nice mathematical style for the labels thanks to matplotlib's internal LaTeX parser and layout engine.

Estimates of distribution with the kernel density plot

A kernel density estimate plot, instead of being a pure empirical representation of the data by estimating the true distribution of the data which and smooths the data into a continuous plot. Kernel density estimation plots can be created by using the `.plot()` method and setting `kind='kde'`. The following code generates a normal distributed set of numbers, displays it as a histogram, and overlays the `kde` plot:



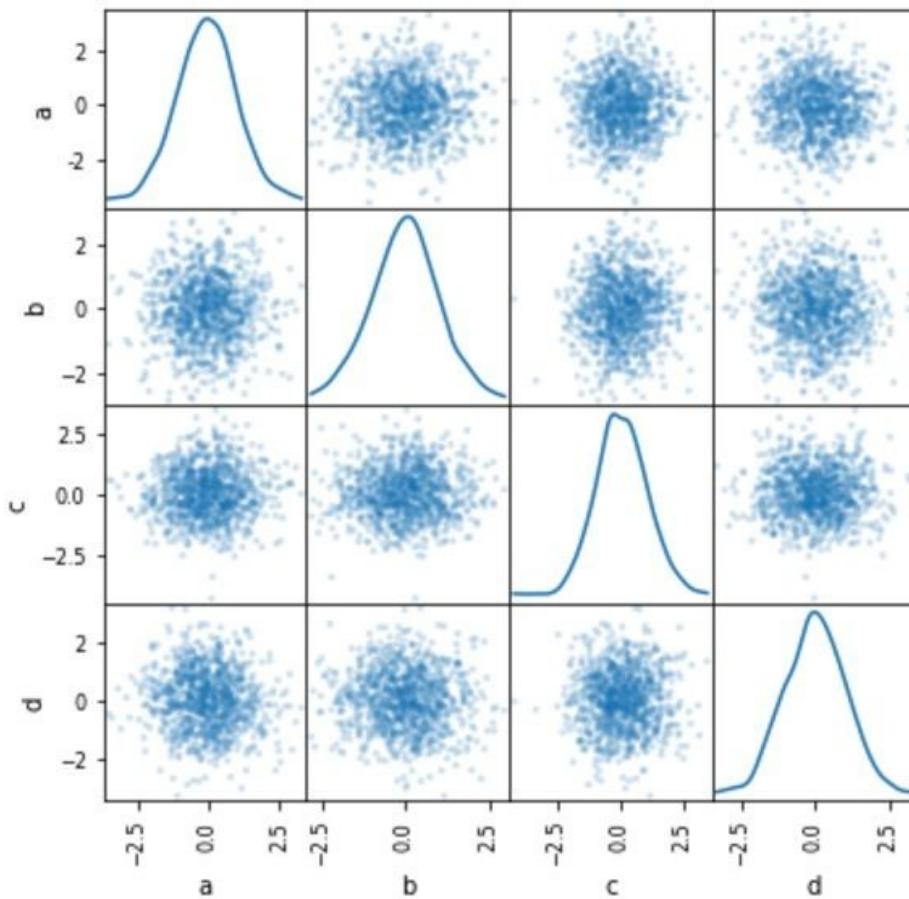
Correlations between multiple variables with the scatter plot matrix

A scatter plot matrix is a popular way of determining whether there is a linear correlation between multiple variables. This code creates a scatter plot matrix with random values, and renders a scatter plot for each variable combination as well as a kde graph for each variable along the diagonal:

```
In [37]: # create a scatter plot matrix
# import this class
from pandas.plotting import scatter_matrix

# generate DataFrame with 4 columns of 1000 random numbers
np.random.seed(111111)
df_spm = pd.DataFrame(np.random.randn(1000, 4),
                      columns=['a', 'b', 'c', 'd'])

# create the scatter matrix
scatter_matrix(df_spm, alpha=0.2, figsize=(6, 6), diagonal='kde');
```



We will see this plot again when it is applied to finance, when correlations of various stocks are calculated.

Strengths of relationships in multiple variables with heatmaps

A heatmap is a graphical representation of data wherein values within a matrix are represented by colors. This is an effective means to show relationships of values that are measured at the intersection of two variables.

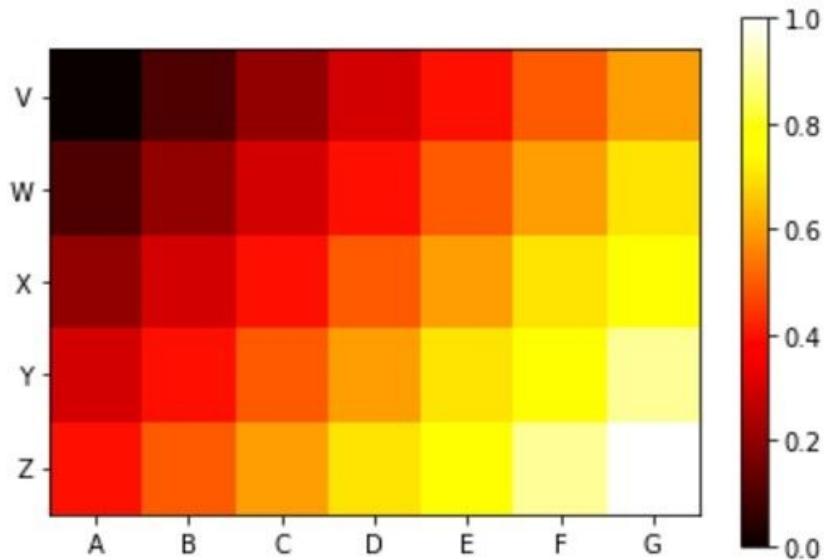
A common scenario is to have the values in the matrix normalized to 0.0 through 1.0 and have the intersections between a row and column represent the correlation between the two variables. Values with less correlation (0.0) are the darkest and those with the highest correlation (1.0) are white.

Heatmaps are easily created with pandas and `matplotlib` using the `.imshow()` function:

```
In [38]: # create a heatmap
# start with data for the heatmap
s = pd.Series([0.0, 0.1, 0.2, 0.3, 0.4],
              ['V', 'W', 'X', 'Y', 'Z'])
heatmap_data = pd.DataFrame({'A' : s + 0.0,
                             'B' : s + 0.1,
                             'C' : s + 0.2,
                             'D' : s + 0.3,
                             'E' : s + 0.4,
                             'F' : s + 0.5,
                             'G' : s + 0.6
                            })
heatmap_data
```

```
Out[38]:      A      B      C      D      E      F      G
V  0.0  0.1  0.2  0.3  0.4  0.5  0.6
W  0.1  0.2  0.3  0.4  0.5  0.6  0.7
X  0.2  0.3  0.4  0.5  0.6  0.7  0.8
Y  0.3  0.4  0.5  0.6  0.7  0.8  0.9
Z  0.4  0.5  0.6  0.7  0.8  0.9  1.0
```

```
In [39]: # generate the heatmap
plt.imshow(heatmap_data, cmap='hot', interpolation='none')
plt.colorbar() # add the scale of colors bar
# set the labels
plt.xticks(range(len(heatmap_data.columns)), heatmap_data.columns)
plt.yticks(range(len(heatmap_data)), heatmap_data.index);
```



Heatmaps will also be revisited to demonstrate correlations in the next chapter.

Manually rendering multiple plots in a single chart

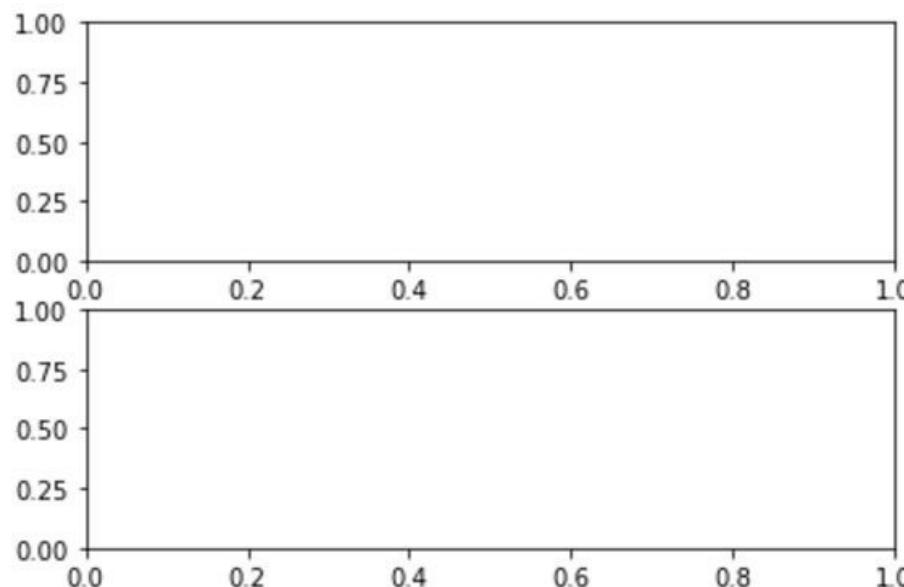
It is often useful to contrast data by displaying multiple plots next to each other. We have seen that pandas does this automatically for several types of graphs. It is also possible to manually render multiple plots on the same canvas.

To render multiple subplots on a canvas with `matplotlib`, make multiple calls to `plt.subplot2grid()`. Each time pass the size of the grid the subplot is to be located on (`shape=(height, width)`) and the location on the grid of the upper-left location of the subplot (`loc=(row, column)`). The dimensions are in the overall number of columns and not pixels.

The return value from each call to `plt.subplot2grid()` is a different `AxesSubplot` object that can be used to specify the position of the rendering of the subplot.

The following code demonstrates this by creating a plot based on two rows and one column (`shape=(2,1)`). The first subplot, referenced by `ax1`, is located in the first row (`loc=(0,0)`), and the second, referenced by `ax2`, is in the second row (`loc=(1,0)`):

```
In [40]: # create two sub plots on the new plot using a 2x1 grid
# ax1 is the upper row
ax1 = plt.subplot2grid(shape=(2,1), loc=(0,0))
# and ax2 is in the lower row
ax2 = plt.subplot2grid(shape=(2,1), loc=(1,0))
```

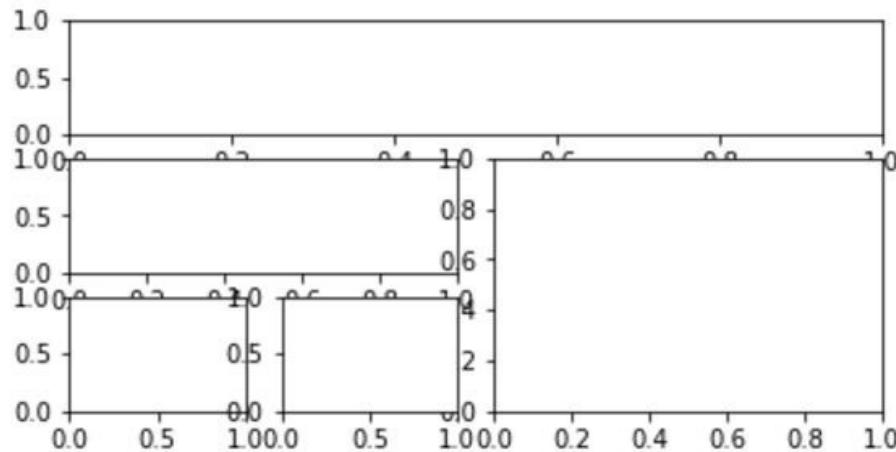


The subplots have been created but we have not drawn into either yet.

The size of any subplot can be specified using the `rowspan` and `colspan` parameters in each call to `plt.subplot2grid()`. This means of specifying spans is a lot like specifying spans in HTML tables. This code demonstrates using spans to create a more complicated layout of five plots by specifying different row,

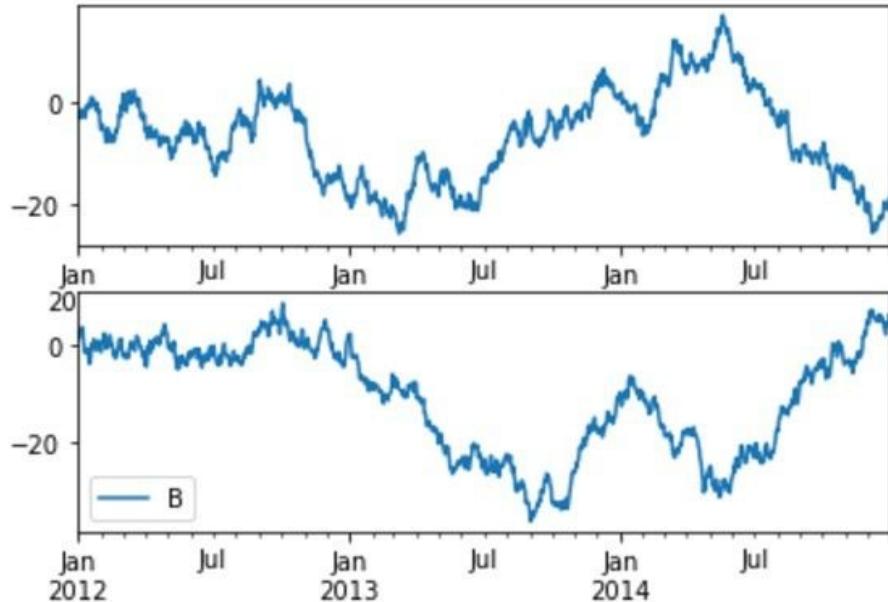
column, and spans for each:

```
In [41]: # layout sub plots on a 4x4 grid
# ax1 on top row, 4 columns wide
ax1 = plt.subplot2grid((4,4), (0,0), colspan=4)
# ax2 is row 2, leftmost and 2 columns wide
ax2 = plt.subplot2grid((4,4), (1,0), colspan=2)
# ax3 is 2 cols wide and 2 rows high, starting
# on second row and the third column
ax3 = plt.subplot2grid((4,4), (1,2), colspan=2, rowspan=2)
# ax4 1 high 1 wide, in row 4 column 0
ax4 = plt.subplot2grid((4,4), (2,0))
# ax4 1 high 1 wide, in row 4 column 1
ax5 = plt.subplot2grid((4,4), (2,1));
```



To render into a specific subplot, pass the target axis object to `.plot()` using the `ax` parameter. The following code demonstrates this by extracting each series from the random walk created at the beginning of the chapter, and then by rendering each within different subplots:

```
In [42]: # demonstrating drawing into specific sub-plots
# generate a layout of 2 rows 1 column
# create the subplots, one on each row
ax5 = plt.subplot2grid((2,1), (0,0))
ax6 = plt.subplot2grid((2,1), (1,0))
# plot column 0 of walk_df into top row of the grid
walk_df[walk_df.columns[0]].plot(ax = ax5)
# and column 1 of walk_df into bottom row
walk_df[[walk_df.columns[1]]].plot(ax = ax6);
```



Using this technique, we can perform combinations of different series of data, such as a stock close versus volume graph. Given the data we read during a previous example for Google, this code will plot the volume versus the closing price:

```
In [43]: # draw the close on the top chart
top = plt.subplot2grid((4,4), (0, 0), rowspan=3, colspan=4)
top.plot(stock_data.index, stock_data['Close'], label='Close')
plt.title('Google Opening Stock Price 2001')

# draw the volume chart on the bottom
bottom = plt.subplot2grid((4,4), (3,0), rowspan=1, colspan=4)
bottom.bar(stock_data.index, stock_data['Volume'])
plt.title('Google Trading Volume')

# set the size of the plot
plt.gcf().set_size_inches(15,8)
```



Summary

In this chapter, we examined many of the most common means of visualizing data from pandas. Visualization of your data is one of the best ways to quickly understand the story that is being told within the data. Python, pandas, and `matplotlib` (and even a few other libraries) provide a means of very quickly getting to the underlying message and displaying it beautifully too, with only a few lines of code.

In the next and final chapter of the book, we will examine the application of pandas in finance, particularly stock price analysis.

Historical Stock Price Analysis

In this final chapter, we will use pandas to perform various financial analyses of stock data obtained from Google Finance. This will also cover several topics in financial analysis. The emphasis will be on using pandas to derive practical time-series stock data and not on details of the financial theory. However, we will cover many useful topics and learn how easy it is to apply pandas to this domain and to others.

Specifically, in this chapter, we will progress through the following tasks:

- Fetching and organizing stock data from Google Finance
- Plotting time-series prices
- Plotting volume-series data
- Calculating simple daily percentage change
- Calculating simple daily cumulative returns
- Resampling data from daily to monthly returns
- Analyzing distribution of returns
- Performing moving-average calculations
- Comparing average daily returns across stocks
- Correlating stocks based on the daily percentage change of closing prices
- Calculating stock volatility
- Visualizing risk relative to expected returns

Setting up the IPython notebook

The examples in this chapter will all be based on the following imports and default settings:

```
In [1]: # import numpy and pandas
import numpy as np
import pandas as pd

# used for dates
import datetime
from datetime import datetime, date

# Set formattign options
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# bring in matplotlib for graphics
import matplotlib.pyplot as plt
%matplotlib inline
```


Obtaining and organizing stock data from Google

Our first task is to write a couple of functions that help us with retrieving stock data from Google Finance. We have already seen that this data can be read using a pandas `DataReader` object, but we will need to organize the data a little differently than how it is provided by Google Finance, as we are going to perform various pivots of this information later.

The following function will get all the Google Finance data for a specific stock between the two specified dates, and also add the stock's symbol in a column (this will be needed later for pivots).

```
In [2]: # import data reader package
import pandas_datareader as pdr

# read data from Yahoo! Finance for a specific
# stock specified by ticker and between the start and end dates
def get_stock_data(ticker, start, end):
    # read the data
    data = pdr.data.DataReader(ticker, 'google', start, end)

    # rename this column
    data.insert(0, "Ticker", ticker)
    return data
```

The data will consist of a fixed 3-year window, spanning the years of 2012 through 2014. The following reads data for this 3-year period for the MSFT ticker:

```
In [3]: # request the three years of data for MSFT
start = datetime(2012, 1, 1)
end = datetime(2014, 12, 31)
get_stock_data("MSFT", start, end)[:5]
```

```
Out[3]:
```

Date	Ticker	Open	High	Low	Close	Volume
2012-01-03	MSFT	26.55	26.96	26.39	26.76	64735391
2012-01-04	MSFT	26.82	27.47	26.78	27.40	80519402
2012-01-05	MSFT	27.38	27.73	27.29	27.68	56082205
2012-01-06	MSFT	27.53	28.19	27.52	28.10	99459469
2012-01-09	MSFT	28.05	28.10	27.72	27.74	59708266

Now that we have a function that can get data for a single ticker, it will be convenient to have a function that can read the data for multiple tickers and return them all in a single data structure. The following code performs this task:

```
In [4]: # gets data for multiple stocks
# tickers: a list of stock symbols to fetch
# start and end are the start end end dates
def get_data_for_multiple_stocks(tickers, start, end):
    # we return a dictionary
    stocks = dict()
    # loop through all the tickers
    for ticker in tickers:
        # get the data for the specific ticker
        s = get_stock_data(ticker, start, end)
        # add it to the dictionary
        stocks[ticker] = s
    # return the dictionary
    return stocks
```

The examples in this chapter will use historical quotes for **Apple (AAPL)**, **Microsoft (MSFT)**, **General Electric (GE)**, **IBM (IBM)**, **American Airlines (AA)**, **Delta Airlines (DAL)**, **United Airlines (UAL)**, **Pepsi (PEP)**, and **Coca Cola (KO)**.

These stocks were chosen deliberately to have a sample of multiple stocks in each of the three different sectors: technology, airlines, and soft drinks. The purpose of this is to demonstrate how to derive correlations in various stock price measurements over the selected time-period among the stocks in similar sectors, and to also demonstrate the difference in stocks between sectors.

We can read these with the following:

```
In [5]: # get the data for all the stocks that we want
raw = get_data_for_multiple_stocks(
    ["MSFT", "AAPL", "GE", "IBM", "AA", "DAL", "UAL", "PEP", "KO"],
    start, end)
```



Note: During testing, it has been identified that depending on your location, this may cause some errors due to URL accessibility.

```
In [6]: # take a peek at the data for MSFT
raw['MSFT'][:5]
```

	Ticker	Open	High	Low	Close	Volume
Date						
2012-01-03	MSFT	26.55	26.96	26.39	26.76	64735391
2012-01-04	MSFT	26.82	27.47	26.78	27.40	80519402
2012-01-05	MSFT	27.38	27.73	27.29	27.68	56082205
2012-01-06	MSFT	27.53	28.19	27.52	28.10	99459469
2012-01-09	MSFT	28.05	28.10	27.72	27.74	59708266

We will be particularly interested in the close price values in the close column. However, it would have

been more convenient for us if we had a `DataFrame` object indexed by date, and where each column was the price for a specific stock, with the rows being the close value for that ticker at that date. This can be done by pivoting the data, and is the reason for adding the Ticker column when reading the data. The following function will do this for us:

```
In [7]: # given the dictionary of data frames,
# pivots a given column into values with column
# names being the stock symbols
def pivot_tickers_to_columns(raw, column):
    items = []
    # loop through all dictionary keys
    for key in raw:
        # get the data for the key
        data = raw[key]
        # extract just the column specified
        subset = data[["Ticker", column]]
        # add to items
        items.append(subset)

    # concatenate all the items
    combined = pd.concat(items)
    # reset the index
    ri = combined.reset_index()
    # return the pivot
    return ri.pivot("Date", "Ticker", column)
```

The following uses the function to pivot the data to the new organization:

```
In [8]: # do the pivot
close_px = pivot_tickers_to_columns(raw, "Close")
# peek at the result
close_px[:5]
```

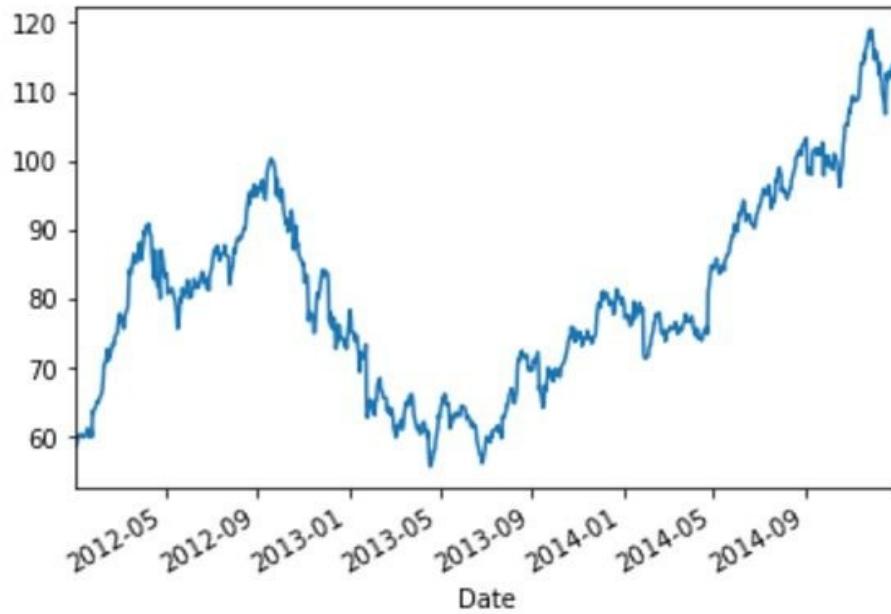
```
Out[8]: Ticker      AAPL      DAL       GE     ...
          Date
          2012-01-03  58.75    8.04    18.36    ...
          2012-01-04  59.06    8.01    18.56    ...
          2012-01-05  59.72    8.33    18.55    ...
          2012-01-06  60.34    8.32    18.65    ...
          2012-01-09  60.25    8.28    18.86    ...
[5 rows x 8 columns]
```

The close values for all stocks are now values in a column for each respective stock. In this format, it will be simple to compare the closing prices of each stock against the others.

Plotting time-series prices

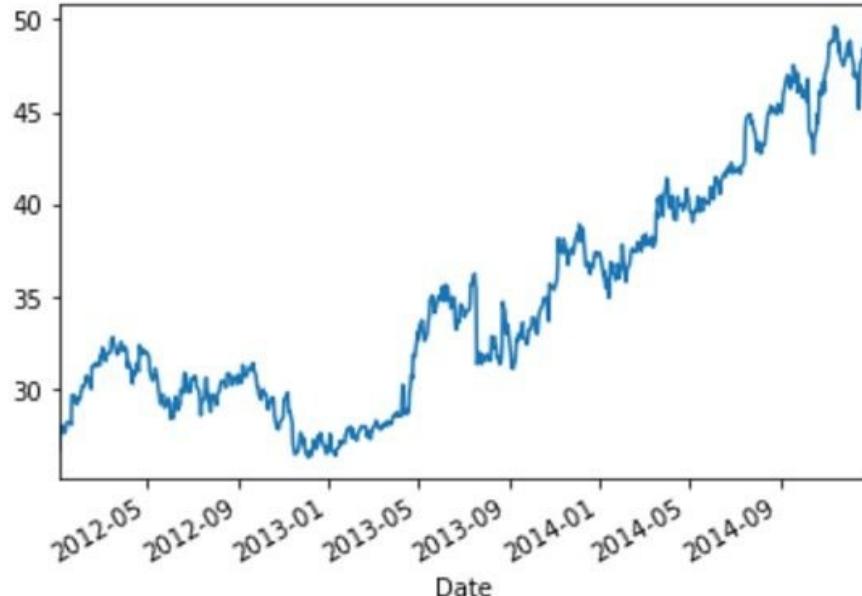
Now let's perform a graphical comparison of the closing values for AAPL and MSFT. The following plots the adjusted closing price for AAPL:

```
In [9]: # plot the closing prices of AAPL  
close_px['AAPL'].plot();
```



The following shows MSFT:

```
In [10]: # plot the closing prices of MSFT  
close_px['MSFT'].plot();
```



Both sets of closing values can easily be displayed on a single chart to give a side-by-side (or one over the other) comparison:

```
In [11]: # plot MSFT vs AAPL on the same chart  
close_px[['MSFT', 'AAPL']].plot();
```



Plotting volume-series data

Volume data can be plotted using bar charts. We first need to get the volume data, which can be done using the `pivot_tickers_to_columns()` function created earlier:

```
In [12]: # pivot the volume data into columns
volumes = pivot_tickers_to_columns(raw, "Volume")
volumes.tail()
```

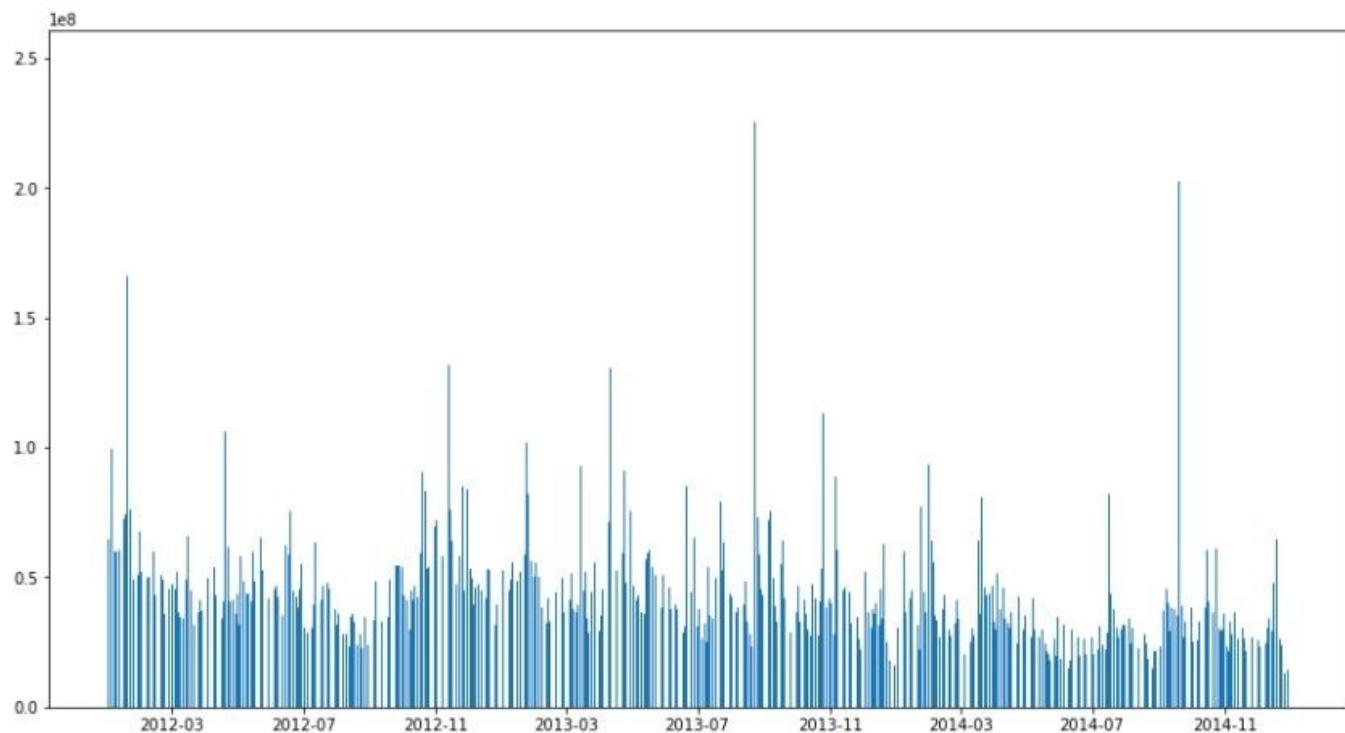
```
Out[12]: Ticker          AAPL        DAL        GE      ...
Date
2014-12-24  14479611  4303380  17865138  ...
2014-12-26  33720951  5303077  14978547  ...
2014-12-29  27598920  6659665  20858159  ...
2014-12-30  29881477  7318917  22184545  ...
2014-12-31  41403351  7801706  28206452  ...

Ticker          MSFT        PEP        UAL
Date
2014-12-24  11442790  1608616  2714807
2014-12-26  13197817  1492689  3062153
2014-12-29  14439518  2453829  2874473
2014-12-30  16384692  2134434  2644611
2014-12-31  21552450  3727376  4451235

[5 rows x 8 columns]
```

We can now use this data to plot a bar chart. The following plots the volume for MSFT:

```
In [13]: # plot the volume for MSFT
msft_volume = volumes[["MSFT"]]
plt.bar(msft_volume.index, msft_volume["MSFT"])
plt.gcf().set_size_inches(15,8)
```



A common type of financial graph plots a stock volume relative to its closing price. The following sample creates this type of visualization:

```
In [14]: # draw the price history on the top
top = plt.subplot2grid((4,4), (0, 0), rowspan=3, colspan=4)
top.plot(close_px['MSFT'].index, close_px['MSFT'],
         label='MSFT Close')
plt.title('MSFT Close Price 2012 - 2014')
plt.legend(loc=2)

# and the volume along the bottom
bottom = plt.subplot2grid((4,4), (3,0), rowspan=1, colspan=4)
bottom.bar(msft_volume.index, msft_volume['MSFT'])
plt.title('Microsoft Trading Volume 2012 - 2014')
plt.subplots_adjust(hspace=0.75)
plt.gcf().set_size_inches(15,8)
```



Calculating the simple daily percentage change in closing price

The simple daily percentage change in closing price (without dividends and other factors) is the percentage change in the value of a stock over a single day of trading. It is defined by the following formula:

$$r_t = \frac{p_t}{p_{t-1}} - 1$$

This can be easily calculated in pandas using `.shift()`:

```
In [15]: # calculate daily percentage change
daily_pc = close_px / close_px.shift(1) - 1
daily_pc[:5]
```

```
Out[15]: Ticker      AAPL      DAL      GE      ...
          Date
2012-01-03    NaN      NaN      NaN      ...
2012-01-04  0.005277 -0.003731  0.010893      ...
2012-01-05  0.011175  0.039950 -0.000539      ...
2012-01-06  0.010382 -0.001200  0.005391      ...
2012-01-09 -0.001492 -0.004808  0.011260      ...

          Ticker      MSFT      PEP      UAL
          Date
2012-01-03    NaN      NaN      NaN
2012-01-04  0.023916  0.005120 -0.020106
2012-01-05  0.010219 -0.007791 -0.007019
2012-01-06  0.015173 -0.012534 -0.009788
2012-01-09 -0.012811  0.005200 -0.015376

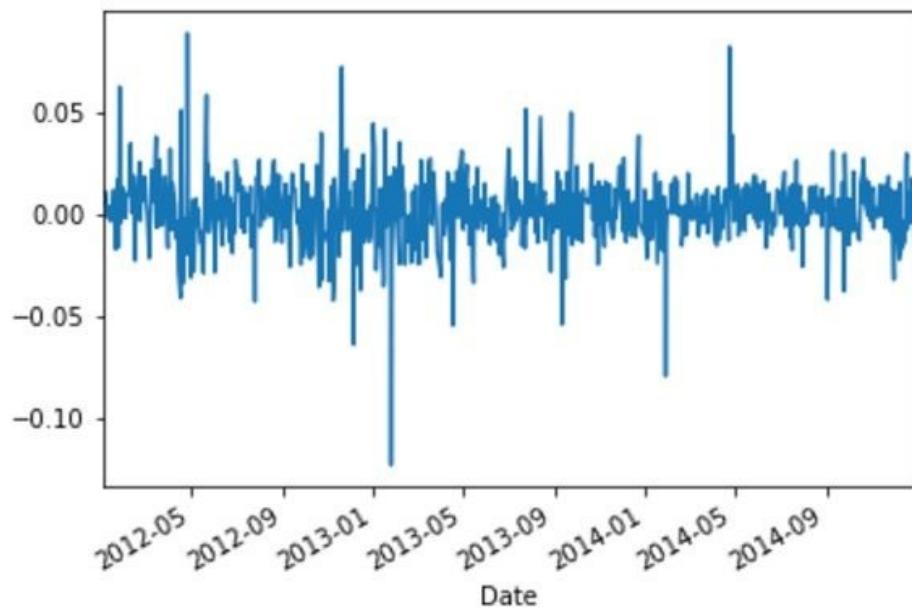
[5 rows x 8 columns]
```

A quick check confirms that the return for AAPL on 2011-09-08 is correct:

```
In [16]: # check the percentage on 2012-01-05
close_px.loc['2012-01-05']['AAPL'] / \
            close_px.loc['2012-01-04']['AAPL'] - 1
```

```
Out[16]: 0.011175076193701283
```

```
In [17]: # plot daily percentage change for AAPL  
daily_pc[ "AAPL" ].plot();
```



A plot of daily percentage change will tend to look like noise, as shown in the preceding rendering. However, when we use the cumulative product of these values, known as the daily cumulative return, it is possible to see how the value of the stock changes over time. That is our next task.

Calculating simple daily cumulative returns of a stock

The simple cumulative daily return is calculated by taking the cumulative product of the daily percentage change. This calculation is represented by the following equation:

$$i_t = (1 + r_t) \cdot i_{t-1}, \quad i_0 = 1$$

This is calculated succinctly using the `.cumprod()` method:

```
In [18]: # calculate daily cumulative return
daily_cr = (1 + daily_pc).cumprod()
daily_cr[:5]
```

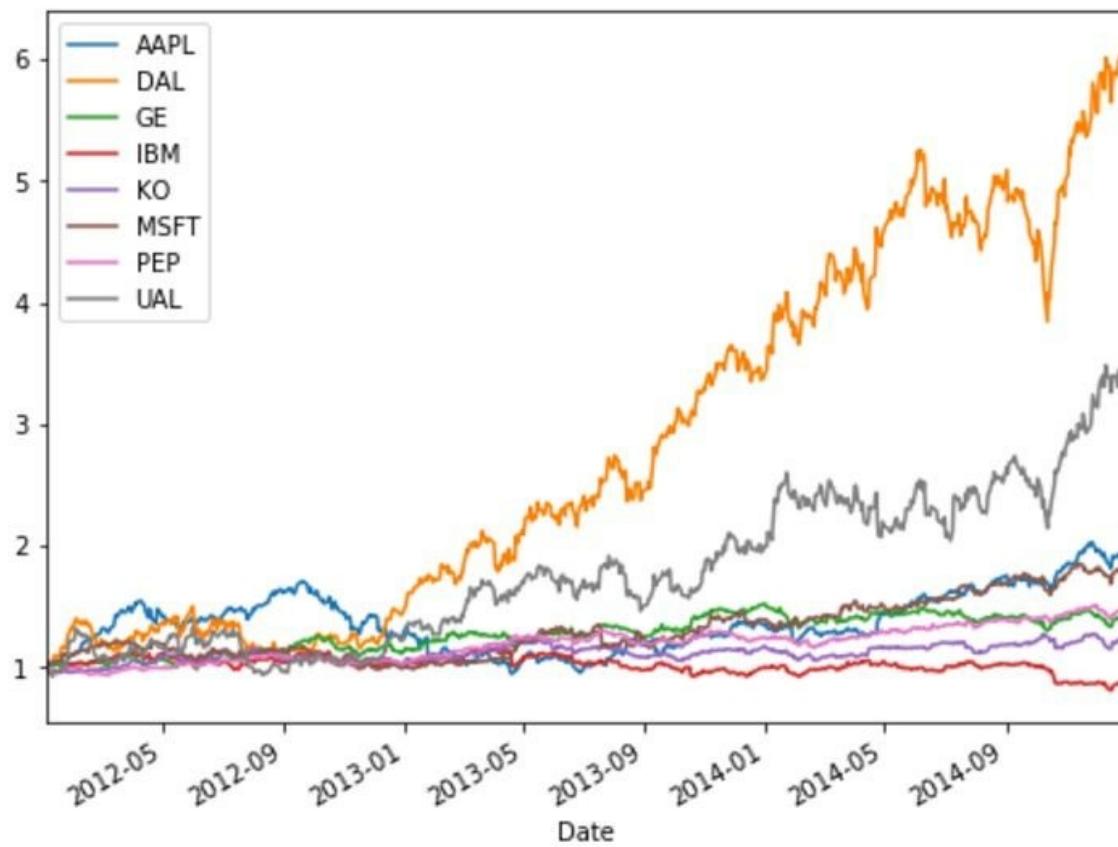
```
Out[18]: Ticker      AAPL      DAL      GE      ...
          Date
2012-01-03    NaN      NaN      NaN      ...
2012-01-04  1.005277  0.996269  1.010893  ...
2012-01-05  1.016511  1.036070  1.010349  ...
2012-01-06  1.027064  1.034826  1.015795  ...
2012-01-09  1.025532  1.029851  1.027233  ...

          Ticker      MSFT      PEP      UAL
          Date
2012-01-03    NaN      NaN      NaN
2012-01-04  1.023916  1.005120  0.979894
2012-01-05  1.034380  0.997289  0.973016
2012-01-06  1.050075  0.984789  0.963492
2012-01-09  1.036622  0.989910  0.948677

[5 rows x 8 columns]
```

It is now possible to plot cumulative returns to see how the various stocks compare in value over time:

```
In [19]: # plot all the cumulative returns to get an idea  
# of the relative performance of all the stocks  
daily_cr.plot(figsize=(8,6))  
plt.legend(loc=2);
```



Resampling data from daily to monthly returns

To calculate the monthly rate of return, we can use a little pandas magic and resample the original daily returns. During this process, we will also need to throw out the days that are not an end of month as well as forward fill any missing values. This can be done using the `.ffill()` on the result of the resampling:

```
In [20]: # resample to end of month and forward fill values
monthly = close_px.asfreq('M').ffill()
monthly[:5]
```

```
Out[20]: Ticker      AAPL      DAL      GE     ...
          Date
2012-01-31  65.21   10.55   18.71   ...
2012-02-29  77.49    9.81   19.05   ...
2012-03-31  77.49    9.81   19.05   ...
2012-04-30  83.43   10.96   19.58   ...
2012-05-31  82.53   12.10   19.09   ...

          Ticker      UAL
          Date
2012-01-31  23.10
2012-02-29  20.65
2012-03-31  20.65
2012-04-30  21.92
2012-05-31  25.17

[5 rows x 8 columns]
```

Note the date of the entries and that they are now all month-end dates. Values have not changed, as the resample only selects the dates at the end of the month, or fills with the value prior to that date if it did not exist in the source.

Now we can use this to calculate the monthly percentage changes:

```
In [21]: # calculate the monthly percentage changes
monthly_pc = monthly / monthly.shift(1) - 1
monthly_pc[:5]
```

```
Out[21]: Ticker      AAPL      DAL      GE      ...
Date
2012-01-31      NaN      NaN      NaN      ...
2012-02-29  0.188315 -0.070142  0.018172      ...
2012-03-31  0.000000  0.000000  0.000000      ...
2012-04-30  0.076655  0.117227  0.027822      ...
2012-05-31 -0.010787  0.104015 -0.025026      ...

Ticker      MSFT      PEP      UAL
Date
2012-01-31      NaN      NaN      NaN
2012-02-29  0.074839 -0.041571 -0.106061
2012-03-31  0.000000  0.000000  0.000000
2012-04-30  0.008822  0.048618  0.061501
2012-05-31 -0.088382  0.028030  0.148266

[5 rows x 8 columns]
```

This now allows the calculation of monthly cumulative returns:

```
In [22]: # calculate monthly cumulative return
monthly_cr = (1 + monthly_pc).cumprod()
monthly_cr[:5]
```

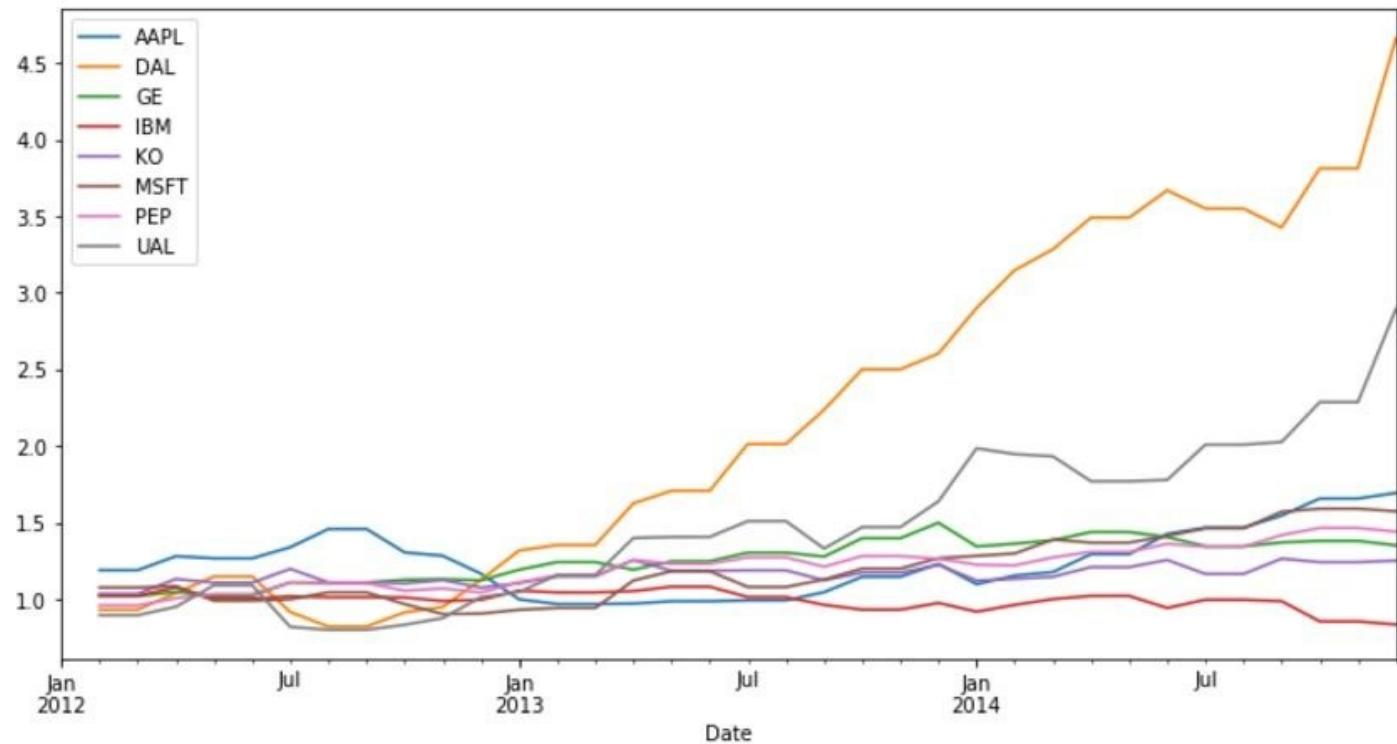
```
Out[22]: Ticker      AAPL      DAL      GE      ...
Date
2012-01-31      NaN      NaN      NaN      ...
2012-02-29  1.188315  0.929858  1.018172      ...
2012-03-31  1.188315  0.929858  1.018172      ...
2012-04-30  1.279405  1.038863  1.046499      ...
2012-05-31  1.265603  1.146919  1.020310      ...

Ticker      MSFT      PEP      UAL
Date
2012-01-31      NaN      NaN      NaN
2012-02-29  1.074839  0.958429  0.893939
2012-03-31  1.074839  0.958429  0.893939
2012-04-30  1.084321  1.005025  0.948918
2012-05-31  0.988486  1.033196  1.089610

[5 rows x 8 columns]
```

The monthly returns have the following visualization:

```
In [23]: # plot the monthly cumulative returns  
monthly_cr.plot(figsize=(12,6))  
plt.legend(loc=2);
```



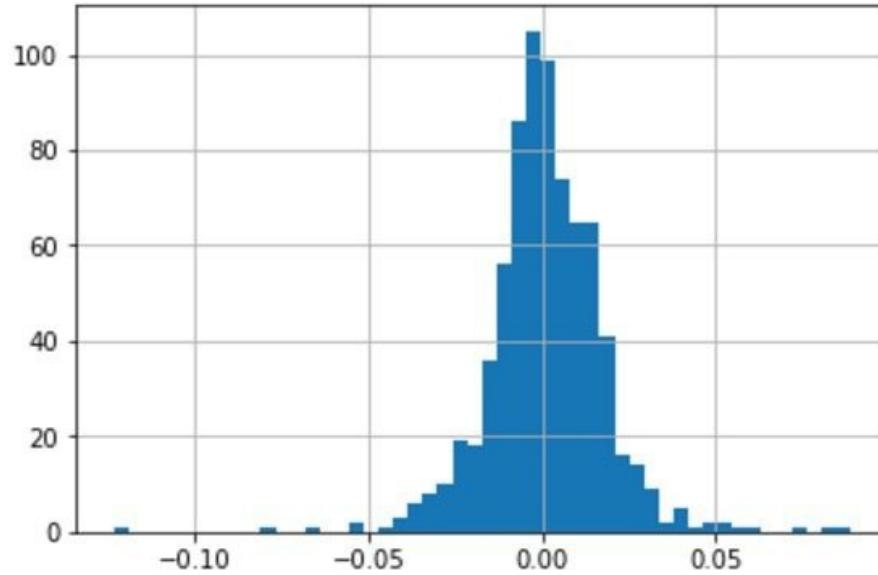
This looks like the daily returns, but overall, it is not as smooth. This is because it uses roughly a 30th of the data and is tied to the end of month.

Analyzing distribution of returns

It is possible to get a feel for the difference in distribution of the daily percentage changes for a specific stock by plotting that data in a histogram. A trick with generating histograms for data such as daily returns is to select the number of bins to lump values into. The example will use 50 bins, which gives a good feel for the distribution of daily changes across three years of data.

The following shows you the distribution of the daily percentage change for AAPL:

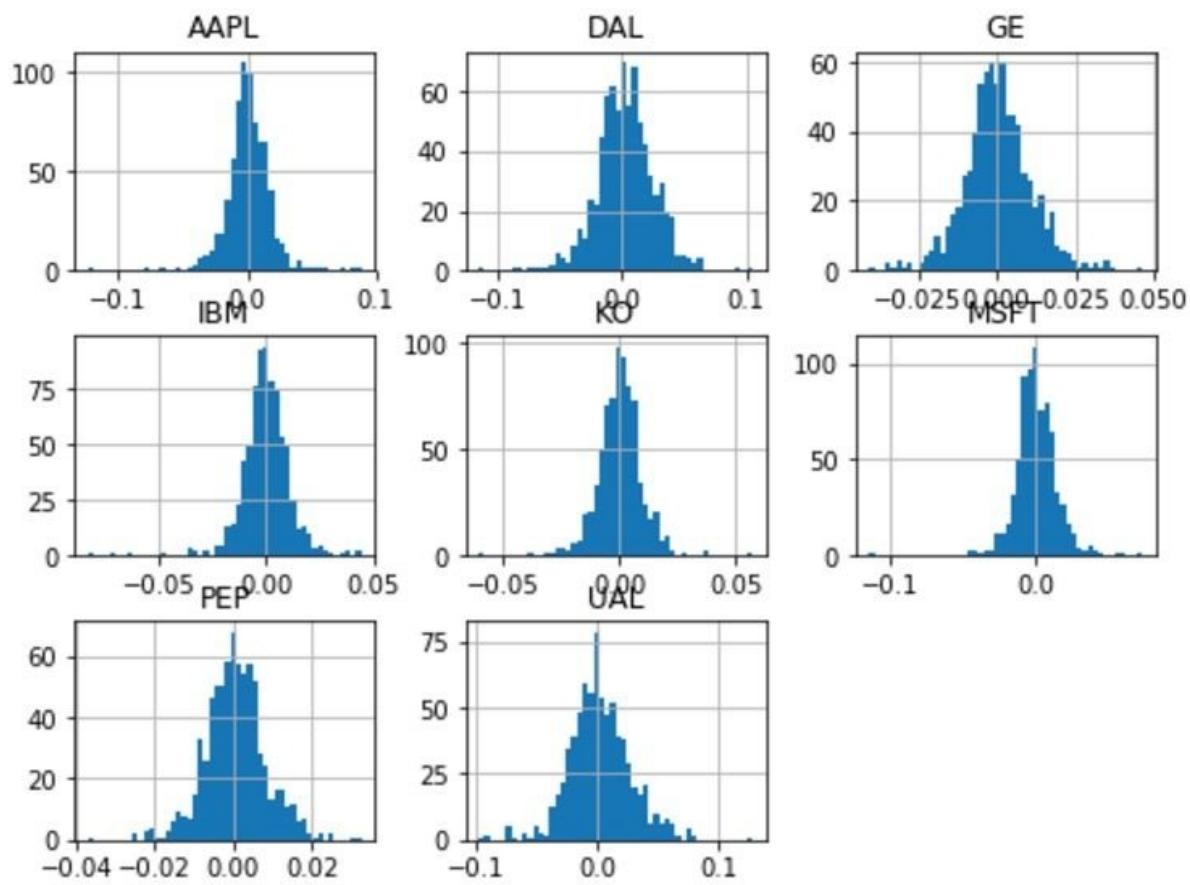
```
In [24]: # histogram of the daily percentage change for AAPL  
aapl = daily_pc['AAPL']  
aapl.hist(bins=50);
```



This visualization tells us several things. First, most of the daily movements center around 0.0. Second, there is a small amount of skew to the left, but the data appears relatively symmetric.

We can plot the histograms of the daily percentage change for all the stocks in a single histogram matrix plot. This gives us a means to quickly determine the differences in stock behavior over these three years:

```
In [25]: # matrix of all stocks daily & changes histograms  
daily_pc.hist(bins=50, figsize=(8,6));
```



The labels on the axis are a bit squished together, but it's the histogram shape that is important.

From this chart, the difference in the performances of these nine stocks can be seen, particularly, the skewedness (more exceptional values on one side of the mean). We can also see the difference in the overall distribution widths, which gives a quick view of stocks that are more or less volatile.

Performing a moving-average calculation

The moving average of a stock can be calculated using `.rolling().mean()`. The moving average will give you a sense of the performance of a stock over a given time-period, by eliminating "noise" in the performance of the stock. The larger the moving window, the smoother and less random the graph will be, but at the expense of accuracy.

The following sample calculates the moving average for MSFT over 30- and 90-day periods, using the daily close value. The difference in the reduction of noise can be easily determined from the visual:

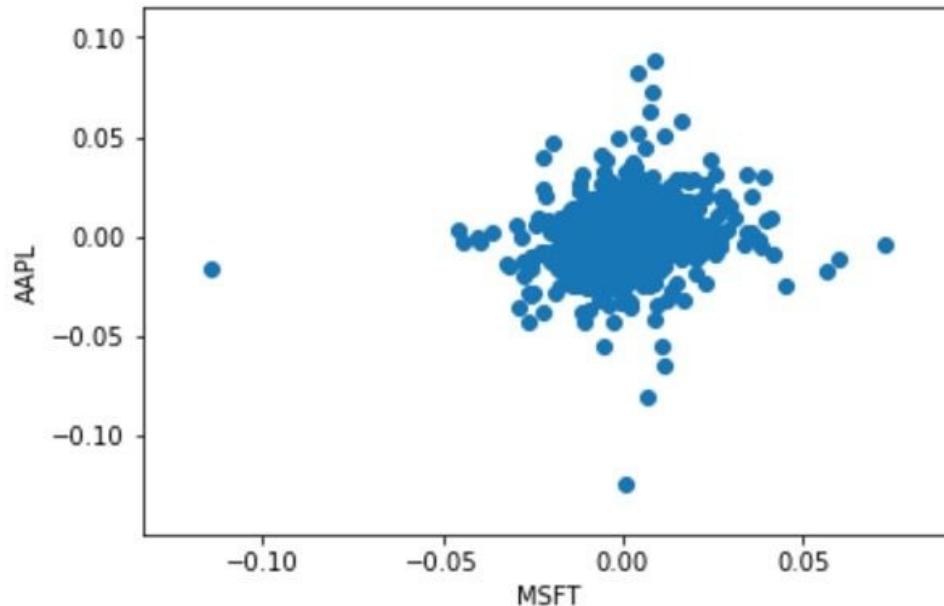
```
In [26]: # extract just MSFT close
msft_close = close_px[['MSFT']]['MSFT']
# calculate the 30 and 90 day rolling means
ma_30 = msft_close.rolling(window=30).mean()
ma_90 = msft_close.rolling(window=90).mean()
# compose into a DataFrame that can be plotted
result = pd.DataFrame({'Close': msft_close,
                       '30_MA_Close': ma_30,
                       '90_MA_Close': ma_90})
# plot all the series against each other
result.plot(title="MSFT Close Price")
plt.gcf().set_size_inches(12,8)
```



Comparison of average daily returns across stocks

A scatter plot is a very effective means of being able to visually determine the relationship between the rates of change in stock prices between two stocks. The following graphs the relationship of the daily percentage change in the closing price between MSFT and AAPL:

```
In [27]: # plot the daily percentage change of MSFT vs AAPL  
plt.scatter(daily_pc['MSFT'], daily_pc['AAPL'])  
plt.xlabel('MSFT')  
plt.ylabel('AAPL');
```

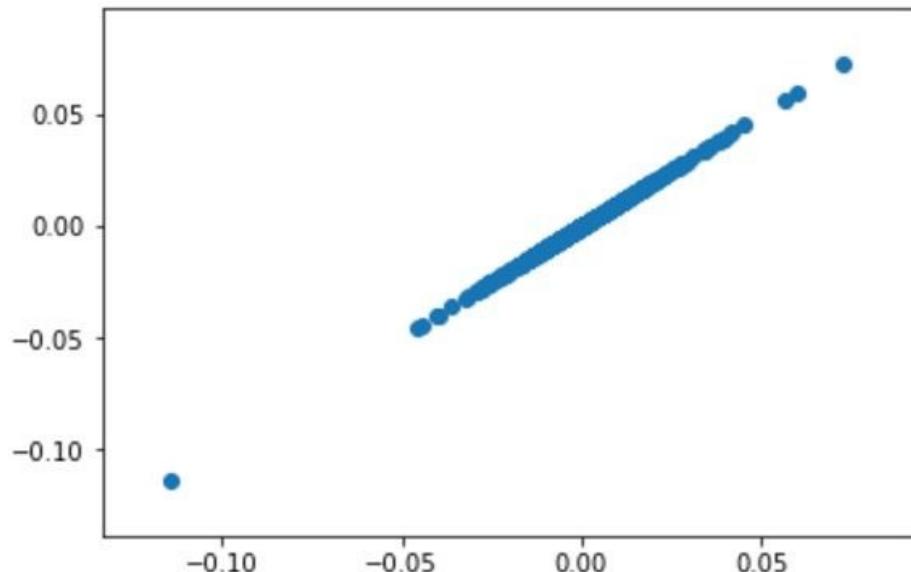


This gives us a very quick view of the overall correlation of the daily returns between the two stocks. Each dot represents a single day for both the stocks. Each dot is plotted along the vertical based on the percentage change for AAPL, and along the horizontal for MSFT.

If for every amount that AAPL changed in value, MSFT also changed an identically proportional amount that day, then all the dots would fall along a perfect vertical diagonal from the lower-left to upper-right section. In this case, the two variables would be perfectly correlated with a correlation value of 1.0. If the two variables were perfectly uncorrelated, the correlation and hence the slope of the line would be 0, which is perfectly horizontal.

To demonstrate what a perfect correlation would look like, we can plot MSFT versus MSFT. Any such series when correlated with itself will always be 1.0:

```
In [28]: # demonstrate perfect correlation  
plt.scatter(daily_pc['MSFT'], daily_pc['MSFT']);
```

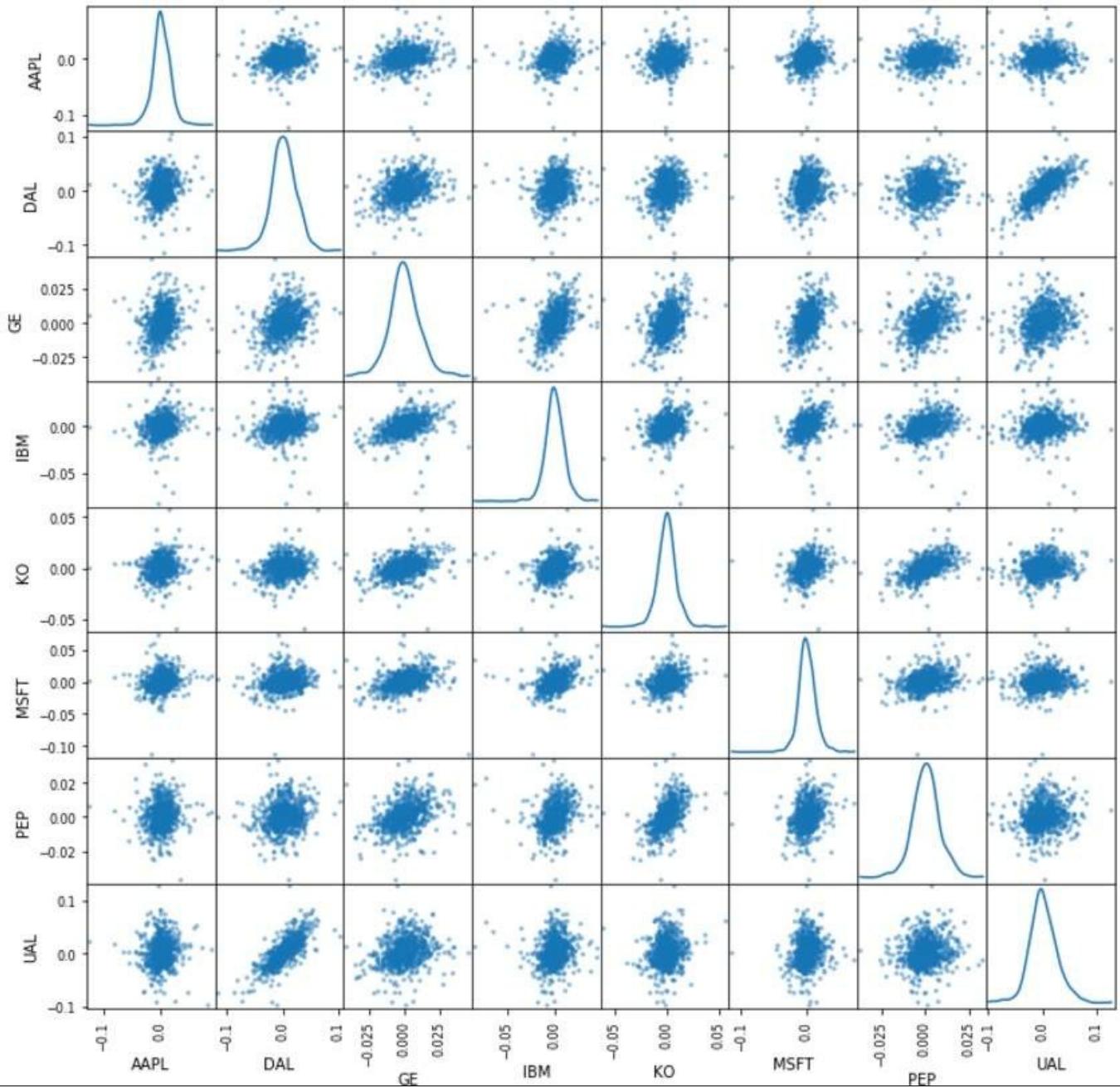


Getting back to the plot of AAPL versus MSFT, excluding several outliers, this cluster appears to demonstrate only a moderate correlation between the two stocks, as the values all appear to cluster around the center.

An actual correlation calculation (which will be examined in the next section) shows the correlation to be 0.1827 (which would be the slope of the regression line). This regression line would be more horizontal than diagonal. This means that for any specific change in the price of AAPL, statistically, it would more times than not, not be able to predict the change in price of MSFT on the given day from the price change in AAPL.

To facilitate the bulk visual analysis of multiple correlations, we can use a scatter plot matrix graph:

```
In [29]: from pandas.plotting import scatter_matrix  
# plot the scatter of daily price changed for ALL stocks  
scatter_matrix(daily_pc, diagonal='kde', figsize=(12,12));
```



The diagonal in this plot is a kernel density estimation graph. The narrower curves are less volatile than those that are wider, with the skew representing the tendency for greater returns or losses. Combined with the scatter plots, this gives a quick summary of the comparison of any two stocks with two different visual metrics.

Correlation of stocks based on the daily percentage change of the closing price

Correlation is a measure of the strength of the association between two variables. A correlation coefficient of 1.0 means that every change in value in one set of data has a proportionate change in value in the other set of data. A 0.0 correlation means that the data sets have no relationship. The higher the correlation, the more ability there is to predict a change in each, based on one or the other.

The scatter plot matrix gave us a quick visual idea of the correlation between two stocks, but it was not an exact number. The exact correlation between the columns of data in `DataFrame` can be calculated using the `.corr()` method. This will produce a matrix of all possible correlations between the variables represented the columns.

The following example calculates the correlation in the daily percentage change in the close price for all these stocks over the three years of the sample:

```
In [30]: # calculate the correlation between all the stocks relative
# to daily percentage change
corrs = daily_pc.corr()
corrs
```

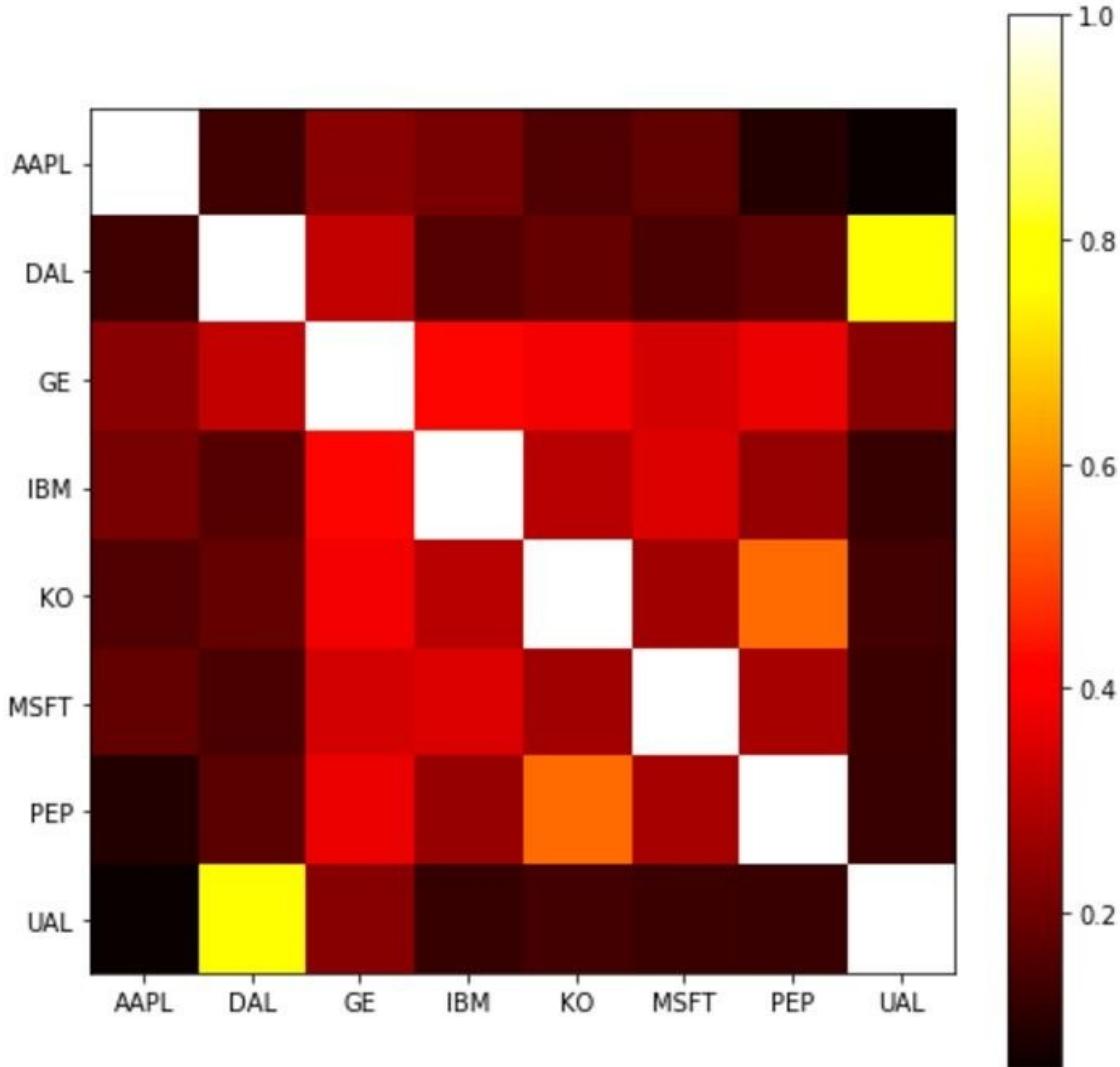
Ticker	AAPL	DAL	GE	...	MSFT	\
AAPL	1.000000	0.136735	0.238862	...	0.182679	
DAL	0.136735	1.000000	0.318175	...	0.152912	
GE	0.238862	0.318175	1.000000	...	0.340657	
IBM	0.213404	0.166197	0.427995	...	0.354523	
KO	0.157938	0.187258	0.386182	...	0.267434	
MSFT	0.182679	0.152912	0.340657	...	1.000000	
PEP	0.096834	0.174259	0.373409	...	0.280881	
UAL	0.061678	0.761239	0.236525	...	0.127748	

Ticker	PEP	UAL
AAPL	0.096834	0.061678
DAL	0.174259	0.761239
GE	0.373409	0.236525
IBM	0.252805	0.122614
KO	0.553575	0.139636
MSFT	0.280881	0.127748
PEP	1.000000	0.124301
UAL	0.124301	1.000000

[8 rows x 8 columns]

The diagonal is always 1.0, as a stock is always perfectly correlated with itself. This correlation matrix can be visualized using a heat map:

```
In [31]: # plot a heatmap of the correlations
plt.imshow(corr, cmap='hot', interpolation='none')
plt.colorbar()
plt.xticks(range(len(corr)), corr.columns)
plt.yticks(range(len(corr)), corr.columns)
plt.gcf().set_size_inches(8,8)
```



The idea with this diagram is that you can see the level of correlation via color, by finding the intersection of vertical and horizontal variables. The darker the color, the less the correlation; the lighter the color, the greater the correlation.

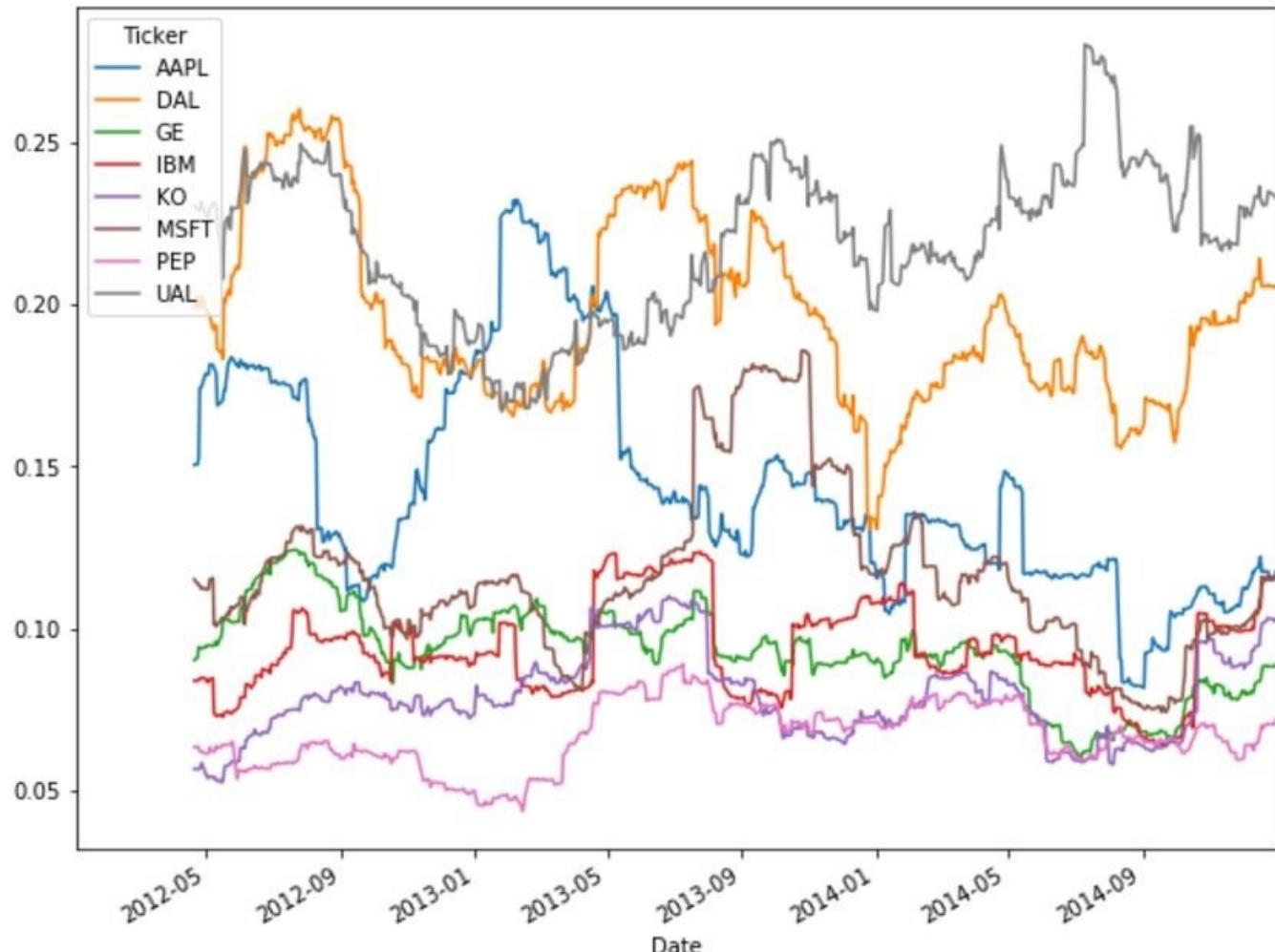
Calculating the volatility of stocks

The volatility of a stock is a measurement of the amount of change of variance in the price of a stock over a specific time-period. It is common to compare the volatility of a stock to another stock to get a feel for which may have less risk, or to a market index to compare the stock's volatility to the overall market. Generally, the higher the volatility, the riskier the investment is in that stock.

Volatility is calculated by taking a rolling-window standard deviation on the percentage change in a stock (and scaling it relative to the size of the window). The size of the window affects the overall result. The wider a window, the less representative the measurement will become. As the window narrows, the result approaches the standard deviation. So, it is a bit of an art to pick the proper window size, based on the data sampling frequency. Fortunately, pandas makes this very easy to modify interactively.

As a demonstration, the following calculates the volatility of the stocks in our sample, given a window of 75 periods:

```
In [32]: # 75 period minimum
min_periods = 75
# calculate the volatility
vol = daily_pc.rolling(window=min_periods).std() * \
      np.sqrt(min_periods)
# plot it
vol.plot(figsize=(10, 8));
```



Lines higher on the chart represent overall higher volatility, and the change of volatility over time is shown as movement in each line.

Determining risk relative to expected returns

A useful analysis is to relate the volatility of a stock's daily percentage change to its expected return. This gives a feel for the risk/return ratio of the investment in the stock. This can be calculated by mapping the mean of the daily percentage change relative to the standard deviation of the same values.

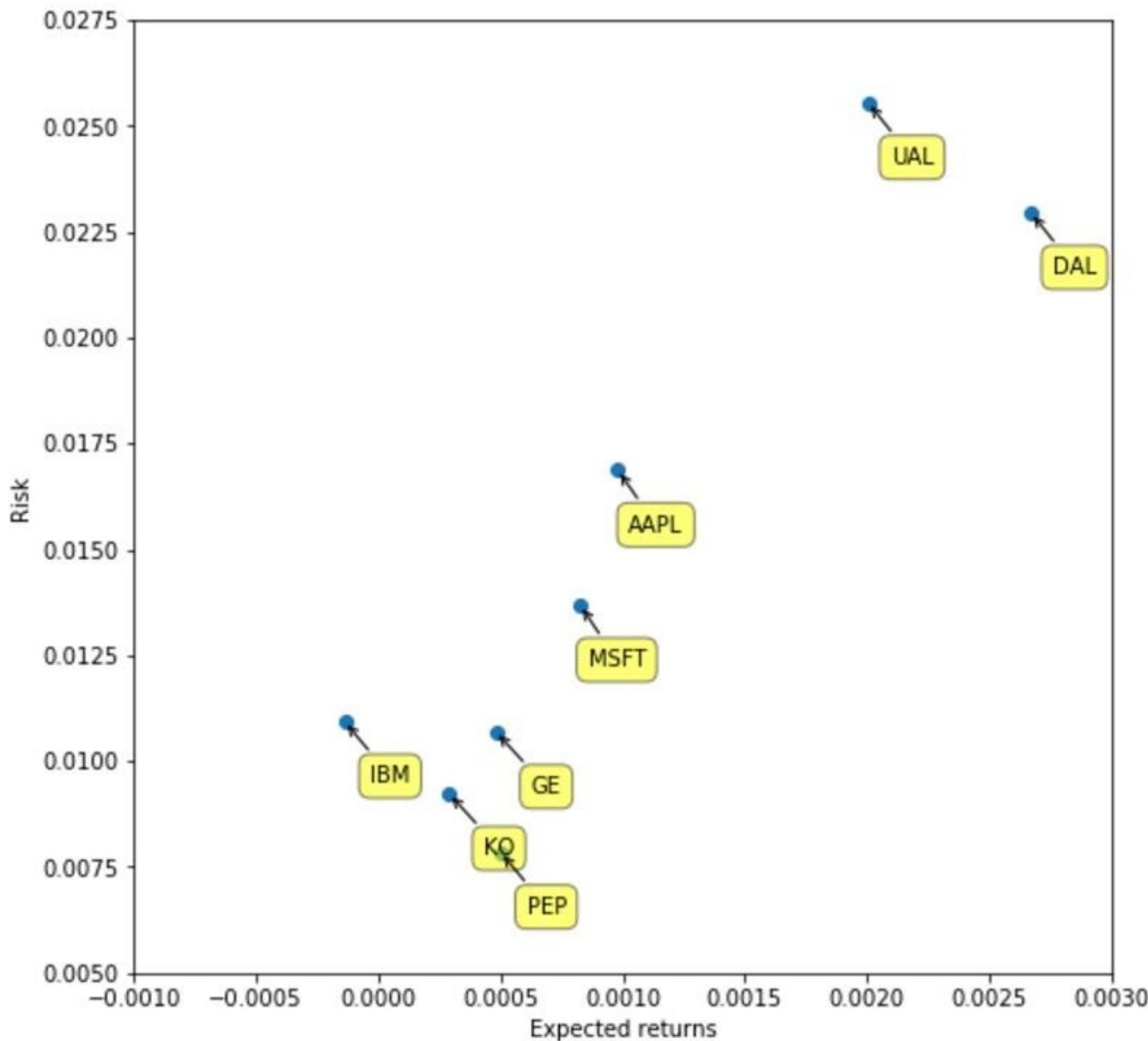
The creates a scatter plot that relates the risk and return of our sample set of stocks, and labels the points with a bubble and arrow too:

```
In [33]: # generate a scatter of the mean vs std of daily % change
plt.scatter(daily_pc.mean(), daily_pc.std())
plt.xlabel('Expected returns')
plt.ylabel('Risk')

# this adds fancy labels to each dot, with an arrow too
for label, x, y in zip(daily_pc.columns,
                       daily_pc.mean(),
                       daily_pc.std()):
    plt.annotate(
        label,
        xy = (x, y), xytext = (30, -30),
        textcoords = 'offset points', ha = 'right',
        va = 'bottom',
        bbox = dict(boxstyle = 'round,pad=0.5',
                    fc = 'yellow',
                    alpha = 0.5),
        arrowprops = dict(arrowstyle = '->',
                         connectionstyle = 'arc3,rad=0'))

# set ranges and scales for good presentation
plt.xlim(-0.001, 0.003)
plt.ylim(0.005, 0.0275)

# set size
plt.gcf().set_size_inches(8,8)
```



The results of this immediately jump out from the visualization, but have been more difficult to see by just looking at tables of numbers:

- Airline stocks (AA, DAL, and UAL) have the highest risk but also have the highest returns (Isn't that the general rule of investing?).
- The tech stocks are of medium risk but also have medium returns.
- Among the tech stocks, IBM and GE are the most conservative of the four.
- The cola stocks have the lowest risk but are also among the lowest returns as a group. This makes sense for a high-volume commodity.

Summary

We have reached the end of our journey in learning about pandas and the features it offers for data manipulation and analysis. Prior to this chapter, we spent most of our time learning the features of pandas, and in many cases, using data designed to demonstrate the concepts instead of using real-world data.

In this chapter, we used everything that we've learned up to this point to demonstrate how easy it is to use pandas to analyze real-world data, specifically stock data, and to derive results from the data. Often, this enables us to draw quick conclusions through visualizations designed to make the patterns in the data apparent.

This chapter also introduced several financial concepts, such as the daily percentage change, calculating returns, and the correlation of time-series data, among others. The focus was not on financial theory but to demonstrate how easy it is to use pandas to manage and derive meaning from what was otherwise just lists and lists of numbers.

In closing, it is worth noting that although pandas was created by financial analysts (hence its ability to provide simple solutions in the financial domain) pandas is in no way limited to just finance. It is a very robust tool for data analysis and can be applied just as effectively to many other domains. Several of these are emerging markets that represent a significant opportunity, such as social network analysis or applications of wearable computing. Whatever your domain of use for pandas, I hope you find using pandas as fascinating as I do.