Community Experience Distilled

Mastering pandas for Finance

Master pandas, an open source Python Data Analysis Library, for financial data analysis

Michael Heydt

[PACKT] PUBLISHING open source* community experience distilled

Mastering pandas for Finance

Master pandas, an open source Python Data Analysis Library, for financial data analysis

Michael Heydt

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Mastering pandas for Finance

First published: May 2015

Production reference: 1190515

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

ISBN 978-1-78398-510-4

www.packtpub.com

Credits

Author
Michael Heydt

Reviewers
James Beveridge
Philipp Deutsch
Jon Gaither
Jim Holmström
Francesco Pochetti

Commissioning Editor
Kartikey Pandey

Content Development Editor
Merwyn D'souza

Technical Editor
Shashank Desai

Copy Editor
Sarang Chari

Project Coordinator
Neha Bhatnagar

Proofreaders
Stephen Copestake
Safis Editing

Indexer
Mariammal Chettiyar

Graphics
Sheetal Aute
Disha Haria

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Author

Michael Heydt is an independent consultant, educator, and trainer with nearly 30 years of professional software development experience, during which time, he focused on Agile software design and implementation using advanced technologies in multiple verticals, including media, finance, energy, and healthcare. He holds an MS degree in mathematics and computer science from Drexel University and an executive master's of technology management degree from the University of Pennsylvania's School of Engineering and Wharton Business School. His studies and research have focused on technology management, software engineering, entrepreneurship, information retrieval, data sciences, and computational finance. Since 2005, he has specialized in building energy and financial trading systems for major investment banks on Wall Street and for several global energy-trading companies, utilizing .NET, C#, WPF, TPL, DataFlow, Python, R, Mono, iOS, and Android. His current interests include creating seamless applications using desktop, mobile, and wearable technologies, which utilize high-concurrency, high-availability, and real-time data analytics; augmented and virtual reality; cloud services; messaging; computer vision; natural user interfaces; and software-defined networks. He is the author of numerous technology articles, papers, and books. He is a frequent speaker at .NET user groups and various mobile and cloud conferences, and he regularly delivers webinars and conducts training courses on emerging and advanced technologies. To know more about Michael, visit his website at `http://bseamless.com/`.

About the Reviewers

James Beveridge is a product analyst and machine learning specialist. He earned his BS degree in mathematics from Cal Poly, San Luis Obispo, CA. He has worked with the finance and analytics teams in technology and marketing companies in the Bay Area, Chicago, and New York. His current work focuses on segmentation and classification modeling, statistics, and product development. He has enjoyed contributing to this book as a technical reviewer.

Philipp Deutsch obtained degrees in mathematics and physics from the University of Vienna and the Vienna University of Technology before starting a career in financial services and consulting. He has worked on a number of projects involving data analytics across Europe, both in the banking and consumer/retail sectors, and has extensive experience in Python, R, and SQL. He currently lives in London.

Jon Gaither is a senior information systems student at Clemson University with a background in finance. He started learning Python during his sophomore year of college. Since then, he has dabbled in frameworks such as Flask, Django, and pandas purely out of interest. Outside of Python, Jon has studied Java, SAS, VBA, and SQL. His professional experience comes from internships in financial services and satellite communications.

Jim Holmström is soon to graduate with a bachelor's degree in engineering physics and a master's degree in machine learning from KTH Royal Institute of Technology, Stockholm.

He is currently a developer and partner at Watty—an electricity data analysis start-up that creates a breakdown of a household's energy spending from the total electricity consumption data. Watty's leading-edge technology stack has pandas as an integral part.

Both professionally and in his free time, he enjoys data analysis, functional programming, and well-structured code.

For more information, visit `http://portfolio.jim.pm`.

Francesco Pochetti graduated in physical chemistry in Rome in 2012 and was employed at Avio in Italy. He worked there for 2 years as a solid rocket propellant specialist, taking care of the formulation and development of rocket fuels for both military and aerospace purposes. In July 2014, he moved to Berlin to attend Data Science Retreat—a 3-month boot camp in data analysis and machine learning in Python and R. After this short German experience, he ended up at Amazon in Luxembourg, where he currently works as a business analyst for Kindle content.

In his spare time, he likes to read and play around with several programming languages, Python being among his preferred ones. You can follow him and his data-related projects at `http://francescopochetti.com/`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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

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

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

Why subscribe?

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

Free access for Packt account holders

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

Table of Contents

Preface

Mastering pandas for Finance will teach you how to use Python and pandas to model and solve real-world financial problems using pandas, Python, and several open source tools that assist in various financial tasks, such as option pricing and algorithmic trading.

This book brings together various diverse concepts related to finance in an attempt to provide a unified reference to discover and learn several important concepts in finance and explains how to implement them using a core of Python and pandas that provides a unified experience across the different models and tools.

You will start by learning about the facilities provided by pandas to model financial information, specifically time-series data, and to use its built-in capabilities to manipulate time-series data, group and derive aggregate results, and calculate common financial measurements, such as percentage changes, correlation of time-series, various moving window operations, and key data visualizations for finance.

After establishing a strong foundation from which to use pandas to model financial time-series data, the book turns its attention to using pandas as a tool to model the data that is required as a base for performing other financial calculations. The book will cover diverse areas in which pandas can assist, including the correlations of Google trends with stock movements, creating algorithmic trading systems, and calculating options payoffs, prices, and behaviors. The book also shows how to model portfolios and their risk and to optimize them for specific risk/return tolerances.

What this book covers

Chapter 1, *Getting Started with pandas Using Wakari.io*, walks you through using Wakari.io, an online collaborative data analytics platform, that utilizes Python, IPython Notebook, and pandas. We will start with a brief overview of Wakari.io and step through how to upgrade the default Python environment and install all of the tools used throughout this text. At the end, you will have a fully functional financial analytics platform supporting all of the examples we will cover.

Chapter 2, *Introducing the Series and DataFrame*, teaches you about the core pandas data structures—the Series and the DataFrame. You will learn how a Series expands on the functionality of the NumPy array to provide much richer representation and manipulation of sequences of data through the use of high-performance indices. You will then learn about the pandas DataFrame and how to use it to model two-dimensional tabular data.

Chapter 3, *Reshaping, Reorganizing, and Aggregating*, focuses on how to use pandas to group data, enabling you to perform aggregate operations on grouped data to assist with deriving analytic results. You will learn to reorganize, group, and aggregate stock data and to use grouped data to calculate simple risk measurements.

Chapter 4, *Time-series*, explains how to use pandas to represent sequences of pricing data that are indexed by the progression of time. You will learn how pandas represents date and time as well as concepts such as periods, frequencies, time zones, and calendars. The focus then shifts to learning how to model time-series data with pandas and to perform various operations such as shifting, lagging, resampling, and moving window operations.

Chapter 5, *Time-series Stock Data*, leads you through retrieving and performing various financial calculations using historical stock quotes obtained from Yahoo! Finance. You will learn to retrieve quotes, perform various calculations, such as percentage changes, cumulative returns, moving averages, and volatility, and finish with demonstrations of several analysis techniques including return distribution, correlation, and least squares analysis.

Chapter 6, *Trading Using Google Trends*, demonstrates how to form correlations between index data and trends in searches on Google. You will learn how to gather index data from Quandl along with trend data from Google and then how to correlate this time-series data and use that information to generate trade signals, which will be used to calculate the effectiveness of the trading strategy as compared to the actual market performance.

Chapter 7, *Algorithmic Trading*, introduces you to the concepts of algorithmic trading through demonstrations of several trading strategies, including simple moving averages, exponentially weighted averages, crossovers, and pairs-trading. You will then learn to implement these strategies with pandas data structures and to use Zipline, an open source back-testing tool, to simulate trading behavior on historical data.

Chapter 8, *Working with Options*, teaches you to model and evaluate options. You will first learn briefly about options, how they function, and how to calculate their payoffs. You will then load options data from Yahoo! Finance into pandas data structures and examine various options attributes, such as implied volatility and volatility smiles and smirks. We then examine the pricing of options with Black-Scholes using Mibian and finish with an overview of Greeks and how to calculate them using Mibian.

Chapter 9, *Portfolios and Risk*, will teach you how to model portfolios of multiple stocks using pandas. You will learn about the concepts of Modern Portfolio Theory and how to apply those theories with pandas and Python to calculate the risk and returns of a portfolio, assign different weights to different instruments in a portfolio, derive the Sharpe ratio, calculate efficient frontiers and value at risk, and optimize portfolio instrument allocation.

What you need for this book

This book assumes that you have some familiarity with programming concepts, but even those without programming, 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 is provided in *Chapter 1*, *Getting Started with pandas Using Wakari.io*, on installing both on Windows, OS X, and Ubuntu systems. For those interested in not installing any software, instructions are also given on using the Wakari.io online Python data analysis service. Data is either provided with the text or is available for download via pandas from data services such as Yahoo! Finance. We will also use several open source software packages such as Zipline and Mibian, the retrieval, installation, and usage of which will be explained during the appropriate chapters.

Who this book is for

If you are interested in quantitative finance, financial modeling, trading, or simply want to learn Python and pandas as applied to finance, then this book is for you. Some knowledge of Python and pandas is assumed, but the book will spend time explaining all of the necessary pandas concepts that are required within the context of application to finance. Interest in financial concepts is helpful, but no prior knowledge is expected.

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/IPython input or output is written as follows:

```
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
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Once dropped, click on the **Upload Files** button and you will see the following files in your **Wakari** directory."

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through 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 on `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 all Packt books you have purchased 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. The code examples in the book are also publicly available on Wakari.io at `https://wakari.io/sharing/bundle/Pandas4Finance/MasteringPandas4Finance_Index`.

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 copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with pandas Using Wakari.io

In *Mastering pandas for Finance*, we will examine the use of pandas to manage financial data and perform various financial analyses with a specific focus on financial processes that can be facilitated using the capabilities provided within pandas, along with an occasional quantitative financial technique. I have made an assumption that you have basic knowledge of Python programming and have used IPython and IPython Notebooks. Knowledge of pandas is preferred, but we will cover enough information on pandas for any reader to be able to understand the technique being used. We will occasionally and briefly touch upon areas of quantitative finance, but those times will be mostly for information purposes and will have implementations that are provided in the code of the text.

During this voyage of discovery, we will begin with an overview/review of concepts and data structures in pandas that are of importance to financial analysis. We will then move into various concepts, techniques, tools, and examples of specific financial analysis problems as solved with Python, pandas, and several other Python libraries and tools, including **Wakari**, **matplotlib**, **SciPy**, **Quandl**, **Zipline**, and **Mibian**. These will be varied in nature, and topics ranging from analysis of historical stock data, correlating search data with trends in stock prices, algorithmic trading and backtesting, options modeling and pricing, and portfolio and risk analysis will be covered.

In this first chapter, we will walk through creating an account and environment in Wakari.io and installing the code samples into that environment. I have chosen Wakari.io as a basis for a pandas-based financial environment because it is relatively painless to get up and running with all of the tools we will utilize, and also the samples provided in the code bundle of this book are in the IPython Notebook format, which is simple to use within Wakari.io.

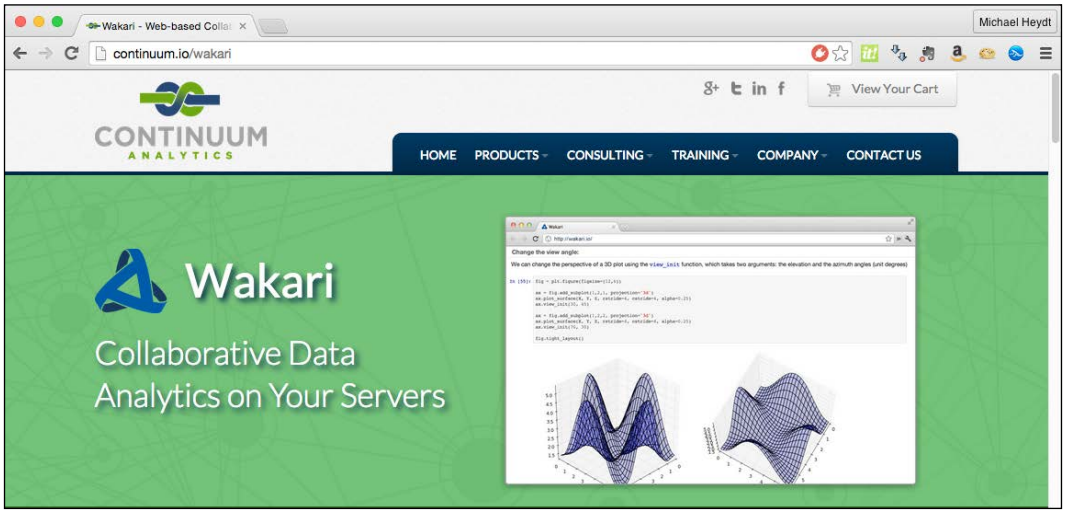
The use of Wakari, however, does not prevent you from using your own Python environment. The examples in the text will run in any Python environment and were originally built using the Anaconda and IPython Notebook formats with all of the mentioned tools installed within the environment. Just in case you don't want to use Wakari, all the code examples in the text are presented as IPython and will run in a properly configured IPython environment.

So, let's get started. In this chapter, we will cover the following topics:

- What is Wakari.io?
- Creating a Wakari account
- Updating the default Wakari environment to run all our examples
- Installing and running the code samples in Wakari

What is Wakari?

Wakari (`http://continuum.io/wakari`) is a collaborative data analytics platform that allows you to explore data and create analytic scripts in collaboration with IPython Notebooks. It is an offering of Continuum Analytics, the creators of the Anaconda Python distribution, which is generally considered to be one of the best Python distributions. Wakari is offered as a solution that you can run in your enterprise at an expense, or as a web- or cloud-based solution offered on a freemium basis. The following screenshot shows Wakari as an offering of Continuum Analytics:



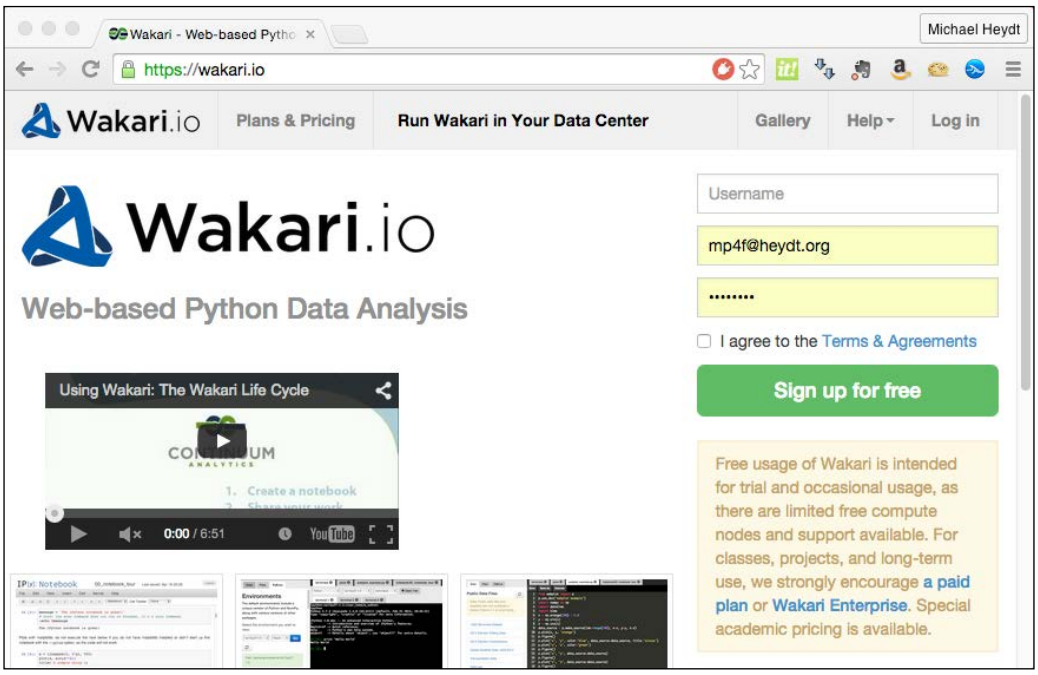
The approach in this text will be to guide you in using the cloud-based Wakari solution. This environment provides an effective quick start to learning pandas and performing all the data analysis in this text but with very minimal effort in managing a local Python installation.

Creating a Wakari cloud account

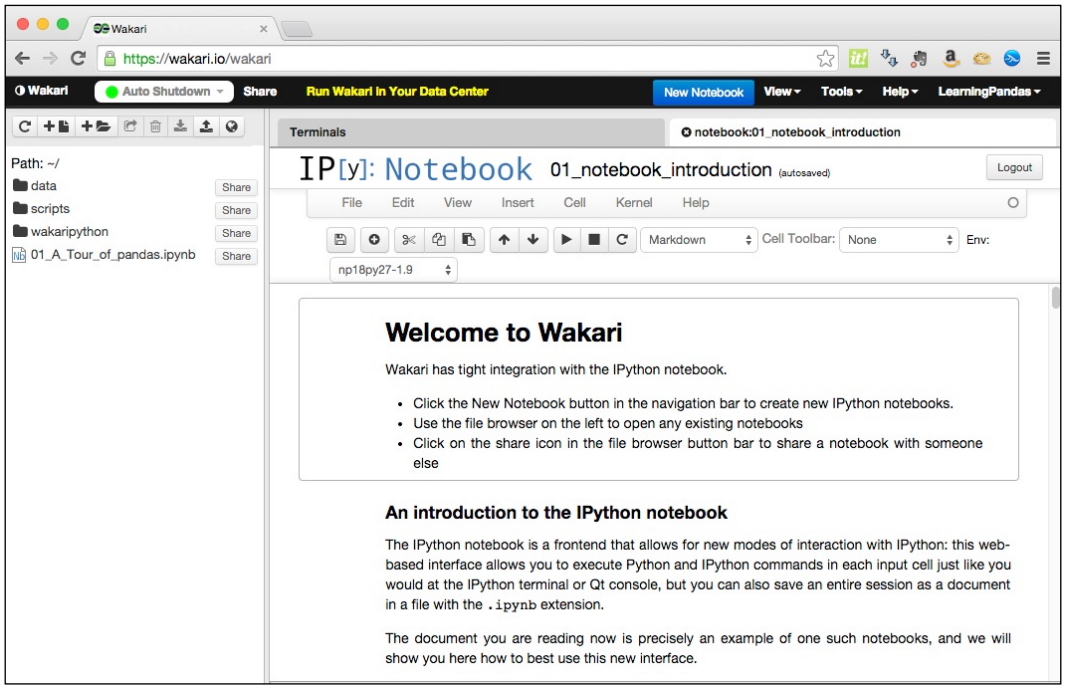
The cloud-based offering for Wakari is available at `https://wakari.io`. For convenience, from this point on, I will refer to Wakari.io as Wakari, but always know that I am referring to the cloud-based solution.

Wakari is a freemium service that allows you to run web-based Python distributions. Specifics on the free part of the freemium services can be found on the site, but all of the examples in this text can be run for free in the Wakari environment (at least at the time of writing this book). Wakari offers very low resistance to success in learning all of the concepts in this text as well as many others.

The guidance in this chapter will take you through creating and setting up an online Python environment, which can run all of the examples in this book. To start, open your browser and enter `https://wakari.io` in the address bar. This will display the following page:



Sign up for a new account, and upon successful registration for the service, you will be presented with the following web interface to manage IPython Notebooks:



IPython Notebooks are a default feature in Wakari for the purpose of developing Python applications. All the examples in this book were developed as IPython Notebooks, although the code can be run sequentially in IPython or even Python. An advantage of IPython Notebooks is the ability to intermix markdown with Python code within a semi-dynamic web page, which allows easy reuse of code, and perhaps more importantly, publishing of code on the Web.

As a matter of fact, you can find all the code files for this book on Wakari at `https://wakari.io/sharing/bundle/Pandas4Finance/MasteringPandas4Finance_Index`.

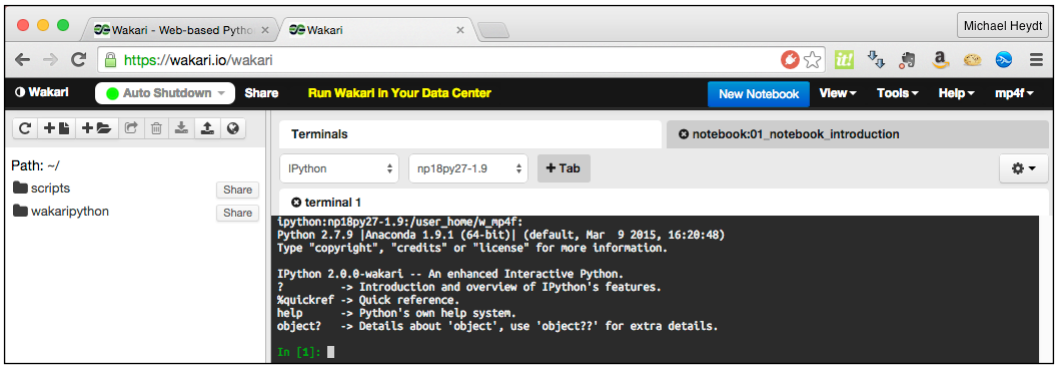
At the time of writing this book, the default Python environment provided by Wakari is Python 2.7.9, and more specifically, Anaconda 1.9.1 (all version numbers are at the time of writing, so when you read this, they may be newer). This is, in general, a good environment for what we want to accomplish in this book, although a few packages need updating and several others need to be installed. In Wakari, pandas is currently at 0.16.0, which is satisfactory for our needs.

The specific packages that either need updating or installing are as follows:

- matplotlib
- Zipline
- Quandl
- html5lib
- Mibian
- tzlocal

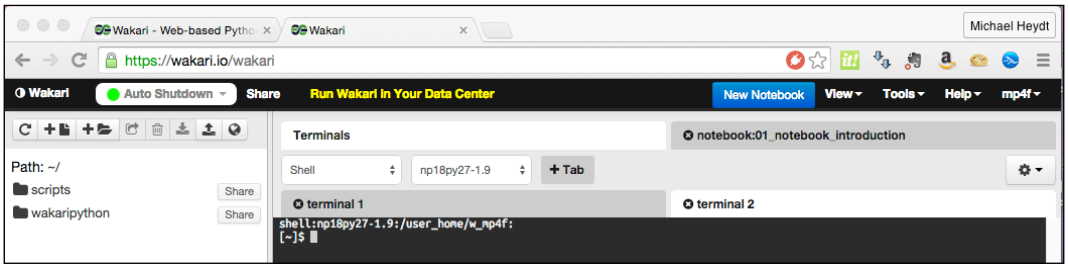
We will go over each of these briefly and also see how to install/update each. In general, the update/install process will be performed using a shell within Wakari. One of the spectacular features of Wakari includes running both interactive IPython sessions and operating system shells directly in the browser.

From a new environment within Wakari, you can open terminals using the **Terminals** tab. Click on the **Terminals** tab, and you will see the following screenshot, which represents a default IPython shell for your account (currently referred to as `np18py27-19`):



You can perform any Python programming within this web-based interface, including all of the examples in this book. However, the default Wakari environment needs a few updates and first-time installs to run all of the examples in the text.

We can perform updates to the environment by opening a shell. This can be performed by selecting **Shell** from the drop-down menu, along with **np18py27-1.9**, and pressing the **+Tab** button. After that, you will be presented with the following screenshot:



We are now in an OS shell that provides you with many options, including updating your Python environment, which we will now perform.

Updating existing packages

We need to update one package in the default Wakari environment—matplotlib. This is the graphics package we will use at various points in this book. For most of the purposes, the version in Wakari (1.3.1) is satisfactory, but the candlestick charts that we will create require an update to matplotlib from 1.3.1 to a higher version. This is performed with the `conda` package manager using the `conda update matplotlib` command. When issuing this, you will see something similar to the following in the terminal tab in your web browser:

```
[~]$ conda update matplotlib
Fetching package metadata: ........
Solving package specifications: .
 Package plan for installation in environment /opt/anaconda/envs/np18py27-1.9:

The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    qt-4.8.6                   |                1        36.4 MB  http://repo.continuum.io/pkgs/free/linux-64/
    setuptools-15.1            |          py27_1         435 KB  http://repo.continuum.io/pkgs/free/linux-64/
    pyqt-4.11.3                |          py27_1         3.5 MB  http://repo.continuum.io/pkgs/free/linux-64/
    ------------------------------------------------------------
                                           Total:        40.4 MB

The following NEW packages will be INSTALLED:

    pyqt:       4.11.3-py27_1    http://repo.continuum.io/pkgs/free/linux-64/
    sip:        4.16.5-py27_0    http://repo.continuum.io/pkgs/free/linux-64/

The following packages will be UPDATED:

    matplotlib: 1.3.1-np18py27_0 http://repo.continuum.io/pkgs/free/linux-64/ --> 1.4.3-np19py27_1
http://repo.continuum.io/pkgs/free/linux-64/
    py2cairo:   1.10.0-py27_1    http://repo.continuum.io/pkgs/free/linux-64/ --> 1.10.0-py27_2
http://repo.continuum.io/pkgs/free/linux-64/
    pyparsing:  2.0.1-py27_0     http://repo.continuum.io/pkgs/free/linux-64/ --> 2.0.3-py27_0
http://repo.continuum.io/pkgs/free/linux-64/
    qt:         4.8.5-0          http://repo.continuum.io/pkgs/free/linux-64/ --> 4.8.6-1
http://repo.continuum.io/pkgs/free/linux-64/
    setuptools: 15.0-py27_0      http://repo.continuum.io/pkgs/free/linux-64/ --> 15.1-py27_1
http://repo.continuum.io/pkgs/free/linux-64/

Proceed ([y]/n)? y

Fetching packages ...
qt-4.8.6-1.tar 100% |################################################################| Time: 0:00:08   4.47 MB/s
setuptools-15. 100% |################################################################| Time: 0:00:00 725.92 kB/s
pyqt-4.11.3-py 100% |################################################################| Time: 0:00:01   2.73 MB/s
Extracting packages ...
[      COMPLETE      ]|#############################################################################| 100%
Unlinking packages ...
[      COMPLETE      ]|#############################################################################| 100%
Linking packages ...
[      COMPLETE      ]|#############################################################################| 100%
[~]$
```

Installing new packages

The remainder of the packages need to be installed. All these package installations follow the same process, although there are slightly different commands, which alternate between using `pip` and the `conda` package manager for installation.

For time zone operations, tzlocal is used and is updated using `pip`. The installation is performed as shown here:

```
[~]$ pip install tzlocal
Collecting tzlocal
  Downloading tzlocal-1.1.3.tar.gz
Requirement already satisfied (use --upgrade to upgrade): pytz in /opt/anaconda/envs/np18py27-1.9/lib/python2.7/site-
packages (from tzlocal)
Installing collected packages: tzlocal
  Running setup.py install for tzlocal
Successfully installed tzlocal-1.1.3
[~]$
```

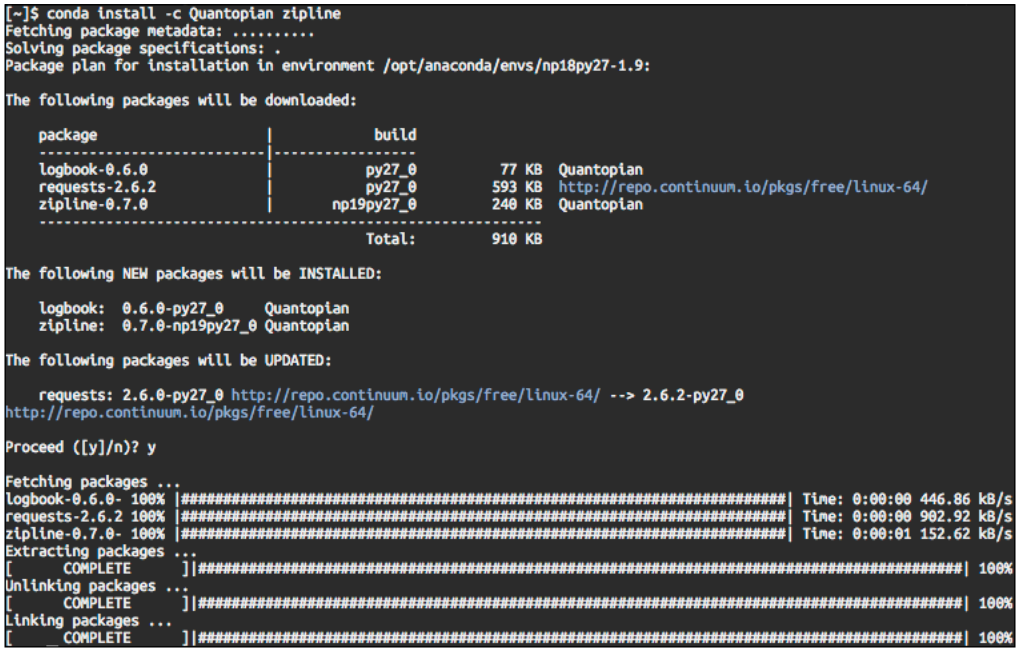
The samples do not use html5lib directly, but other libraries do use it indirectly. We will use these libraries to read and parse data. We need to update this using `conda`, as shown here:

```
[~]$ conda install html5lib
Fetching package metadata: ........
Solving package specifications: .
Package plan for installation in environment /opt/anaconda/envs/np18py27-1.9:

The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    html5lib-0.999             |           py27_0         172 KB  http://repo.continuum.io/pkgs/free/linux-64/

The following NEW packages will be INSTALLED:

    html5lib: 0.999-py27_0 http://repo.continuum.io/pkgs/free/linux-64/

Proceed ([y]/n)? y

Fetching packages ...
html5lib-0.999 100% |###############################################################| Time: 0:00:00 531.84 kB/s
Extracting packages ...
[      COMPLETE      ]|#################################################################################################| 100%
Linking packages ...
[      COMPLETE      ]|#################################################################################################| 100%
```

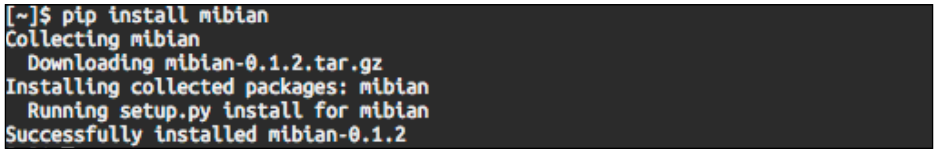
A library provided at `https://www.quandl.com/`, Quandl is a provider of data that you can integrate into your applications via download or the API. The Python API that we will use to access S&P 500 data is free and can be installed using `conda`, as shown here:

```
[~]$ conda install quandl
Fetching package metadata: ........
Solving package specifications: .
Package plan for installation in environment /opt/anaconda/envs/np18py27-1.9:

The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    quandl-2.8.5               |       np19py27_0          11 KB  http://repo.continuum.io/pkgs/free/linux-64/

The following NEW packages will be INSTALLED:

    quandl: 2.8.5-np19py27_0 http://repo.continuum.io/pkgs/free/linux-64/

Proceed ([y]/n)? y

Fetching packages ...
quandl-2.8.5-n 100% |###############################################################| Time: 0:00:00 929.01 kB/s
Extracting packages ...
[      COMPLETE      ]|#################################################################################################| 100%
Linking packages ...
[      COMPLETE      ]|#################################################################################################| 100%
```

Available at `https://www.quantopian.com/`, Zipline is a backtesting/trading simulator that we will use. Quantopian is a website that focuses on algorithmic trading, and it produces Zipline, which it uses as one of its underlying technologies. Although installed using `conda`, Zipline requires the use of a different channel. Notice the slight variation in the use of `conda` to specify the Quantopian channel in the following screenshot:



The final package we need to install is Mibian, a small library that computes Black-Scholes and its derivatives. This is installed using `pip`, as shown here:



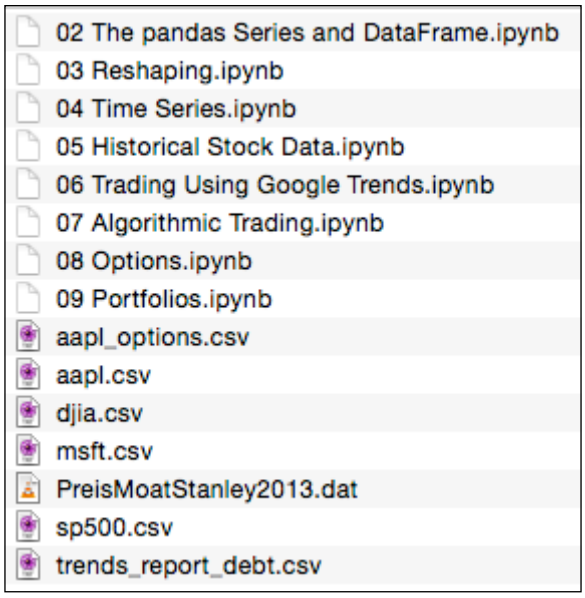
We are now ready to run any of the sample Notebooks.

Downloading the example code

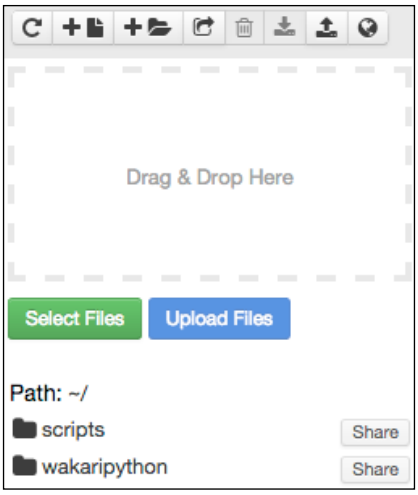
You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Installing the samples in Wakari

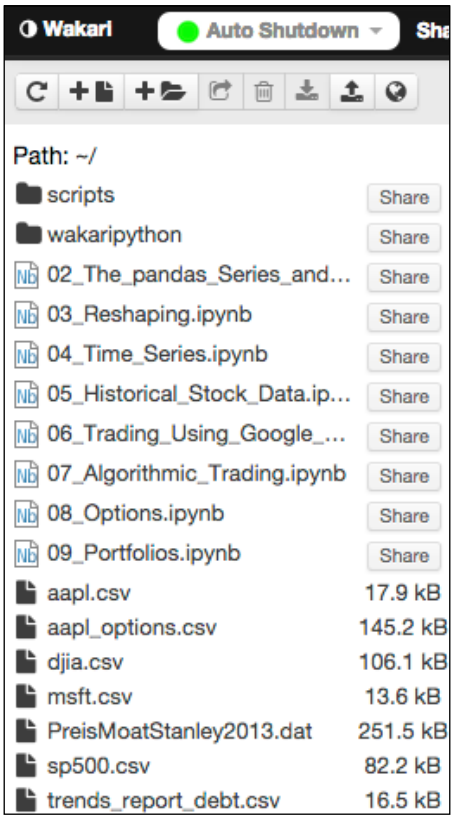
To install the examples in Wakari, download the code bundle and unzip the files to a local directory. You will see a set of files as shown here:



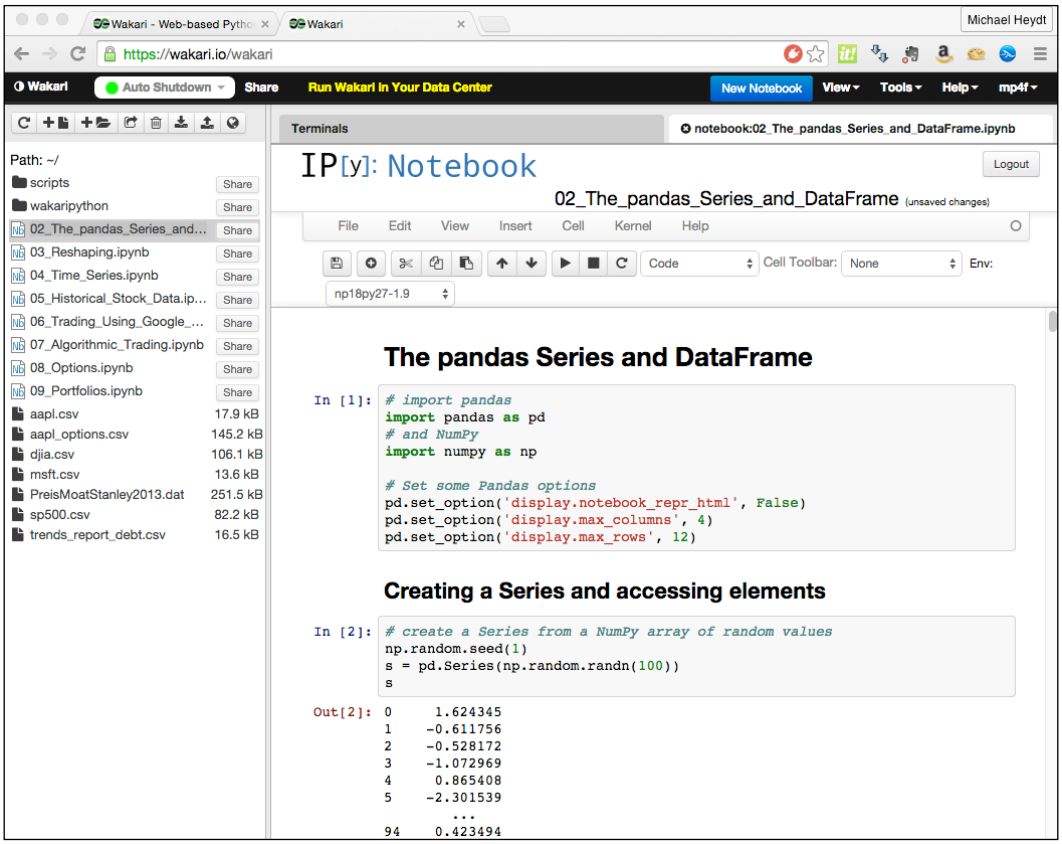
To upload the files to Wakari, click on the upload files icon and drag the files into the **Drag & Drop Here** section of the web page:



Once dropped, click on the **Upload Files** button, and you will see the following files in your **Wakari** directory:



At this point, you should be able to open and run any of the Notebooks and even examine the data in the browser. As an example, the following screenshot demonstrates the Notebook for *Chapter 2, Introducing the Series and DataFrame*, opened in Wakari:



Summary

This chapter was a brief introduction to this book. You learned how to set up a Python environment in Wakari.io to be able to run the code samples provided throughout the text. This included instructions on how to update the default Wakari.io Python environment to support the required packages that are required for all of the examples in the remainder of the text.

In the next chapter, we will dive into using pandas and its core data structures, `Series` and `DataFrame`. These will be core to representing data in later chapters, where we primarily use pandas `DataFrame` objects to represent financial data, which we apply to various financial analyses.

2
Introducing the Series and DataFrame

pandas provides a comprehensive set of data structures for working with and manipulating data and performing various statistical and financial analyses. The two primary data structures in pandas are `Series` and `DataFrame`. In this chapter, we will examine the `Series` object and how it extends a NumPy `ndarray` to provide operations such as indexed data retrieval, axis labeling, and automatic alignment. Then, we will move on to examine how `DataFrame` extends the capabilities of `Series` to use columnar/tabular data, which can be of more than one data type.

The intention of this chapter is to be not only a refresher for those with basic familiarity with pandas, but also a means by which someone who is not initiated with pandas can gain enough familiarity with the two data structures and have a good foundation as we move into more finance-related subjects in later chapters. We will not cover all the details of using `Series` and `DataFrame` but will focus on core functionality related to what will be used later in this book for financial analysis. For extensive coverage of `Series` and `DataFrame`, I recommend the companion book, *Learning pandas*, *Packt Publishing*, which goes into both in extensive detail.

Specifically, this chapter will cover the following topics:

- An overview of the `Series` and `DataFrame` objects
- Creating and accessing elements of a `Series`
- Determining the shapes and counts of items in a `Series`
- Alignment of items in a `Series` via index labels
- Creating a `DataFrame`
- Loading example financial data to demonstrate the `DataFrame`

- Selecting rows of a `DataFrame` through several concepts using its index
- Boolean selection of rows of a `DataFrame` using logical expressions
- Performing arithmetic on a `DataFrame`
- Reindexing the `Series` and `DataFrame` objects

Notebook setup

To utilize the examples in this chapter, we will need to include the following imports and settings in either your IPython or IPython Notebook environment:

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

   pd.set_option('display.notebook_repr_html', False)
   pd.set_option('display.max_columns', 8)
   pd.set_option('display.max_rows', 8)
```

The main pandas data structures – Series and DataFrame

Several classes for manipulating data are provided by pandas. Of those, we are interested in `Series` and more interested in `DataFrame`.

The Series

The `Series` is the primary building block of pandas and represents a one-dimensional labeled array based on the NumPy `ndarray`. The `Series` extends the functionality of the NumPy `ndarray` by adding an associated set of labels that are used to index the elements of the array. A `Series` can hold zero or more instances of any single data type.

This labeled index adds significant power to access the elements of the `Series` over a NumPy array. Instead of simply accessing elements by position, a `Series` allows access to items through the associated index labels. The index also assists in a feature of pandas referred to as alignment, where operations between two Series are applied to values with identical labels.

The DataFrame

The `Series` is the basis for data representation and manipulation in pandas, but since it can only associate a single value with any given index label, it ends up having limited ability to model multiple variables of data at each index label. The pandas `DataFrame` solves this by providing the ability to seamlessly manage multiple `Series`, where each of the `Series` represents a column of the `DataFrame` and also by automatically aligning values in each column along the index labels of the `DataFrame`.

In a sense, a `DataFrame` can be thought of as a dictionary-like container of one or more `Series` objects, as a spreadsheet, or probably the best description for those new to pandas is to compare a `DataFrame` to a relational database table. But even that comparison is limiting, as a `DataFrame` has very distinct qualities (such as automatic alignment of `Series` data by index labels) that make it much more capable of exploratory data analysis than either a spreadsheet or a relational database table.

A good way to think about a `DataFrame` is that it unifies two or more `Series` into a single data structure. Each `Series` then represents a named column of the `DataFrame`, and instead of each column having its own index, the `DataFrame` provides a single index and the data in all columns is aligned to the master index of the `DataFrame`. Each index label then references a slice of data across all of the `Series` at the label, forming what is essentially a record of information associated with that particular index label.

A `DataFrame` also introduces the concept of an axis, which you will often see in the pandas documentation and in many of its methods. A `DataFrame` has two axes, horizontal and vertical. Functions from pandas can then be applied to either axis, in essence, stating that it applies either to all the values in selected rows or to all the items in specific columns.

The basics of the Series and DataFrame objects

Now let's examine using the `Series` and `DataFrame` objects, building up an understanding of their capabilities that will assist us in working with financial data.

Creating a Series and accessing elements

A `Series` can be created by passing a scalar value, a NumPy array, or a Python dictionary/list to the constructor of the `Series` object. The following command creates a `Series` from `100` normally distributed random numbers:

```
In [2]:
   np.random.seed(1)
   s = pd.Series(np.random.randn(100))
   s


Out[2]:
   0     1.624345
   1    -0.611756
   2    -0.528172
   3    -1.072969
           ...
   96   -0.343854
   97    0.043597
   98   -0.620001
   99    0.698032
   Length: 100, dtype: float64
```

Individual elements of a `Series` can be retrieved using the `[]` operator of the `Series` object. The item with the index label `2` can be retrieved using the following code:

```
In [3]:
   s[2]


Out[3]:
   -0.528171752263
```

Multiple values can be retrieved using an array of label values, as shown here:

```
In [4]:
   s[[2, 5, 20]]


Out[4]:
   2    -0.528172
   5    -2.301539
```

```
20   -1.100619
dtype: float64
```

A `Series` supports slicing using the `:` slice notation. The following command retrieves the elements of the `Series` where labels are greater than 3 but less than 8 (the end value is not inclusive in pandas slicing, which is a slight difference from NumPy arrays):

```
In [5]:
   s[3:8]


Out[5]:
   3   -1.072969
   4    0.865408
   5   -2.301539
   6    1.744812
   7   -0.761207
   dtype: float64
```

Note that the slice did not return only the values but each element (index label and value) of the `Series` with the specified labels.

The `.head()` and `.tail()` methods are provided by pandas to examine just the first or last few records in a `Series`. By default, these return the first or last five rows, respectively, but you can use the `n` parameter or just pass in an integer to specify the number of rows:

```
In [6]:
   s.head()


   0    1.624345
   1   -0.611756
   2   -0.528172
   3   -1.072969
   4    0.865408
   dtype: float64


In [7]:
   s.tail()
```

```
Out[7]:
   95    0.077340
   96   -0.343854
   97    0.043597
   98   -0.620001
   99    0.698032
   dtype: float64
```

A `Series` consists of an index and a sequence of values. The index can be retrieved using the `.index` property:

```
In [8]:
   s.index


Out[8]:
   Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
   15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
   31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
   47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
   63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78,
   79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
   95, 96, 97, 98, 99], dtype='int64')
```

The values in the series using the `.values` property are as follows:

```
In [9]:
   s.values


Out[9]:
   array([ 1.62434536, -0.61175641, -0.52817175, -1.07296862,
           0.86540763, -2.3015387, 1.74481176, -0.7612069,
           0.3190391, -0.24937038, 1.46210794, -2.06014071,
           -0.3224172, -0.38405435, 1.13376944,

   ...

          -0.34385368, 0.04359686, -0.62000084, 0.69803203])
```

When creating a `Series` and not explicitly setting the index label values via the `Series` constructor, pandas will assign sequential integer values starting at `0`. To specify non-default index labels, use the `index` parameter of the `Series` object constructor or assign them using the `.index` property after creation.

The following command creates a `Series` and sets the index labels at the time of construction:

```
In [10]:
   s2 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
   s2


Out[10]:
   a    1
   b    2
   c    3
   d    4
   dtype: int64
```

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

```
In [11]:
   s2 = pd.Series({'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5})
   s2


Out[11]:
   a    1
   b    2
   c    3
   d    4
   e    5
   dtype: int64
```

Size, shape, uniqueness, and counts of values

There are several useful methods of determining the size of a `Series` as well as to get measurements of the distinct values and their quantities that are contained within the `Series`.

The number of elements in a `Series` can be determined using the `len()` function:

```
In [12]:
   s = pd.Series([10, 0, 1, 1, 2, 3, 4, 5, 6, np.nan])
   len(s)

Out[12]:
   10
```

This can also be determined using the `.shape` property, which returns a tuple containing the dimensionality of the `Series`. Since a `Series` is one-dimensional, only the length value is provided in the tuple:

```
In [13]:
   s.shape

Out[13]:
   (10,)
```

The number of rows in a `Series` that do not have a value of NaN can be determined with the `.count()` method:

```
In [14]:
   s.count()

Out[14]:
   9
```

To determine all of the unique values in a `Series`, pandas provides the `.unique()` method:

```
In [15]:
   s.unique()

Out[15]:
   array([5.,  0.,  1.,  2.,  3.,  4.,  5.,  6.,  nan])
   dtype: int64
```

The count of each of the unique items in a `Series` can be obtained using `.value_counts()`:

```
In [16]:
   s.value_counts()

Out[16]:
   5    2
   1    2
   6    1
   4    1
   3    1
   2    1
   0    1
dtype: int64
```

This result is sorted by pandas such that the counts are descending so that the most common values are at the top, which can help with quick analysis of data.

Alignment via index labels

A fundamental difference between a NumPy `ndarray` and a pandas `Series` is the ability of a `Series` to automatically align data from another `Series` based upon label values before performing an operation. We will examine alignment using the following two `Series` objects:

```
In [17]:
   s3 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
   s3

Out[17]:
   a    1
   b    2
   c    3
   d    4
   dtype: int64

In [18]:
   s4 = pd.Series([4, 3, 2, 1], index=['d', 'c', 'b', 'a'])
   s4
```

```
Out[18]:
   d    4
   c    3
   b    2
   a    1
   dtype: int64
```

The values in the two series are added in the following:

```
In [19]:
   s3 + s4

Out[19]:
   a    2
   b    4
   c    6
   d    8
   dtype: int64
```

The process of adding two `Series` objects differs from an array as it first aligns data based upon the index label values instead of simply applying the operation to elements in the same position. This becomes significantly powerful when using the pandas `Series` to combine data based upon labels instead of having to first order the data manually.

This is a very different result than if it was a pure NumPy `ndarray` being added. A NumPy `ndarray` would add the items in identical positions of each array, resulting in different values, as shown here:

```
In [20]:
   a1 = np.array([1, 2, 3, 4])
   a2 = np.array([4, 3, 2, 1])
   a1 + a2

Out[20]:
   array([5, 5, 5, 5])
```

Creating a DataFrame

There are a several ways to create a `DataFrame`. Probably, the most straightforward one is creating it from a NumPy array. The following command creates a `DataFrame` from a two-dimensional NumPy array:

```
In [21]:
   pd.DataFrame(np.array([[10, 11], [20, 21]]))


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

Each row of the array forms a row in the `DataFrame`. Since we did not specify an index, pandas creates a default `int64` index in the same manner as a `Series`. Since we also did not specify column names, pandas also assigns the names for each column with a zero-based integer series.

A `DataFrame` can also be initialized by passing a list of `Series` objects:

```
In [22]:
   df1 = pd.DataFrame([pd.Series(np.arange(10, 15)),
                       pd.Series(np.arange(15, 20))])
   df1


Out[22]:
       0   1   2   3   4
   0  10  11  12  13  14
   1  15  16  17  18  19
```

The dimensions of a `DataFrame` can be determined using its `.shape` property. A `DataFrame` is always two-dimensional. The shape informs us with the first value the number of rows and with the second the number of columns:

```
In [23]:
   df1.shape


Out[23]:
   (2, 5)
```

Column names can be specified at the time of creating the `DataFrame` using the `columns` parameter of the `DataFrame` constructor:

```
In [24]:
   df = pd.DataFrame(np.array([[10, 11], [20, 21]]),
                     columns=['a', 'b'])
   df

Out[24]:
       a   b
   0  10  11
   1  20  21
```

The names of the columns of a `DataFrame` can be accessed with its `.columns` property:

```
In [25]:
   df.columns

Out[25]:
   Index([u'a', u'b'], dtype='object')
```

The names of the columns can be changed by assigning the `.columns` property with a list of new names:

```
In [26]:
   df.columns = ['c1', 'c2']
   df

Out[26]:
      c1  c2
   0  10  11
   1  20  21
```

Index labels can likewise be assigned using the `index` parameter of the constructor or by assigning a list directly to the `.index` property:

```
In [27]:
   df = pd.DataFrame(np.array([[0, 1], [2, 3]]),
                     columns=['c1', 'c2'],
```

```
                    index=['r1', 'r2'])
   df
```

```
Out[27]:
       c1  c2
   r1   0   1
   r2   2   3
```

Like the `Series`, the index of a `DataFrame` can be accessed with its `.index` property:

```
In [28]:
   df.index
```

```
Out[28]:
   Index([u'r1', u'r2'], dtype='object')
```

Likewise, the values can be accessed using the `.values` property. Note that the result is a multidimensional array:

```
In [29]:
    df.values
```

```
Out[29]:
   array([[0, 1],
          [2, 3]])
```

A `DataFrame` can also be created by passing a dictionary containing one or more `Series` objects, where the dictionary keys contain the column names and each Series is one column of data:

```
In [30]:
   s1 = pd.Series(np.arange(1, 6, 1))
   s2 = pd.Series(np.arange(6, 11, 1))
   pd.DataFrame({'c1': s1, 'c2': s2})
```

```
Out[30]:
      c1  c2
   0   1   6
   1   2   7
   2   3   8
```

```
   3   4   9
   4   5  10
```

A `DataFrame` also does automatic alignment of the data for each `Series` passed in by a dictionary. As a demonstration, the following command adds a third column in the `DataFrame` initialization. This third `Series` contains two values and will specify its index. When the `DataFrame` is created, all Series in the dictionary are aligned with each other by the index label as it is added to the `DataFrame`:

```
In [31]:
   s3 = pd.Series(np.arange(12, 14), index=[1, 2])
   pd.DataFrame({'c1': s1, 'c2': s2, 'c3': s3})


Out[31]:
      c1  c2  c3
   0   1   6 NaN
   1   2   7  12
   2   3   8  13
   3   4   9 NaN
   4   5  10 NaN
```

The first two `Series` did not have an index specified so they both were indexed with `0` to `4`. The third `Series` has index values; therefore, the values for those indices are placed in the `DataFrame` in the row with the matching index from the previous columns. Then, pandas automatically fills in NaN for the values that were not supplied.

Example data

Wherever possible, the code samples in this chapter will utilize a dataset provided with the code bundle of the book. This dataset makes the examples a little less academic in nature. These will be read from files using the `pd.read_csv()` function, which will load the sample data from the file into a `DataFrame`.

The dataset we will use is a snapshot of the S&P 500 from Yahoo! Finance. For now, we will load this data into a `DataFrame` that can be used to demonstrate various operations. This code only uses four specific columns of data in the file by specifying those columns via the `usecols` parameter to `pd.read_csv()`. The following command reads in the 50 lines of data:

```
In [32]:
   sp500 = pd.read_csv("sp500.csv",
                       index_col='Symbol',
                       usecols=[0, 2, 3, 7])
```

We can examine the first five rows of the `DataFrame` using the `.head()` method:

```
In [33]:
   sp500.head()

Out[33]:
                             Sector   Price  Book Value
   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
```

The index of the `DataFrame` consists of the symbols for the 500 stocks representing the S&P 500:

```
In [34]:
   sp500.index

Out[34]:
   Index([u'MMM', u'ABT', u'ABBV', u'ACN', u'ACE', u'ACT',
   u'ADBE', u'AES', u'AET', u'AFL', u'A', u'GAS', u'APD', u'ARG',
   u'AKAM', u'AA', u'ALXN', u'ATI', u'ALLE', u'AGN', u'ADS',
   u'ALL', u'ALTR', u'MO', u'AMZN', u'AEE', u'AEP', u'AXP',
   u'AIG', u'AMT', u'AMP', u'ABC', u'AME', u'AMGN', u'APH',
   u'APC', u'ADI', u'AON', u'APA', ...], dtype='object')
```

Selecting columns of a DataFrame

Selecting the data in specific columns of a `DataFrame` is performed using the `[]` operator. This can be passed to either a single object or a list of objects. These objects are then used to look up columns either by the zero-based location or by matching the objects to the values in the columns index.

Passing a single integer, or a list of integers, to `[]` will have the `DataFrame` attempt to perform a location-based lookup of the columns. The following command retrieves the data in the second and third columns:

```
In [35]:
   sp500[[1, 2]].head(3)


Out[35]:
             Price  Book Value
   Symbol
   MMM      141.14       26.668
   ABT       39.60       15.573
   ABBV      53.95        2.954
```

Selecting columns by passing a list of values will result in another `DataFrame` with data copied from the original `DataFrame`. This is true even if the list only has a single integer value, as the following command demonstrates:

```
In [36]:
   sp500[[1]].head(3)


Out[36]:
             Price
   Symbol
   MMM      141.14
   ABT       39.60
   ABBV      53.95
```

Note that even though we asked for just a single column by position, the value was still in a list passed to the `[]` operator, hence the double set of brackets `[[]]`. This is important, as not passing a list always results in a value-based lookup of the column.

If the values passed to `[]` consist of non-integers, then the `DataFrame` will attempt to match the values to the values in the columns index. The following command retrieves the `Price` column by name:

```
In [37]:
   sp500['Price']
```

```
Out[37]:
   Symbol
   MMM     141.14
   ABT      39.60
   ABBV     53.95
   ACN      79.79
            ...
   YUM      74.77
   ZMH     101.84
   ZION     28.43
   ZTS      30.53
   Name: Price, dtype: float64
```

Multiple columns can be selected by name by passing a list of the column names and results in a `DataFrame` (even if a single item is passed in the list):

```
In [38]:
   sp500[['Price', 'Sector']]


Out[38]:
              Price                  Sector
   Symbol
   MMM     141.14             Industrials
   ABT      39.60             Health Care
   ABBV     53.95             Health Care
   ACN      79.79  Information Technology
   ...        ...                     ...
   YUM      74.77  Consumer Discretionary
   ZMH     101.84             Health Care
   ZION     28.43              Financials
   ZTS      30.53             Health Care
   [500 rows x 2 columns]
```

Columns can also be retrieved using what is referred to as attribute access. Each column in a `DataFrame` dynamically adds a property to the `DataFrame` for each column where the name of the property is the name of the column. Since this selects a single column, the resulting value is a `Series`:

```
In [39]:
   sp500.Price

Out[39]:
   Symbol
   MMM     141.14
   ABT      39.60
   ABBV     53.95
   ACN      79.79
            ...
   YUM      74.77
   ZMH     101.84
   ZION     28.43
   ZTS      30.53
   Name: Price, dtype: float64
```

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

Selecting rows of a DataFrame using the index

The elements of an array or `Series` are selected using the `[]` operator. The `DataFrame` overloads `[]` to select columns instead of rows except for a specific case of slicing. Therefore, most operations of selecting one or more rows in a `DataFrame` require alternate methods to using `[]`.

Understanding this is important in pandas as it is a common mistake to try to select rows using `[]` due to familiarity with other languages or data structures. When doing so, errors are often received and can often be difficult to diagnose without realizing that `[]` is working along a completely different axis than with a `Series` object.

Row selection using `index` on a `DataFrame` then breaks down into the following general categories of operations:

- Slicing using the `[]` operator
- Label- or location-based lookup using `.loc`, `.iloc`, and `.ix`
- Scalar lookup by label or location using `.at` and `.iat`

We will briefly examine each of these techniques and attributes. Remember, all of these are working against the content of the index of the `DataFrame`. There is no involvement of data in the columns when selecting rows. We will cover this in the next section on Boolean selection.

Slicing using the [] operator

Slicing a `DataFrame` across its index is syntactically identical to slicing a `Series`. Because of this, we will not go into the details of the various permutations of slices in this section and only give representative examples applied to a `DataFrame`.

Slicing works along both positions and labels. The following command demonstrates several examples of slicing by position:

```
In [40]:
   sp500[:3]


Out[40]:
                            Sector   Price  Book Value
   Symbol
   MMM                  Industrials  141.14      26.668
   ABT                  Health Care   39.60      15.573
   ABBV                 Health Care   53.95       2.954
```

The following command returns rows starting with the `XYL` label through the `YUM` label:

```
In [41]:
   sp500['XYL':'YUM']


Out[41]:
                            Sector  Price  Book Value
   Symbol
   XYL                  Industrials  38.42      12.127
   YHOO     Information Technology  35.02      12.768
   YUM      Consumer Discretionary  74.77       5.147
```

In general, although slicing a `DataFrame` has its uses, high-performance systems tend to shy away from it and use other methods. Additionally, the slice notation for rows on a `DataFrame` using integers can be confusing as it looks like accessing columns by position and hence can lead to subtle bugs.

Selecting rows by the index label and location – .loc[] and .iloc[]

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

```
In [42]:
   sp500.loc['MMM']

Out[42]:
   Sector         Industrials
   Price              141.14
   Book Value         26.668
   Name: MMM, dtype: object

In [43]:
   sp500.loc[['MMM', 'MSFT']]

Out[43]:
                            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 [44]:
   sp500.iloc[[0, 2]]

Out[44]:
                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 `index` of a specific label value, which can then be used to retrieve the row(s):

```
In [45]:
   i1 = sp500.index.get_loc('MMM')
   i2 = sp500.index.get_loc('A')
   i1, i2

Out[45]:
   (0, 10)

In [46]:
   sp500.iloc[[i1, i2]]

Out[46]:
                Sector   Price  Book Value
   Symbol
   MMM     Industrials  141.14      26.668
   A       Health Care   56.18      16.928
```

Selecting rows by the index label and/or location – .ix[]

A `DataFrame` also contains an `.ix[]` property that can be used to look up rows by either the index label or location, essentially combining `.loc` and `.iloc` in one. The following command looks up rows by the index label by passing a list of non-integers:

```
In [47]:
   sp500.ix[['MSFT', 'ZTS']]

Out[47]:
                           Sector  Price  Book Value
   Symbol
   MSFT    Information Technology  40.12      10.584
   ZTS                Health Care  30.53       2.150
```

The location-based lookup can be performed by passing a list of integers:

```
In [48]:
   sp500.ix[[10, 200, 450]]

Out[48]:
                     Sector  Price  Book Value
   Symbol
   A            Health Care  56.18      16.928
   GIS     Consumer Staples  53.81      10.236
   TRV            Financials  92.86      73.056
```

In general, the use of `.ix` is not preferred due to potential confusion. The use of `.loc` and `.iloc` is recommended and also results in higher performance.

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

Scalar values can be looked up by label using `.at[]` by passing the row label and then the column name/value:

```
In [49]:
   sp500.at['MMM', 'Price']

Out[49]:
   141.14
```

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

```
In [50]:
   sp500.iat[0, 1]

Out[50]:
   141.14
```

Selecting rows using the Boolean selection

Rows can also be selected using the Boolean selection with an array calculated from the result of applying a log logical condition to the values in any of the columns. This allows us to build more complicated selections than those based simply upon index labels or positions.

Consider the following command that is an array of all companies that have a price below `100.0`:

```
In [51]:
   sp500.Price < 100


Out[51]:
   Symbol
   MMM       False
   ABT        True
   ABBV       True
   ACN        True
   ...
   YUM        True
   ZMH       False
   ZION       True
   ZTS        True
   Name: Price, Length: 500, dtype: bool
```

This results in a `Series` that can be used to select rows where the value is `True`:

```
In [52]:
   sp500[sp500.Price < 100]


Out[52]:
                            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
   ...                        ...    ...         ...
   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]
```

Multiple conditions can be put together using parentheses, and at the same time, it is possible to select only a subset of the columns. The following command retrieves the symbols and price for all stocks with a price less than `10` and greater than `0`:

```
In [53]:
   sp500[(sp500.Price < 10) & (sp500.Price > 0)] [['Price']]

Out[53]:
           Price
   Symbol
   FTR      5.81
   HCBK     9.80
   HBAN     9.10
   SLM      8.82
   WIN      9.38
```

Arithmetic on a DataFrame

Arithmetic operations using scalar values will be applied to every element of a `DataFrame`. To demonstrate this, we will use a `DataFrame` initialized with random values:

```
In [54]:
   np.random.seed(123456)
   df = pd.DataFrame(np.random.randn(5, 4),
                     columns=['A', 'B', 'C', 'D'])
   df

Out[54]:
             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` and will return a new `DataFrame` with the results (leaving the original unchanged):

```
In [55]:
   df * 2


Out[55]:
            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. To demonstrate this, the following example retrieves the first row of the `DataFrame` and then subtracts this from each row of the `DataFrame`. The `Series` is being broadcast by pandas to each row of the `DataFrame`, which aligns each series item with the `DataFrame` item of the same index label and then applies the minus operator on the matched values:

```
In [56]:
   df - df.iloc[0]


Out[56]:
            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
```

An arithmetic operation between two `DataFrame` objects will align with both the column and index labels. The following command extracts a small portion of `df` and subtracts it from `df`. The result demonstrates that the aligned values subtract to `0`, while the others are set to `NaN`:

```
In [57]:
   subframe = df[1:4][['B', 'C']]
   subframe

Out[57]:
              B         C
   1  -0.173215  0.119209
   2  -2.104569 -0.494929
   1  -0.706771 -1.039575

In [58]:
   df - subframe

Out[58]:
       A   B   C   D
   0 NaN NaN NaN NaN
   1 NaN   0   0 NaN
   2 NaN   0   0 NaN
   3 NaN   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 particular axis. The following command demonstrates subtraction along a column axis by using the `DataFrame` object; the `.sub()` method subtracts the `A` column from every column:

```
In [59]:
   a_col = df['A']
   df.sub(a_col, axis=0)
```

```
Out[59]:
      A         B         C         D
   0  0 -0.751976 -1.978171 -1.604745
   1  0 -1.385327 -1.092903 -2.256348
   2  0 -1.242720  0.366920  1.933653
   3  0 -1.428326 -1.761130 -0.449695
   4  0  0.991993  0.701204 -0.662428
```

Reindexing the Series and DataFrame objects

Reindexing in pandas is a process that makes the data present in a `Series` or `DataFrame` match with a given set of labels along a particular axis. This is core to the functionalities of pandas as it enables label alignment across multiple objects.

The process of performing a reindex does the following:

- Reorders existing data to match a set of labels
- Inserts NaN markers where no data exists for a label
- Fills missing data for a label using a type of logic (defaulting to adding NaNs)

The following is a simple example of reindexing a `Series`. The following `Series` has an index with numerical values, and the index is modified to be alphabetic by simply assigning a list of characters to the `.index` property, making the values able to be accessed via the character labels in the new index:

```
In [60]:
   np.random.seed(1)
   s = pd.Series(np.random.randn(5))
   s


Out[60]:
   0    1.624345
   1   -0.611756
   2   -0.528172
   3   -1.072969
   4    0.865408
```

```
   dtype: float64

In [61]:
   s.index = ['a', 'b', 'c', 'd', 'e']
   s

Out[61]:
   a    1.624345
   b   -0.611756
   c   -0.528172
   d   -1.072969
   e    0.865408
   dtype: float64
```

Greater flexibility in creating a new index is provided using the `.reindex()` method. One example of flexibility of `.reindex()` over assigning the `.index` property directly is that the list provided to `.reindex()` can be of a different length than the number of rows in the `Series`:

```
In [62]:
   s2 = s.reindex(['a', 'c', 'e', 'g'])
   s2['a'] = 0
   s2

Out[62]:
   a    0.000000
   c   -0.528172
   e    0.865408
   g         NaN
   dtype: float64

In [63]:
   s['a']

Out[63]:
   1.6243453636632417
```

There are several things here that are important to point out about `.reindex()`:

- The result is a new `Series` (the value of `s['a']`) remains unchanged) with the labels provided as a parameter, and if the existing `Series` had a matching label, that value is copied to the new `Series`
- If there is an index label created for which the `Series` did not have an already existing label, the value will be assigned NaN

Reindexing is also useful when you want to align two `Series` to perform an operation on matching elements from each series, but for some reason, the two `Series` had index labels that would not initially align.

The following example demonstrates this, where the first `Series` has indices as sequential integers, but the second one has string representation of what would be sequential integers.

The addition of both `Series` has the following result, which is all NaNs and an `Int64Index` that has repeated label values:

```
In [64]:
   s1 = pd.Series([0, 1, 2], index=[0, 1, 2])
   s2 = pd.Series([3, 4, 5], index=['0', '1', '2'])
   s1 + s2


Out[64]:
   0   NaN
   1   NaN
   2   NaN
   0   NaN
   1   NaN
   2   NaN
   dtype: float64
```

This is almost an epic fail situation that can happen if values intended to be numeric are presented with one being numeric and the other as string. In this case, pandas first tries to align with the indices and finds no matches, so it copies the index labels from the first `Series` and tries to append the indices from the second `Series`. But since they are a different type, it defaults back to a zero-based integer sequence, which results in duplicate values. And finally, all the resulting values are NaN because the operation tries to add the item in the first series with the integer label `0`, which has the value `0` but can't find the item in the other series with the integer label `0`; therefore, the result is NaN (and this fails six times in this case).

Once this situation is identified, it becomes fairly simple to fix with reindexing the second `Series` by casting the values to `int`:

```
In [65]:
   s2.index = s2.index.values.astype(int)
   s1 + s2


Out[65]:
   0    3
   1    5
   2    7
dtype: int64
```

The default action of inserting NaN as a missing value during `.reindex()` can be changed using `fill_value` of the method. The following command demonstrates using `0` instead of NaN:

```
In [66]:
   s2 = s.copy()
   s2.reindex(['a', 'f'], fill_value=0)


Out[66]:
   a   1.624345
   f    0.000000
   dtype: float64
```

When performing a reindex 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 of values in *Chapter 4*, *Time-series*, but the following examples introduce the concept. To demonstrate the concept, let's use the following `Series`:

```
In [67]:
   s3 = pd.Series(['red', 'green', 'blue'], index=[0, 3, 5])
   s3


Out[67]:
   0      red
   3    green
```

```
   5     blue
   dtype: object
```

The following command demonstrates forward filling, often referred to as the last known value. The Series is reindexed to create a contiguous integer index, and using the `method='ffill'` parameter, any new index labels are assigned a value from the previously seen value along the `Series`. Here's the command:

```
In [68]:
   s3.reindex(np.arange(0,7), method='ffill')


Out[68]:
   0      red
   1      red
   2      red
   3    green
   4    green
   5     blue
   6     blue
   dtype: object
```

By contrast, the result of the same `Series` when backwards filling using the `method='bfill'` parameter is shown here:

```
In [69]:
   s3.reindex(np.arange(0,7), method='bfill')


Out[69]:
   0      red
   1    green
   2    green
   3    green
   4     blue
   5     blue
   6      NaN
dtype: object
```

Summary

In this chapter, we briefly overviewed the pandas `Series` and `DataFrame` objects, how they are used to represent data, and how to select data in both via queries, columns, and indices. The concept of reindexing both classes of objects is also introduced, and as we get into the later chapters, it will be common to perform reindexing of time-series data.

In the next chapter, we will examine indexing in more depth with an eye towards how performing various aggregations of data can derive results from the information represented in pandas. As we progress into more specific financial analysis, this combination of reindexing and aggregation will form the basis of much of the analysis performed later in the book.

3
Reshaping, Reorganizing, and Aggregating

In the first two chapters, we gave you a general overview of pandas and examined some of the basics of the pandas `DataFrame`. Our coverage of the `DataFrame` was focused solely upon simple manipulation of a single `DataFrame`, such as adding and removing columns and rows, indexing the contents, selecting content, basic indexing, and performing simple arithmetic upon its data.

In this chapter, we will expand our scope of data operations on `DataFrame` objects to include more complex techniques of manipulating data and deriving results from grouped sets of financial data. The examples in this chapter will focus on retrieving, organizing, reshaping, and grouping/aggregating data to be able to perform basic statistical operations.

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

- Loading historical stock data from the Web or from files
- Concatenating and merging stock price data along multiple axes
- Merging data in multiple `DataFrame` objects
- Pivoting stock price data between axes
- Stacking, unstacking, and melting of stock data
- Splitting and grouping stock data to be able to calculate aggregate results

Notebook setup

To utilize the examples in this chapter, we will need to include the following imports and settings in either your IPython or IPython Notebook environment, as shown here:

```
In [1]:
   import pandas as pd
   import numpy as np
   import datetime
   import matplotlib.pyplot as plt
   %matplotlib inline
   pd.set_option('display.notebook_repr_html', False)
   pd.set_option('display.max_columns', 15)
   pd.set_option('display.max_rows', 8)
   pd.set_option('precision', 3)
```

Loading historical stock data

The examples in this chapter will utilize data extracted from Yahoo! Finance. This information can be extracted live from the web services or from files provided with the source. This data consists of stock prices for MSFT and AAPL for the year 2012.

The following command can be used to load the stock information directly from the Web:

```
In [2]:
   import pandas.io.data as web

   start = datetime.datetime(2012, 1, 1)
   end = datetime.datetime(2012, 12, 30)

   msft = web.DataReader("MSFT", 'yahoo', start, end)
   aapl = web.DataReader("AAPL", 'yahoo', start, end)

   # these save the data to file - optional for the examples
   #msft.to_csv("msft.csv")
   #aapl.to_csv("aapl.csv")
```

If you are not online or just want to load the data from the file, you can use the following command. I actually recommend using this data as even though the online data is historical, the adjusted close values are sometimes changed to represent other events and can potentially cause some output different than what is in the text:

```
In [3]:
   msft = pd.read_csv("msft.csv", index_col=0, parse_dates=True)
   aapl = pd.read_csv("aapl.csv", index_col=0, parse_dates=True)
```

Organizing the data for the examples

With this information in hand, various slices of data are created to facilitate the various examples through the chapter, as shown here:

```
In [4]:
   msft[:3]

Out[4]:
                Open   High    Low  Close    Volume  Adj Close
   Date
   2012-01-03  26.55  26.96  26.39  26.77  64731500   24.42183
   2012-01-04  26.82  27.47  26.78  27.40  80516100   24.99657
   2012-01-05  27.38  27.73  27.29  27.68  56081400   25.25201

In [5]:
   aapl[:3]

Out[5]:
                 Open    High     Low   Close    Volume  Adj Close
   Date
   2012-01-03  409.40  412.50  409.00  411.23  75555200      55.41
   2012-01-04  410.00  414.68  409.28  413.44  65005500      55.71
   2012-01-05  414.95  418.55  412.67  418.03  67817400      56.33
```

Reorganizing and reshaping data

When working with financial information, it is often the case that data retrieved from almost any data source will not be in the format that you need to perform the analyses that you want.

Or perhaps, just as likely, the data from a specific source may be incomplete and require collection of data from another source, at which point, the data needs to be either concatenated or merged through join-like operations across the data.

Even if the data is complete or after combining it from various sources, it may still be organized in a manner that is not conducive to a specific type of analysis. Hence, it needs to be restructured.

Fortunately, pandas provides rich capabilities for concatenating, merging, and pivoting data. These following sections take us through several common scenarios of each, using stock data.

Concatenating multiple DataFrame objects

Concatenation in pandas is the process of creating a new pandas object by combining data from two (or more pandas) objects into a new pandas object along a single, specified axis of the two objects. Concatenation with stock data is useful to combine values taken at different time periods, to create additional columns representing other measurements at a particular date and time for a specific stock, or to add a column for the same measurement of a different stock but for the same time period.

`DataFrame` objects are concatenated by pandas along a specified axis—the two axes being the index labels of the rows and the columns. This is done by first extracting the labels from both the `DataFrame` object indices along the specified axis, using that set as the index of the new `DataFrame`, and then copying the values along the other axis into the result in an orderly manner, that is, from the first `DataFrame` and then from the second `DataFrame`.

The result of a concatenation always contains the union of the number of items in both objects along the specific axis. As we will see later in this section, this is different than a merge or join that could result in the resulting number of items not necessarily being equivalent to the union of the number of items in the source `DataFrame` objects.

The tricky part of concatenation is how pandas deals with the items along the other axis during the concatenation. The set of values, be they rows when concatenating along the columns or columns when concatenating along rows, is defined using relational algebra on the values in that axis's index.

To demonstrate various forms of concatenation, we will start with the following data that shows the adjusted closing prices for MSFT for the months of January and February 2012 represented in the following command. This dataset simulates the retrieval of stock information representing two different time periods and stores the data in two different `DataFrame` objects, as shown here:

```
In [6]:
   msftA01 = msft['2012-01'][['Adj Close']]
   msftA02 = msft['2012-02'][['Adj Close']]
   msftA01[:3]


Out[6]:
               Adj Close
   Date
   2012-01-03      24.42
   2012-01-04      25.00
   2012-01-05      25.25

In [7]:
   msftA02[:3]


Out[7]:
               Adj Close
   Date
   2012-02-01      27.27
   2012-02-02      27.32
   2012-02-03      27.59
```

To combine both of these sets of data into a single `DataFrame`, we perform a concatenation. To demonstrate the following concatenates, the first three rows from each `DataFrame` are as follows:

```
In [8]:
   pd.concat([msftA01.head(3), msftA02.head(3)])


Out[8]:
               Adj Close
   Date
   2012-01-03      24.42
```

```
   2012-01-04      25.00
   2012-01-05      25.25
   2012-02-01      27.27
   2012-02-02      27.32
   2012-02-03      27.59
```

The resulting `DataFrame` contains an index identical in structure to both of the objects, with labels from the first object and then the second object copied into the new object. At first glance, it may appear that the concatenation is a pure copy of the rows from each `DataFrame` into the new `DataFrame`, but as we will see, the process is more elaborate (and hence flexible). This will become more evident as we take a look at more examples.

The following example concatenates the first five adjusted close values in January for both MSFT and AAPL. These have identical index labels and result in duplicate index labels in the new `DataFrame`. During a concatenation along the row axis, pandas will not align the index labels. They will be copied and this can create duplicate, identical index labels:

```
In [9]:
   aaplA01 = aapl['2012-01'][['Adj Close']]
   withDups = pd.concat([msftA01[:3], aaplA01[:3])
   withDups


Out[9]:
               Adj Close
   Date
   2012-01-03      24.42
   2012-01-04      25.00
   2012-01-05      25.25
   2012-01-03      55.41
   2012-01-04      55.71
   2012-01-05      56.33
```

This has resulted in duplicated index labels and will result in multiple items being returned for those labels, as shown here:

```
In [10]:
   withDups.ix['2012-01-03']


Out[10]:
```

```
              Adj Close
   Date
   2012-01-03      24.42
   2012-01-03      55.41
```

This concatenation has lost whether the `Adj Close` value in the new `DataFrame` came from the MSFT or AAPL `DataFrame`. This source `DataFrame` of each row can be preserved during concatenation by specifying the value of the keys in the new `DataFrame`. These keys will add an additional level to the index (making a `MultiIndex`), which then can be used to identify the source `DataFrame`:

```
In [11]:
   closes = pd.concat([msftA01[:3], aaplA01[:3]],
                      keys=['MSFT', 'AAPL'])
   closes


Out[11]:
                    Adj Close
         Date
   MSFT 2012-01-03      24.42
        2012-01-04      25.00
        2012-01-05      25.25
   AAPL 2012-01-03      55.41
        2012-01-04      55.71
        2012-01-05      56.33
```

Using this new `MultiIndex`, it is then possible to extract the values for either stock from this new `DataFrame` by only using the index labels. The following command does this for the MSFT entries:

```
In [12]:
   closes.ix['MSFT'][:3]


Out[12]:
              Adj Close
   Date
   2012-01-03      24.42
   2012-01-04      25.00
   2012-01-05      25.25
```

Concatenation along the row axis can also be performed using `DataFrame` objects with multiple columns. The following command modifies the previous example to use the `Adj Close` and `Volume` columns in each `DataFrame`. Although not evident from the output, there are duplicate rows for each date in the result:

```
In [13]:
   msftAV = msft[['Adj Close', 'Volume']]
   aaplAV = msft[['Adj Close', 'Volume']]
   pd.concat([msftAV, aaplAV])


Out[13]:
               Adj Close     Volume
   Date
   2012-01-03      24.42   64731500
   2012-01-04      25.00   80516100
   2012-01-05      25.25   56081400
   2012-01-06      25.64   99455500
   ...               ...        ...
   2012-12-24      70.72   43938300
   2012-12-26      69.74   75609100
   2012-12-27      70.02  113780100
   2012-12-28      69.28   88569600

   [498 rows x 2 columns]
```

The columns in the `DataFrame` objects in a concatenation do not have to have the same names. The following command demonstrates a concatenation where the `aaplA DataFrame` only consists of the `Adj Close` column, whereas the MSFT `DataFrame` has both `Adj Close` and `Volume` columns:

```
In [14]:
   aaplA = aapl[['Adj Close']]
   pd.concat([msftAV, aaplA])


Out[14]:
               Adj Close    Volume
   Date
   2012-01-03      24.42  64731500
   2012-01-04      25.00  80516100
```

```
2012-01-05      25.25  56081400
2012-01-06      25.64  99455500
...               ...       ...
2012-12-24      70.72       NaN
2012-12-26      69.74       NaN
2012-12-27      70.02       NaN
2012-12-28      69.28       NaN

[498 rows x 2 columns]
```

Since the rows originating from the `aapl DataFrame` do not have a `Volume` column, pandas inserts `NaN` into the `Volume` column for those rows.

The set of columns that results from a concatenation along the row axis is the result of relational algebra across the names of the columns. In this default scenario, the resulting column is the union of column names from each `DataFrame`. This can be changed to an intersection using the `join` parameter. The following command makes the set of resulting columns the intersection of the column names by specifying `join='inner'`:

```
In [15]:
   pd.concat([msftAV, aaplA], join='inner')


Out[15]:
              Adj Close
   Date
   2012-01-03      24.42
   2012-01-04      25.00
   2012-01-05      25.25
   2012-01-06      25.64
   ...               ...
   2012-12-24      70.72
   2012-12-26      69.74
   2012-12-27      70.02
   2012-12-28      69.28

   [498 rows x 1 columns]
```

We can change the axis for concatenation to the columns using `axis=1`:

```
In [16]:
   msftA = msft[['Adj Close']]
   closes = pd.concat([msftA, aaplA], axis=1)
   closes[:3]

Out[16]:
               Adj Close  Adj Close
   Date
   2012-01-03      24.42      55.41
   2012-01-04      25.00      55.71
   2012-01-05      25.25      56.33
```

Note that this `DataFrame` has two `Adj Close` columns and only consists of `249` rows (the concatenation along `axis=0` has `498`). Because of the use of `axis=1`, the union of the index labels is derived instead from the column names, and the columns are copied one by one in an orderly manner from the `DataFrame` objects, including duplicates.

It is also possible to concatenate with multiple columns where the `DataFrame` objects do not have the same set of index labels. The following command concatenates the first five `msftAV` values and the first three `aaplAV` values:

```
In [17]:
   pd.concat([msftAV[:5], aaplAV[:3]], axis=1,
             keys=['MSFT', 'AAPL'])

Out[17]:
                    MSFT                 AAPL
               Adj Close    Volume Adj Close    Volume
   Date
   2012-01-03      24.42  64731500      55.41  75555200
   2012-01-04      25.00  80516100      55.71  65005500
   2012-01-05      25.25  56081400      56.33  67817400
   2012-01-06      25.64  99455500        NaN       NaN
   2012-01-09      25.31  59706800        NaN       NaN
```

This results in duplicate column names, so we use the `keys` parameter to create `MultiIndex` for the columns. Since there were row index labels that were not found in `aaplCV`, pandas fills those with `NaN`.

Just as with concatenation along the row axis, the type of join performed by `pd.concat()` can be changed using the `join` parameter. The following command performs an inner join instead of an outer join, which results in the intersection of row index labels:

```
In [18]:
   pd.concat([msftA[:5], aaplA[:3]], axis=1,
             join='inner', keys=['MSFT', 'AAPL'])

Out[18]:
                    MSFT       AAPL
                Adj Close Adj Close
   Date
   2012-01-03      24.42      55.41
   2012-01-04      25.00      55.71
   2012-01-05      25.25      56.33
```

The resulting `DataFrame` only has three rows because those index labels were the only ones in common in the two concatenated `DataFrame` objects.

If you want to ignore indices in the result of `pd.concat()`, you can use the `ignore_index=True` parameter, which will drop the index and create a default zero-based integer index, as shown here:

```
In [19]:
   pd.concat([msftA[:3], aaplA[:3]], ignore_index=True)

Out[19]:
      Adj Close
   0      24.42
   1      25.00
   2      25.25
   3      55.41
   4      55.71
   5      56.33
```

Merging DataFrame objects

The combination of pandas objects is allowed using relational database-like join operations, high-performance in-memory operations, and the `pd.merge()` function.

Merging in pandas differs from concatenation in that the `pd.merge()` function combines data based on the values of the data in one or more columns instead of using the index label values along a specific axis.

The default process that `pd.merge()` uses is to first identify the columns the data of which will be used in the merge, and then to perform an inner join based upon that information. The columns used in the join are, by default, selected as those in both `DataFrame` objects with common names (an intersection of the column labels).

To demonstrate a merge, we will use the following two `DataFrame` objects, one with the volumes and the other with the adjusted close values for MSFT. Both have the index reset:

```
In [20]:
   msftAR = msftA.reset_index()
   msftVR = msft[['Volume']].reset_index()
   msftAR[:3]

Out[20]:
           Date  Adj Close
   0 2012-01-03      24.42
   1 2012-01-04      25.00
   2 2012-01-05      25.25

In [21]:
   msftVR[:3]

Out[21]:
           Date    Volume
   0 2012-01-03  64731500
   1 2012-01-04  80516100
   2 2012-01-05  56081400
```

Instead of using `Date` as the index, these have `Date` as a column so that it can be used in the merge. Our goal is to create a `DataFrame` that contains a `Date` column and both `AdjClose` and `Volume` columns. This can be accomplished with the following statement:

```
In [22]:
   msftCVR = pd.merge(msftAR, msftVR)
   msftCVR[:5]


Out[22]:
            Date  Adj Close    Volume
   0 2012-01-03      24.42  64731500
   1 2012-01-04      25.00  80516100
   2 2012-01-05      25.25  56081400
   3 2012-01-06      25.64  99455500
   4 2012-01-09      25.31  59706800
```

The column in common is `Date`; therefore, pandas performs an inner join on the values in that column across both `DataFrame` objects. Once that set is calculated, pandas copies in the appropriate values for each row from both `DataFrame` objects.

The types of joins supported by `pd.merge()` are similar to the different types of joins supported in relational databases. They are as follows:

- `left`: Use keys from the left `DataFrame` (equivalent to SQL's left-outer join)
- `right`: Use keys from the right `DataFrame` (equivalent to SQL's right-outer join)
- `outer`: Use the union of keys from both `DataFrame` objects (equivalent to SQL's full outer join)
- `inner`: Use the intersection of keys from both `DataFrame` objects (equivalent to SQL's inner join)

To demonstrate each difference in the results between inner and outer joins, we will use the following data:

```
In [23]:
   # we will demonstrate join semantics using this DataFrame
   msftAR0_5 = msftAR[0:5]
   msftAR0_5
```

```
Out[23]:
           Date  Adj Close
   0 2012-01-03      24.42
   1 2012-01-04      25.00
   2 2012-01-05      25.25
   3 2012-01-06      25.64
   4 2012-01-09      25.31

In [24]:
   msftVR2_4 = msftVR[2:4]
   msftVR2_4

Out[24]:
           Date    Volume
   2 2012-01-05  56081400
   3 2012-01-06  99455500
```

For an inner join, since there are only two rows with matching dates, the result only has two rows and merges the `DataFrame` objects where `Date` values are in common, as shown here:

```
In [25]:
   pd.merge(msftAR0_5, msftVR2_4)

Out[25]:
           Date  Adj Close    Volume
   0 2012-01-05      25.25  56081400
   1 2012-01-06      25.64  99455500
```

This can be changed to an outer join with `how='outer'`. All rows from the outer `DataFrame` are returned (`msftAR0_5`), and values not found in the inner `DataFrame` (`msftVR2_4`) are replaced with `NaN`:

```
In [26]:
   pd.merge(msftAR0_5, msftVR2_4, how='outer')

Out[26]:
           Date  Adj Close    Volume
   0 2012-01-03      24.42       NaN
```

```
   1 2012-01-04      25.00       NaN
   2 2012-01-05      25.25  56081400
   3 2012-01-06      25.64  99455500
   4 2012-01-09      25.31       NaN
```

Pivoting

Financial data is often stored in a format where the data is not normalized and, therefore, has repeated values in many columns or values that logically should exist in other tables. An example of this would be the following, where the historical prices for multiple stocks are represented in a single `DataFrame` using a `Symbol` column. The following command creates a `DataFrame` with this schema and populates the records:

```
In [27]:
   msft.insert(0, 'Symbol', 'MSFT')
   aapl.insert(0, 'Symbol', 'AAPL')
   combined = pd.concat([msft, aapl]).sort_index()
   s4p = combined.reset_index();
   s4p[:5]


Out[27]:
           Date Symbol   Open   High    Low  Close   Volume Adj Close
   0 2012-01-03   MSFT  26.55  26.96  26.39  26.77 64731500     24.42
   1 2012-01-03   AAPL 409.40 412.50 409.00 411.23 75555200     55.41
   2 2012-01-04   MSFT  26.82  27.47  26.78  27.40 80516100     25.00
   3 2012-01-04   AAPL 410.00 414.68 409.28 413.44 65005500     55.71
   4 2012-01-05   MSFT  27.38  27.73  27.29  27.68 56081400     25.25
```

Now let's suppose we want to extract, from this `DataFrame`, a new `DataFrame` that is indexed by date and has columns representing the `AdjClose` value for all of the stocks listed in the `Symbol` column. This can be performed using the `.pivot()` method of the `DataFrame`:

```
In [28]:
   closes = s4p.pivot(index='Date', columns='Symbol',
                      values='Adj Close')
   closes[:3]
```

```
Out[28]:
   Symbol       AAPL   MSFT
   Date
   2012-01-03  55.41  24.42
   2012-01-04  55.71  25.00
   2012-01-05  56.33  25.25
```

This has taken all the distinct values from the `Symbol` column, *pivoted* them into columns on the new `DataFrame`, and then entered the values in those columns from the `AdjClose` value for the specific symbol from the original `DataFrame`.

Stacking and unstacking

The `DataFrame` methods similar in operation to the `pivot` function are `.stack()` and `.unstack()`. Stacking unpivots the column labels into another level of the index. To demonstrate this, the following command pivots the MSFT and AAPL columns into the index:

```
In [29]:
   stackedCloses = closes.stack()
   stackedCloses


Out[29]:
   Date        Symbol
   2012-01-03  AAPL      55.41
               MSFT      24.42
   2012-01-04  AAPL      55.71
               MSFT      25.00
                         ...
   2012-12-27  AAPL      70.02
               MSFT      25.29
   2012-12-28  AAPL      69.28
               MSFT      24.91
   dtype: float64
```

This has created a new index with an additional level named `Symbol` (from the name of the columns index). Each row is then indexed by both `Date` and `Symbol`. And for each unique `Date` and `Symbol` level, pandas has inserted the appropriate `Adj Close` value.

The result of this allows the efficient lookup of any `Adj Close` value using the index. To look up the `Adj Close` value for `AAPL` on `2012-01-03`, we can use the following command:

```
In [30]:
   stackedCloses.ix['2012-01-03', 'AAPL']

Out[30]:
   55.41
```

The result here is equivalent to the following value-based lookup, but is significantly more efficient, uses less typing, and is also organized better, causing less mental clutter.

Using a `MultiIndex`, it is also possible to look up values for just a specific `Date`:

```
In [31]:
   stackedCloses.ix['2012-01-03']

Out[31]:
   Symbol
   AAPL      55.41
   MSFT      24.42
   dtype: float64
```

For a specific `Symbol`, here is the command:

```
In [32]:
   stackedCloses.ix[:, 'MSFT']

Out[32]:
   Date
   2012-01-03    24.42
   2012-01-04    25.00
   2012-01-05    25.25
   2012-01-06    25.64
                 ...
   2012-12-24    25.38
   2012-12-26    25.20
   2012-12-27    25.29
```

```
   2012-12-28    24.91
   dtype: float64
```

The `.unstack()` method performs the opposite function; that is, it pivots a level of an index into a column in a new `DataFrame`. The following command unstacks the last level of the `MultiIndex` and results in a `DataFrame` equivalent to the original `unstackedCloses`:

```
In [33]:
   unstackedCloses = stackedCloses.unstack()
   unstackedCloses[:3]


Out[33]:
   Symbol       AAPL   MSFT
   Date
   2012-01-03  55.41  24.42
   2012-01-04  55.71  25.00
   2012-01-05  56.33  25.25
```

Melting

Melting is the process of transforming a `DataFrame` into a format where each row represents a unique `id-variable` combination. The following command demonstrates melting the `s4p DataFrame` into an `id-variable` combination consisting of the `Date` and `Symbol` columns as the ID and the other columns mapped into the variables:

```
In [34]:
   melted = pd.melt(s4p, id_vars=['Date', 'Symbol'])
   melted[:5]


Out[34]:
           Date Symbol variable   value
   0 2012-01-03   MSFT     Open   26.55
   1 2012-01-03   AAPL     Open  409.40
   2 2012-01-04   MSFT     Open   26.82
   3 2012-01-04   AAPL     Open  410.00
   4 2012-01-05   MSFT     Open   27.38
```

During a melt, the column(s) specified by the `id_vars` parameter remain columns (in this case `Date` and `Symbol`). All other columns have their names mapped to the values in the `variable` column—one row for each `variable` column of an `id_var` column value combination.

This organization of data is useful to select chunks of information based upon a specific ID variable and then one or more variables. As an example, the following command returns all measurements for `2012-01-03` and the `MSFT` symbol:

```
In [35]:
   melted[(melted.Date=='2012-01-03') & melted.Symbol=='MSFT')]


Out[35]:
              Date Symbol   variable        value
   0    2012-01-03   MSFT       Open        26.55
   498  2012-01-03   MSFT       High        26.96
   996  2012-01-03   MSFT        Low        26.39
   1494 2012-01-03   MSFT      Close        26.77
   1992 2012-01-03   MSFT     Volume  64731500.00
   2490 2012-01-03   MSFT  Adj Close        24.42
```

Grouping and aggregating

Data in pandas can be easily split into groups and then summarized using various statistical and quantitative calculations. This process in pandas nomenclature is often referred to as the split-apply-combine pattern.

In this section, we will look at using this pattern as applied to stock data. We will split the data by various time and symbol combinations and then apply statistical operations to begin analyzing the risk and return on our sample data.

Splitting

Objects in pandas are split into groups using the `.groupby()` method. To demonstrate this, we will use the stock price data introduced earlier in the chapter but slightly reorganized to facilitate understanding of the grouping process:

```
In [36]:
   s4g = combined[['Symbol', 'AdjClose']].reset_index()
   s4g.insert(1, 'Year', pd.DatetimeIndex(s4g['Date']).year)
```

```
s4g.insert(2, 'Month[:5]',pd.DatetimeIndex(s4g['Date']).month)
s4g[:5]
```

```
Out[36]:
```

	Date	Year	Month	Symbol	Adj Close
0	2012-01-03	2012	1	MSFT	24.42
1	2012-01-03	2012	1	AAPL	55.41
2	2012-01-04	2012	1	MSFT	25.00
3	2012-01-04	2012	1	AAPL	55.71
4	2012-01-05	2012	1	MSFT	25.25

This data differs from before as only the `AdjClose` value is utilized, and the `Date` column is broken apart into two other columns, `Year` and `Month`. This splitting of the date is done to be able to provide the ability to group the data by `Month` and `Year` for each `Symbol` variable.

This data consists of four categorical variables (`Date`, `Symbol`, `Year`, and `Month`) and one continuous variable, `AdjClose`. In pandas, it is possible to group by any single categorical variable by passing its name to `.groupby()`. The following command groups by the `Symbol` column:

```
In [37]:
s4g.groupby('Symbol')

Out[37]:
<pandas.core.groupby.DataFrameGroupBy object at 0x7ffaeeb49a10>
```

The result of calling `.groupby()` on a `DataFrame` is not the actual grouped data but a `DataFrameGroupBy` object (a `SeriesGroupBy` for a grouping on a `Series`). The grouping has not actually been performed yet as grouping is a deferred/lazy process in pandas.

This result of `.groupby()` is a subclass of a `GroupBy` object and is an interim description of the grouping to be performed (if you are a C# programmer, this feels a lot like an expression tree created by LINQ). This allows pandas to first validate that the grouping description provided to it is valid relative to the data before the processing starts.

There are number of useful properties on a `GroupBy` object. The `.groups` property will return a Python dictionary whose keys represent the name of each group (if multiple columns are specified, it is a tuple), and the values are an array of the index labels contained within each group:

```
In [38]:
   grouped = s4g.groupby('Symbol')
   type(grouped.groups)

Out[38]:
   dict

In [39]:
   grouped.groups

Out[39]:
   {u'AAPL': [1, 3, 5, 7, 9, 11, 13, 14, 16, 18, 20, 23, 25, 27, 29,
   30, 33, 34, 37, 38, 41, 43, 45, 46, 48, 50, 53, 54, 56, 58, 61,
   63, 64, 67, 69, 71, 72, 75, 77, 79, 81, 82, 84, 89, 91, 92, 94,
   ...
   452, 455, 456, 458, 460, 463, 464, 466, 468, 471, 472, 474, 476,
   478, 480, 482, 484, 487, 488, 490, 492, 494, 497]
   'MSFT': [0, 2, 4, 6, 10, 12, 15, 17, 19, 21, 22, 24, 26, 28, 31, 32,
   ...

   477, 479, 481, 483, 485, 486, 489, 491, 493, 495, 496]}
```

The Python `len()` function can be used to return the number of groups, which will result from the grouping as well as the `.ngroups` property:

```
In [40]:
   len(grouped), grouped.ngroups

Out[40]:
   (2, 2)
```

Splitting is not performed until you take some type of action on the `GroupBy` object. It is, however, possible to iterate over several properties of the object to view how the data will be grouped (hence forcing it to be grouped). The following `helper` function demonstrates this and will be used frequently throughout this chapter:

```
In [41]:
   def print_groups (groupobject):
       for name, group in groupobject:
           print name
           print group.head()


In [42]:
   print_groups(grouped)


Out[42]:
   AAPL
           Date  Year  Month Symbol  Adj Close
   1 2012-01-03  2012      1   AAPL      55.41
   3 2012-01-04  2012      1   AAPL      55.71
   5 2012-01-05  2012      1   AAPL      56.33
   7 2012-01-06  2012      1   AAPL      56.92
   9 2012-01-09  2012      1   AAPL      56.83
   MSFT
           Date  Year  Month Symbol  Adj Close
   0 2012-01-03  2012      1   MSFT      24.42
   2 2012-01-04  2012      1   MSFT      25.00
   4 2012-01-05  2012      1   MSFT      25.25
   6 2012-01-06  2012      1   MSFT      25.64
   8 2012-01-09  2012      1   MSFT      25.31
```

Looking at these results gives us some insight into what pandas is doing with this specific splitting operation. It has created, for each distinct value in the `Symbol` column of the original `DataFrame`, a group consisting of a `DataFrame` (this is different from the functionality provided by `itertools.groupby`, so be careful if you are used to using that library for this functionality). It then copies the non-grouped columns and data into each of those `DataFrame` objects and then uses the values from the specified column(s) as the group name.

The `.size()` method of the object gives a nice summary of the size of all the groups:

```
In [43]:
   grouped.size()


Out[43]:
   Symbol
   AAPL      249
   MSFT      249
   dtype: int64
```

If you want the data for the items in any given group, you can use the `.get_group()` property. The following command retrieves the MSFT group:

```
In [44]:
   grouped.get_group('MSFT')


Out[44]:
              Date  Year  Month Symbol  Adj Close
   0    2012-01-03  2012      1   MSFT      24.42
   2    2012-01-04  2012      1   MSFT      25.00
   4    2012-01-05  2012      1   MSFT      25.25
   6    2012-01-06  2012      1   MSFT      25.64
   ..          ...   ...    ...    ...        ...
   491  2012-12-24  2012     12   MSFT      25.38
   493  2012-12-26  2012     12   MSFT      25.20
   495  2012-12-27  2012     12   MSFT      25.29
   496  2012-12-28  2012     12   MSFT      24.91

   [249 rows x 5 columns]
```

Grouping can be performed upon multiple columns by passing a list of column names. The following command groups the data by the `Symbol` and `Year` and `Month` variables:

```
In [45]:
   mcg = s4g.groupby(['Symbol', 'Year', 'Month'])
   print_groups(mcg)
```

```
Out[45]:
   ('AAPL', 2012, 1)
           Date  Year  Month Symbol  Adj Close
   1 2012-01-03  2012      1   AAPL      55.41
   3 2012-01-04  2012      1   AAPL      55.71
   5 2012-01-05  2012      1   AAPL      56.33
   7 2012-01-06  2012      1   AAPL      56.92
   9 2012-01-09  2012      1   AAPL      56.83
   ('AAPL', 2012, 2)
            Date  Year  Month Symbol  Adj Close
   41 2012-02-01  2012      2   AAPL      61.47
   43 2012-02-02  2012      2   AAPL      61.33
   45 2012-02-03  2012      2   AAPL      61.94
   46 2012-02-06  2012      2   AAPL      62.52
   48 2012-02-07  2012      2   AAPL      63.18
   ('AAPL', 2012, 3)
   ...
```

Since multiple columns were specified, the name of each group is now a tuple with the value from `Symbol`, `Year`, and `Month` that represents the group.

The examples up to this point have used a `DataFrame` without any specific indexing (just the default sequential numerical index). This type of data would actually be very well suited for a hierarchical index, which can then be used directly to group the data based upon index label(s). To demonstrate this, the following command creates a new `DataFrame` with a `MultiIndex` consisting of the original `Symbol`, `Year`, and `Month` columns:

```
In [46]:
   mi = s4g.set_index(['Symbol', 'Year', 'Month'])
   mi


Out[46]:
                              Date  Adj Close
   Symbol Year Month
   MSFT   2012 1     2012-01-03      24.42
   AAPL   2012 1     2012-01-03      55.41
   MSFT   2012 1     2012-01-04      25.00
   AAPL   2012 1     2012-01-04      55.71
```

```
...                          ...        ...
             12    2012-12-27      70.02
MSFT   2012 12     2012-12-27      25.29
            12    2012-12-28      24.91
AAPL   2012 12     2012-12-28      69.28

[498 rows x 2 columns]
```

Grouping can now be performed using the levels of the hierarchical index. The following groups by the index level `0` (`Symbol` names):

```
In [47]:
   mig_l1 = mi.groupby(level=0)
   print_groups(mig_l1)


Out[47]:
   AAPL
                           Date  Adj Close
   Symbol Year Month
   AAPL   2012 1      2012-01-03      55.41
               1      2012-01-04      55.71
               1      2012-01-05      56.33
               1      2012-01-06      56.92
               1      2012-01-09      56.83
   MSFT
                           Date  Adj Close
   Symbol Year Month
   MSFT   2012 1      2012-01-03      24.42
               1      2012-01-04      25.00
               1      2012-01-05      25.25
               1      2012-01-06      25.64
               1      2012-01-09      25.31
```

Grouping by multiple levels can be performed by passing multiple levels to `.groupby()`. Also, if the `MultiIndex` has names specified, then those names can be used instead of the integers for the levels. The following command groups the three levels of the `MultiIndex` by using their names:

```
In [48]:
   mig_l12 = mi.groupby(level=['Symbol', 'Year', 'Month'])
   print_groups(mig_l12)

Out[48]:
   ('AAPL', 2012, 1)
                           Date  Adj Close
   Symbol Year Month
   AAPL   2012 1     2012-01-03      55.41
               1     2012-01-04      55.71
               1     2012-01-05      56.33
               1     2012-01-06      56.92
               1     2012-01-09      56.83
   ('AAPL', 2012, 2)
   ...
   ('MSFT', 2012, 12)
                           Date  Adj Close
   Symbol Year Month
   MSFT   2012 12    2012-12-03      24.79
               12    2012-12-04      24.74
               12    2012-12-05      25.02
               12    2012-12-06      25.07
               12    2012-12-07      24.82
```

Aggregating

Armed with the capability to group the stock data on a monthly basis, we can now start to drive analysis of the data. Specifically, we will develop methods to calculate the risk on a stock based on a time-window of a calendar month.

Aggregation is performed using the `.aggregate()`, or in short `.agg()`, method of the `GroupBy` object. The parameter set to `.agg()` is a reference to a function that is applied to each group. The following command will calculate the mean of the values across each `Symbol`, `Year`, and `Month`:

```
In [49]:
   mig_l12.agg(np.mean)

Out[49]:
                        Adj Close
   Symbol Year Month
   AAPL   2012 1           57.75
               2           67.05
               3           77.82
               4           81.66
   ...                       ...
   MSFT   2012 9           28.64
               10          27.04
               11          26.00
               12          25.31

   [24 rows x 1 columns]
```

The result of the aggregation will have an identically structured index as the original data. If you do not want this to happen, you can use the `as_index=False` option of the `.groupby()` method to specify not to duplicate the structure of the index, which may be convenient in several situations, including where a function expects the data with a numerical index:

```
In [50]:
   s4g.groupby(['Symbol', 'Year', 'Month'],
               as_index=False).agg(np.mean)[:5]

Out[50]:
     Symbol  Year  Month  Adj Close
   0   AAPL  2012      1      57.75
   1   AAPL  2012      2      67.05
   2   AAPL  2012      3      77.82
   3   AAPL  2012      4      81.66
   4   AAPL  2012      5      76.09
```

This has derived the same results, but there is a slightly different organization.

It is possible to apply multiple aggregation functions to each group in a single call to `.agg()` by passing them in a list:

```
In [51]:
   mig_l12.agg([np.mean, np.std])

Out[51]:
                      Adj Close
                           mean   std
   Symbol Year Month
   AAPL   2012 1          57.75  1.80
                2          67.05  3.57
                3          77.82  4.16
                4          81.66  3.06
   ...                       ...   ...
   MSFT   2012 9          28.64  0.43
                10         27.04  0.67
                11         26.00  1.00
                12         25.31  0.36

   [24 rows x 2 columns]
```

Summary

In this chapter, we examined several fundamental techniques for loading (importing and reading data), combining, grouping, and analyzing stock pricing data with pandas. In the next chapter on time-series data with pandas, we will dive deeper into working with data provided in different time frequencies, converting the periods of data into other frequencies, and working with aggregating data based upon sliding/rolling windows instead of simple calendar months.

4
Time-series

A time-series is a sequence of data points, typically consisting of successive measurements made at a regular frequency and over a specific time interval. Time-series analysis is composed of various methods for making decisions based upon the data in a time-series by extracting meaningful statistics. Time-series forecasting is the process of developing a model based upon data in a time-series, and using it to predict future values based upon previously observed values. Regression analysis is the process of testing whether one or more independent time-series affect the current value of another time-series.

There is extensive support for working with time-series data in pandas. In this chapter, we will examine representing time-series data with the pandas `Series` and `DataFrame` as well as several common techniques for manipulating this data. The techniques learned in this chapter will set the basis for the remaining chapters, where we will examine several financial processes using time-series data, including a historical analysis of stock performance, correlating multiple streams of financial and social data to develop trading strategies, optimize portfolio allocation, and calculate risk based upon historical data.

In this chapter, we will cover the following:

- `DatetimeIndex` and its use in time-series data
- Creating time-series with specific frequencies
- Calculation of new dates using date offsets
- Representation of intervals of time user periods
- Shifting and lagging time-series data
- Frequency conversion of time-series data
- Upsampling and downsampling of time-series data

There are very robust facilities for handling date and time in pandas that we are not going to cover in detail in this chapter. We will focus on just the time-series capabilities we require for the later chapters. For more extensive coverage of everything that can be done, refer to my book *Learning pandas*, *Packt Publishing* or the online pandas documentation on time-series and data functionality at `http://pandas.pydata.org/pandas-docs/stable/timeseries.html`.

Notebook setup

The examples in this chapter will utilize the following configuration of the Python environment:

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

   pd.set_option('display.notebook_repr_html', False)
   pd.set_option('display.max_columns', 10)
   pd.set_option('display.max_rows', 8)
   pd.set_option('precision', 7)

   import datetime
   from datetime import datetime

   import matplotlib.pyplot as plt
   %matplotlib inline
   pd.options.display.mpl_style = 'default'
```

Time-series data and the DatetimeIndex

Excelling at manipulating time-series data, pandas was created initially for use in finance, and from its inception, it has had facilities for managing complete date and time-series operations to handle complex financial scenarios. These capabilities have been progressively expanded and refined over all of its versions.

The representations of dates, times, and time intervals and periods provided by pandas, which are pandas's own, are above and beyond those provided in other Python frameworks such as SciPy and NumPy. The pandas implementations provide additional capabilities that are required to model time-series data, and to transform data across different frequencies, periods, and calendars for different organizations and financial markets.

Specific dates and times in pandas are represented using the pandas `Timestamp` class. `Timestamp` is based on NumPy's dtype `datetime64` and has higher precision than Python's built-in `datetime` object. This increased precision is frequently required for accurate financial calculations.

Sequences of timestamp objects are represented by pandas as a `DatetimeIndex`, which is a type of pandas index that is optimized for indexing by dates and times. There are several ways to create `DatetimeIndex` objects in pandas. The following command creates a `DatetimeIndex` from an array of `datetime` objects:

```
In [2]:
   dates = [datetime(2014, 8, 1), datetime(2014, 8, 2)]
   dti = pd.DatetimeIndex(dates)
   dti


Out[2]:
   <class 'pandas.tseries.index.DatetimeIndex'>
    [2014-08-01, 2014-08-02]
   Length: 2, Freq: None, Timezone: None
```

A `Series` will also automatically construct a `DatetimeIndex` as its index when passing a list of datetime objects as the `index` parameter:

```
In [3]:
   np.random.seed(123456)
   ts = pd.Series(np.random.randn(2), dates)
   type(ts.index)


Out[3]:
   pandas.tseries.index.DatetimeIndex
```

The `Series` object has taken the `datetime` objects and constructed a `DatetimeIndex` from the date values, where each value of the `DatetimeIndex` is a `Timestamp` object, and each element of the index can be used to access the corresponding value in the `Series` object. To demonstrate this, the following command shows several ways in which we can access the value in the `Series` with the date `2014-08-02` as an index label:

```
In [4]:
   ts[datetime(2014, 8, 2)]


Out[4]:
   -0.28286334432866328


In [5:
   ts['2014-8-2']


Out[5]:
   -0.28286334432866328
```

The `Series` object can also create a `DatetimeIndex` when passing a list of strings, which pandas will gladly recognize as dates and perform the appropriate conversions:

```
In [6]:
   np.random.seed(123456)
   dates = ['2014-08-01', '2014-08-02']
   ts = pd.Series(np.random.randn(2), dates)
   ts


Out[6]:
   2014-08-01    0.469112
   2014-08-02   -0.282863
   dtype: float64
```

Also provided by pandas is the `pd.to_datetime()` function, which is used to perform a conversion of a list of potentially mixed type items into a `DatetimeIndex`:

```
In [7]:
   dti = pd.to_datetime(['Aug 1, 2014', '2014-08-02',
                         '2014.8.3', None])
   dti

Out[7]:
   <class 'pandas.tseries.index.DatetimeIndex'>
    [2014-08-01, ..., NaT]
   Length: 4, Freq: None, Timezone: None
```

Notice that `None` is converted into a not-a-time value, `NaT`, which represents that the source data could not be converted into `datetime`.

But be careful as, by default, the `pd.to_datetime()` function will fall back to returning a NumPy array of objects if it cannot parse a value, as demonstrated here:

```
In [8]:
   dti2 = pd.to_datetime(['Aug 1, 2014', 'foo'])
   type(dti2)

Out[8]:
   numpy.ndarray
```

To force the function to convert to dates and `DatetimeIndex`, you can use the `coerce=True` parameter, as shown here:

```
In [9]:
   pd.to_datetime(['Aug 1, 2014', 'foo'], coerce=True)

Out[9]:
   <class 'pandas.tseries.index.DatetimeIndex'>
   [2014-08-01, NaT]
   Length: 2, Freq: None, Timezone: None
```

The pandas default is that date strings are always month first. If you need to parse dates with the day as the first component, you can use the `dayfirst=True` option, which can be useful as data can often have day first, particularly when it is non-U.S. data. The following command demonstrates this in action and also shows how the ordering can be changed:

```
In [10]:
   dti1 = pd.to_datetime(['8/1/2014'])
   dti2 = pd.to_datetime(['1/8/2014'], dayfirst=True)
   dti1[0], dti2[0]


Out[10]:
    (Timestamp('2014-08-01 00:00:00'),
    Timestamp('2014-08-01 00:00:00'))
```

A range of timestamps at a specific frequency can easily be created using the `pd.date_range()` function. The following command creates a `Series` from a `DatetimeIndex` of `10` consecutive days:

```
In [11]:
   np.random.seed(123456)
   dates = pd.date_range('8/1/2014', periods=10)
   s1 = pd.Series(np.random.randn(10), dates)
   s1[:5]


Out[11]:
   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
   Freq: D, dtype: float64
```

Like any pandas index, a `DatetimeIndex` can be used for various index operations, such as data alignment, selection, and slicing. To demonstrate slicing using a `DatetimeIndex`, we will refer to the Yahoo! Finance stock quotes for MSFT from 2012 through 2014 using the pandas `DataReader` class (more info on `DataReader` is available at `http://pandas.pydata.org/pandas-docs/version/0.15.2/remote_data.html`):

```
In [12]:
   import pandas.io.data as web
   msft = web.DataReader("MSFT", 'yahoo', '2012-1-1', '2013-12-30')
   msft.head()

Out[12]:
                Open   High    Low  Close    Volume  Adj Close
   Date
   2012-01-03  26.55  26.96  26.39  26.77  64731500   24.42183
   2012-01-04  26.82  27.47  26.78  27.40  80516100   24.99657
   2012-01-05  27.38  27.73  27.29  27.68  56081400   25.25201
   2012-01-06  27.53  28.19  27.53  28.11  99455500   25.64429
   2012-01-09  28.05  28.10  27.72  27.74  59706800   25.30675
```

The `msft` variable is a `DataFrame` that represents a time-series of multiple data points (`Open`, `High`, `Low`, and so on) for the MSFT stock. To make these examples easier, from this `DataFrame`, we can create a pandas `Series` consisting of just the `Adj Close` values:

```
In [13]:
   msftAC = msft['Adj Close']
   msftAC.head(3)

Out[13]:
   Date
   2012-01-03    24.42183
   2012-01-04    24.99657
   2012-01-05    25.25201
   Name: Adj Close, dtype: float64
```

The `msftAC` variable is a pandas `Series` object. I point this out as several of the operations to retrieve values from `Series` objects differ, depending upon whether the operation is being applied to a `Series` or a `DataFrame`. This can cause some slight confusion if this is not recognized.

The slicing notation is overridden to very conveniently allow the passing of strings representing dates as the values for the slice. The following command retrieves MSFT data for dates from `2012-01-01` to `2012-01-05`:

```
In [14]:
   msft['2012-01-01':'2012-01-05']


Out[14]:
                Open   High    Low  Close    Volume  Adj Close
   Date
   2012-01-03  26.55  26.96  26.39  26.77  64731500   24.42183
   2012-01-04  26.82  27.47  26.78  27.40  80516100   24.99657
   2012-01-05  27.38  27.73  27.29  27.68  56081400   25.25201
```

A specific item can be retrieved from a time-series represented by a `DataFrame` by specifying the date/time index value and using the `.loc` method. The result is a `Series` where the index labels are the column names, with the values for each being in a specific row for each of the columns:

```
In [15]:
   msft.loc['2012-01-03']


Out[15]:
   Open               26.55000
   High               26.96000
   Low                26.39000
   Close              26.77000
   Volume       64731500.00000
   Adj Close          24.42183
   Name: 2012-01-03 00:00:00, dtype: float64
```

Note that the following syntax does not work as the `DataFrame` attempts to look for a column with the name `2012-01-03`:

```
In [16]:
   # msft['2012-01-03'] # commented to prevent killing the notebook
```

This syntax does work on a `Series` object that is a time-series, and this looks for an index label with the matching date:

```
In [17]:
   msftAC['2012-01-03']


Out[17]:
   24.42183
```

This is a subtle difference that sometimes causes headaches when using time-series data in pandas. So be careful or always convert your `Series` objects to `DataFrame` objects to use a lookup, using `.loc` to lookup using the index.

One of the advantages of pandas is the ability to be able to select based upon partial `datetime` specifications. As an example, the following command selects MSFT data for the month of February 2012:

```
In [18]:
   msft['2012-02'].head(5)


Out[18]:
                Open   High    Low  Close    Volume  Adj Close
   Date
   2012-02-01  29.79  30.05  29.76  29.89  67409900   27.26815
   2012-02-02  29.90  30.17  29.71  29.95  52223300   27.32289
   2012-02-03  30.14  30.40  30.09  30.24  41838500   27.58745
   2012-02-06  30.04  30.22  29.97  30.20  28039700   27.55096
   2012-02-07  30.15  30.49  30.05  30.35  39242400   27.68781
```

Note that this did not require the use of the `.loc` method, as pandas first identifies this as a partial date and then looks along the index of the `DataFrame` instead of a column (although `.loc` can be used to perform an equivalent operation).

It is also possible to slice, starting at the beginning of a specific month and ending at a specific day of the month:

```
In [19]:
   msft['2012-02':'2012-02-09'][:5]


Out[19]:
                Open   High    Low  Close    Volume  Adj Close
   Date
   2012-02-01  29.79  30.05  29.76  29.89  67409900   27.26815
   2012-02-02  29.90  30.17  29.71  29.95  52223300   27.32289
   2012-02-03  30.14  30.40  30.09  30.24  41838500   27.58745
   2012-02-06  30.04  30.22  29.97  30.20  28039700   27.55096
   2012-02-07  30.15  30.49  30.05  30.35  39242400   27.68781
```

Creating time-series with specific frequencies

Time-series data in pandas can also be created to represent intervals of time 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 introduce the creation of nondaily frequencies, the following command creates a `DatetimeIndex` with one-minute intervals using `freq='T'`:

```
In [20]:
   bymin = pd.Series(np.arange(0, 90*60*24),
                     pd.date_range('2014-08-01',
                                   '2014-10-29 23:59:00',
                                   freq='T'))
   bymin


Out[20]:
   2014-08-01 00:00:00         0
   2014-08-01 00:01:00         1
   2014-08-01 00:02:00         2
                              ...
```

```
   2014-10-29 23:57:00    129597
   2014-10-29 23:58:00    129598
   2014-10-29 23:59:00    129599
   Freq: T, dtype: int64
```

This time-series allows us to use forms of slicing at finer resolution. Earlier, we saw slicing at day and month levels, but now we have a time-series with minute-based data that we can slice down to hours and minutes (and smaller intervals if we use finer frequencies):

```
In [21]:
   bymin['2014-08-01 12:30':'2014-08-01 12:59']


Out[21]:
   2014-08-01 12:30:00    750
   2014-08-01 12:31:00    751
   2014-08-01 12:32:00    752
                         ...
   2014-08-01 12:57:00    777
   2014-08-01 12:58:00    778
   2014-08-01 12:59:00    779
   Freq: T, dtype: int64
```

A complete list can be found at `http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases`.

Representing intervals of time using periods

It is often required to represent not just a specific time or sequence of timestamps, but to represent an interval of time using a start date and an end date (an example of this would be a financial quarter). This representation of a bounded interval of time can be represented in pandas using `Period` objects.

`Period` objects consist of a start time and an end time and are created from a start date with a given frequency. The start time is referred to as the anchor of the `Period` object, and the end time is then calculated from the start date and the period specification.

To demonstrate this, the following command creates a period representing a 1-month period anchored in August 2014:

```
In [22]:
   aug2014 = pd.Period('2014-08', freq='M')
   aug2014


Out[22]:
   Period('2014-08', 'M')
```

The `Period` function has `start_time` and `end_time` properties that inform us of the derived start and end times of `Period`:

```
In [23]:
   aug2014.start_time, aug2014.end_time


Out[23]:
   (Timestamp('2014-08-01 00:00:00'),
    Timestamp('2014-08-31 23:59:59.999999999'))
```

Since we specified a period that starts using a partial date specification of August 2014, pandas determines the anchor (`start_time`) as `2014-08-01 00:00:00` and then calculates the `end_time` property based upon the specified frequency; in this case, calculating 1 month from the `start_time` anchor and returning the last unit of time prior to this.

Mathematical operations are overloaded on `Period` objects, so as to calculate another period based upon the value represented in `Period`. As an example, the following command creates a new `Period` based upon the `aug2014` period object by adding `1` to the period. Since `aug2014` has a period of 1 month, the resulting value is that start date (`2014-08-01`) + `1 * 1` month (the period represented by the object), and, hence, the result is the last moment of time prior to `2014-09-01`:

```
In [24]:
   sep2014 = aug2014 + 1
   sep2014


Out[24]:
   Period('2014-09', 'M')
```

This seems as though pandas has simply added one to the month in the `Period` object `aug2014`. However, examining `start_time` and `end_time` of `sep2014`, we can see something interesting:

```
In [25]:
   sep2014.start_time, sep2014.end_time

Out[25]:
   (Timestamp('2014-09-01 00:00:00'),
    Timestamp('2014-09-30 23:59:59.999999999'))
```

The `Period` object has the ability to know that September has 30 days and not 31. This is the advantage that the `Period` object has over simple addition. It is not simply adding 30 days (in this example), but one unit of the period. This helps to solve many difficult date management problems.

`Period` objects are useful when combined into a collection referred to as a `PeriodIndex`. The following command creates a pandas `PeriodIndex` consisting of 1-month intervals for the year of 2013:

```
In [26]:
   mp2013 = pd.period_range('1/1/2013', '12/31/2013', freq='M')
   mp2013

Out[26]:
   <class 'pandas.tseries.period.PeriodIndex'>
    [2013-01, ..., 2013-12]
   Length: 12, Freq: M
```

A `PeriodIndex` differs from a `DatetimeIndex` in that in a `PeriodIndex`, the index labels are `Period` objects:

```
In [27]:
   for p in mp2013:
       print "{0} {1} {2} {3}".format(p, p.freq,
                                      p.start_time, p.end_time)
```

```
Out[27]:
   2013-01 M 2013-01-01 00:00:00 2013-01-31 23:59:59.999999999
   2013-02 M 2013-02-01 00:00:00 2013-02-28 23:59:59.999999999
   2013-03 M 2013-03-01 00:00:00 2013-03-31 23:59:59.999999999
   2013-04 M 2013-04-01 00:00:00 2013-04-30 23:59:59.999999999
   2013-05 M 2013-05-01 00:00:00 2013-05-31 23:59:59.999999999
   2013-06 M 2013-06-01 00:00:00 2013-06-30 23:59:59.999999999
   2013-07 M 2013-07-01 00:00:00 2013-07-31 23:59:59.999999999
   2013-08 M 2013-08-01 00:00:00 2013-08-31 23:59:59.999999999
   2013-09 M 2013-09-01 00:00:00 2013-09-30 23:59:59.999999999
   2013-10 M 2013-10-01 00:00:00 2013-10-31 23:59:59.999999999
   2013-11 M 2013-11-01 00:00:00 2013-11-30 23:59:59.999999999
   2013-12 M 2013-12-01 00:00:00 2013-12-31 23:59:59.999999999
```

With a `PeriodIndex`, we can then construct a `Series` using it as the index:

```
In [28]:
   np.random.seed(123456)
   ps = pd.Series(np.random.randn(12), mp2013)
   ps

Out[28]:
   2013-01    0.469112
   2013-02   -0.282863
   2013-03   -1.509059
                ...
   2013-10   -2.104569
   2013-11   -0.494929
   2013-12    1.071804
   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, such as the average value of a security in a given month, instead of at a specific time. This becomes very useful when we perform resampling of the time-series to another frequency, which we will do a little later in this chapter.

Shifting and lagging time-series data

A common operation on time-series data is to shift or "lag" the values back and forward in time, such as to calculate percentage change from sample to sample. The pandas method for this is `.shift()`, which will shift the values in the index by a specified number of units of the index's period.

To demonstrate shifting and lagging, we will use the adjusted close values for MSFT. As a refresher, the following command shows the first 10 items in that time-series:

```
In [29]:
   msftAC[:5]


Out[29]:
   Date
   2012-01-03    24.42183
   2012-01-04    24.99657
   2012-01-05    25.25201
   2012-01-06    25.64429
   2012-01-09    25.30675
   Name: Adj Close, dtype: float64
```

The following command shifts the adjusted closing prices forward by 1 day:

```
In [30]:
   shifted_forward = msftAC.shift(1)
   shifted_forward[:5]


Out[30]:
   Date
   2012-01-03         NaN
   2012-01-04    24.42183
   2012-01-05    24.99657
   2012-01-06    25.25201
   2012-01-09    25.64429
   Name: Adj Close, dtype: float64
```

Notice that the value of the index label of `2012-01-03` is now `NaN`. When shifting at the same frequency as that of the index, the shift will result in one or more NaN values being added for the labels at one end of the `Series`, and a loss of the same number of values at the other end. The amount of NaN values is the same as the number of specified periods.

If we examine the tail of both the original and shifted `Series`, we will see that the last value in the `Series` was shifted away:

```
In [31]:
   msftAC.tail(5), shifted_forward.tail(5)


Out[31]:
    (Date
    2013-12-23    35.39210
    2013-12-24    35.83668
    2013-12-26    36.18461
    2013-12-27    36.03964
    2013-12-30    36.03964
    Name: Adj Close, dtype: float64, Date
    2013-12-23    35.56607
    2013-12-24    35.39210
    2013-12-26    35.83668
    2013-12-27    36.18461
    2013-12-30    36.03964
    Name: Adj Close, dtype: float64)
```

The original `Series` ended with two values of `36.04` for both `2013-12-27` and `2013-12-30`, and the value that was originally at `2013-12-30` is now lost.

It is also possible to shift values in the opposite direction. The following command demonstrates this by shifting the `Series` by `-2`:

```
In [32]:
   shifted_backwards = msftAC.shift(-2)[:10]
   shifted_backwards[:5]
```

```
Out[32]:
   Date
   2012-01-03    25.25201
   2012-01-04    25.64429
   2012-01-05    25.30675
   2012-01-06    25.39797
   2012-01-09    25.28850
   Name: Adj Close, dtype: float64
```

This results in two NaN values at the tail of the resulting `Series`:

```
In [33]:
   shifted_backwards.tail(5)


Out[34]:
   Date
   2013-12-23    36.18461
   2013-12-24    36.03964
   2013-12-26    36.03964
   2013-12-27         NaN
   2013-12-30         NaN
   Name: Adj Close, dtype: float64
```

It is possible to shift by different frequencies using the `freq` parameter. This will create a time-series with a new index, where the index labels are adjusted by the number of specified units of the given frequency. As an example, the following command shifts forward the time-series with a frequency of `1` day by one second:

```
In [35]:
   msftAC.shift(1, freq="S")


Out[35]:
   Date
   2012-01-03 00:00:01    24.42183
   2012-01-04 00:00:01    24.99657
   2012-01-05 00:00:01    25.25201
                             ...
```

```
   2013-12-26 00:00:01    36.18461
   2013-12-27 00:00:01    36.03964
   2013-12-30 00:00:01    36.03964
   Name: Adj Close, dtype: float64
```

The resulting `DataFrame` or `Series` is essentially the same as the original, with the specified number of units of frequency added to each index label. No data will be shifted out or replaced with NaN as this is not performing realignment.

An alternate form of shifting is provided by pandas using the `.tshift()` method. Rather than changing the alignment of the data, `.tshift()` simply results in a new `Series` or `DataFrame`, where the values of the index labels are changed by the specified number of offsets of the value of the `freq` parameter. This is demonstrated by the following command, which modifies the index labels by 1 day:

```
In [37]:
   msftAC.tshift(1, freq="D")


Out[37]:
   Date
   2012-01-04    24.42183
   2012-01-05    24.99657
   2012-01-06    25.25201
                   ...
   2013-12-27    36.18461
   2013-12-28    36.03964
   2013-12-31    36.03964
   Name: Adj Close, dtype: float64
```

A practical application of shifting is the calculation of daily percentage changes from the previous day. The following command calculates the day-to-day percentage change in the adjusted closing price for MSFT:

```
In [38]:
   msftAC / msftAC.shift(1) - 1


Out[38]:
   Date
   2012-01-03         NaN
   2012-01-04    0.023534
```

```
   2012-01-05    0.010219
                   ...
   2013-12-26    0.009709
   2013-12-27   -0.004006
   2013-12-30    0.000000
   Name: Adj Close, dtype: float64
```

Frequency conversion of time-series data

The frequency of the data in a time-series can be converted in pandas using the `.asfreq()` method of a `Series` or `DataFrame`. To demonstrate, we will use the following small subset of the MSFT stock closing values:

```
In [39]:
   sample = msftAC[:2]
   sample


Out[39]:
   Date
   2012-01-03    24.42183
   2012-01-04    24.99657
   Name: Adj Close, dtype: float64
```

We have extracted the first 2 days of adjusted close values. Let's suppose we want to resample this to have hourly sampling of data in-between the index labels. We can do this with the following command:

```
In [40]:
   sample.asfreq("H")


Out[40]:
   2012-01-03 00:00:00    24.42183
   2012-01-03 01:00:00         NaN
   2012-01-03 02:00:00         NaN
                             ...
   2012-01-03 22:00:00         NaN
   2012-01-03 23:00:00         NaN
   2012-01-04 00:00:00    24.99657
   Freq: H, Name: Adj Close, dtype: float64
```

A new index with hourly index labels has been created by pandas, but when aligning to the original time-series, only two values were found, thereby leaving the others filled with NaN.

We can change this default behavior using the `method` parameter of the `.asfreq()` method. One method is `pad` or `ffill` that will fill with the last known value:

```
In [41]:
   sample.asfreq("H", method="ffill")


Out[41]:
   2012-01-03 00:00:00    24.42183
   2012-01-03 01:00:00    24.42183
   2012-01-03 02:00:00    24.42183
                             ...
   2012-01-03 22:00:00    24.42183
   2012-01-03 23:00:00    24.42183
   2012-01-04 00:00:00    24.99657
   Freq: H, Name: Adj Close, dtype: float64
```

The other method is to use `backfill/bfill`, which will use the next known value:

```
In [42]:
   sample.asfreq("H", method="bfill")


Out[42]:
   2012-01-03 00:00:00    24.42183
   2012-01-03 01:00:00    24.99657
   2012-01-03 02:00:00    24.99657
                             ...
   2012-01-03 22:00:00    24.99657
   2012-01-03 23:00:00    24.99657
   2012-01-04 00:00:00    24.99657
   Freq: H, Name: Adj Close, dtype: float64
```

Resampling of time-series

Frequency conversion provides basic conversion of data using the new frequency intervals and allows the filling of missing data using either NaN, forward filling, or backward filling. More elaborate control is provided through the process of resampling.

Resampling can be either downsampling, where data is converted to wider frequency ranges (such as downsampling from day-to-day to month-to-month) or upsampling, where data is converted to narrower time ranges. Data for the associated labels are then calculated by a function provided to pandas instead of simple filling.

To demonstrate upsampling, we will calculate the daily cumulative returns for the MSFT stock over 2012 and 2013 and resample it to monthly frequency. We will examine the return calculation in more detail in *Chapter 5*, *Time-series Stock Data*, but for now, we will use it as a demonstration of the mechanics of up and down resampling of time-series data.

The cumulative daily return for MSFT can be calculated with the following command using `.shift()` and application of the `.cumprod()` method, as shown here:

```
In [43]:
   msft_cum_ret = (1 + (msftAC / msftAC.shift() - 1)).cumprod()
   msft_cum_ret

Out[43]:
   Date
   2012-01-03         NaN
   2012-01-04    1.023534
   2012-01-05    1.033993
                    ...
   2013-12-26    1.481650
   2013-12-27    1.475714
   2013-12-30    1.475714
   Name: Adj Close, dtype: float64
```

A time-series can be resampled using the `.resample()` method. This method provides a very flexible means to specify the frequency conversion involved in the resampling, as well as the means by which the resampled values are selected or calculated.

The following command downsamples the daily cumulative returns from day-to-day to month-to-month:

```
In [44]:
   msft_monthly_cum_ret = msft_cum_ret.resample("M")
   msft_monthly_cum_ret


Out[44]:
   Date
   2012-01-31    1.068675
   2012-02-29    1.155697
   2012-03-31    1.210570
                   ...
   2013-10-31    1.350398
   2013-11-30    1.471915
   2013-12-31    1.482362
   Freq: M, Name: Adj Close, dtype: float64
```

As the resample period is specified as monthly, pandas will break the index labels into monthly intervals bounded on calendar months, and the new index label for a group will be the month's end date. The value for each index entry will be the mean of the values for the month. This can be verified for January 2012 with the following command:

```
In [45]:
   msft_cum_ret['2012-01'].mean()


Out[45]:
   1.0687314108366739
```

The means by which the value for each index label is calculated can be controlled using the `how` parameter. Any function that is available via dispatching can be used and given to the `how` parameter by name. The default is to use the `np.mean()` function, as we can see in the following example:

```
In [46]:
   ms6t_cum_ret.resample("M", how="mean")
```

```
Out[46]:
   Date
   2012-01-31    1.068731
   2012-02-29    1.155794
   2012-03-31    1.210669
   ...
   2013-10-31    1.350497
   2013-11-30    1.472052
   2013-12-31    1.482453
   Freq: M, Name: Adj Close, Length: 24
```

We can use `how="ohlc"`, which will give us a summary of the `open`, `high`, `low`, and `close` values during each sampling period. For each resampling period (monthly in this example), pandas will return the first value in the period (`open`), the maximum value (`high`), the lowest value (`low`), and the final value in the period (`close`):

```
In [47]:
   msft_cum_ret.resample("M", how="ohlc")[:5]


Out[47]:
                  open      high       low     close
   Date
   2012-01-31  1.023751  1.110565  1.023751  1.103194
   2012-02-29  1.116708  1.198608  1.116708  1.193694
   2012-03-31  1.214169  1.235463  1.186732  1.212940
   2012-04-30  1.214169  1.219083  1.141278  1.203931
   2012-05-31  1.203522  1.203522  1.099918  1.104832
```

The type of index resulting from a resampling is controlled by the `kind` parameter, which can be set to timestamp (the default) or period. In the resampling examples up to this point, the resample has returned `Timestamp` and, in particular, returned the last day of the month. The following command demonstrates returning an index based on periods instead of time stamps, which can be quite useful if we need to have the start and end timestamps for each sample:

```
In [48]:
   by_periods = msft_cum_ret .resample("M",
                                                  how="mean",
```

```
                                                    kind="period")
   for i in by_periods.index[:5]:
       print ("{0}:{1} {2}".format(i.start_time, i.end_time,
                                   by_periods[i]))
   2012-01-01 00:00:00:2012-01-31 23:59:59.99999999 1.06873141084
   2012-02-01 00:00:00:2012-02-29 23:59:59.99999999 1.15579443079
   2012-03-01 00:00:00:2012-03-31 23:59:59.99999999 1.21066934703
   2012-04-01 00:00:00:2012-04-30 23:59:59.99999999 1.18474610975
   2012-05-01 00:00:00:2012-05-31 23:59:59.99999999 1.14058893604
```

To demonstrate upsampling, we will examine the process using the second and third days of MSFT's adjusted close values:

```
In [49]:
   sample = msft_cum_ret[1:3]
   sample

Out[49]:
   Date
   2012-01-04    1.023751
   2012-01-05    1.033989
   Name: Adj Close, dtype: float64
```

Our upsample example will have to resample this data to an hourly interval:

```
In [50]:
   by_hour = sample.resample("H")
   by_hour

Out[50]:
   Date
   2012-01-04 00:00:00    1.023751
   2012-01-04 01:00:00         NaN
   2012-01-04 02:00:00         NaN
   ...
   2012-01-04 22:00:00         NaN
   2012-01-04 23:00:00         NaN
   2012-01-05 00:00:00    1.033989
   Freq: H, Name: Adj Close, Length: 25
```

Hourly index labels have been created by pandas, but the alignment only propagates two values into the new time-series and fills the others with NaN. This is an inherent issue with upsampling as in the result there is missing information. By default, pandas uses NaN but provide other methods to fill in values.

As with frequency conversion, the new index labels can be forward filled or back filled using the `fill_method` parameter and specifying `bfill` or `ffill`. Another option is to interpolate the missing data, which can be done using the time-series object's `.interpolate()` method, which will perform a linear interpolation:

```
In [51]:
   by_hour.interpolate()


Out[51]:
   Date
   2012-01-04 00:00:00    1.023751
   2012-01-04 01:00:00    1.024178
   2012-01-04 02:00:00    1.024604
   ...
   2012-01-04 22:00:00    1.033135
   2012-01-04 23:00:00    1.033562
   2012-01-05 00:00:00    1.033989
   Freq: H, Name: Adj Close, Length: 25
```

Summary

In this chapter, we examined the many ways in pandas to represent various units of time and time-series data. Understanding date and time-series as well as frequency conversion is critical to analyzing financial information. We examined several ways of manipulating time-series data represented by stock price information, working with dates, times, time zones, and calendars. In closing, the chapter examined the means of converting the data in time-series into different frequencies.

In the next chapter, we will dive deeper into an analysis of historical stock data using time-series in pandas, greatly expanding our knowledge of both pandas and using it to analyze financial data.

5
Time-series Stock Data

In the previous chapter, we looked at time-series operations with pandas. The focus of the chapter was on the mechanics of time-series albeit with examples drawn from finance using historical stock data. In this chapter, we will continue to examine historical stock data, focusing on performing common financial analyses upon this data. At this point in the book, pandas moves to become a tool to facilitate analysis instead of being the story itself.

We will first look at gathering historical stock and index data from web sources and how to organize it to easily perform the various analyses we will undertake. We will then move on to demonstrating common visualizations for these types of data. These visualizations are used extensively, and they will help you gain a quick and intuitive understanding of patterns hidden in the data. Finally, we will dive into several common financial analyses performed on historical stock quotes, explaining how to use pandas to perform these operations. The focus will be on the analysis of historical data as we will revisit this data in later chapters on trading algorithms, which are used to predict future values.

Specifically, in this chapter, we will progress through the following tasks:

- Fetching and organizing stock and index data from Yahoo!
- Plotting closing prices, volumes, combined prices and volumes, and candlestick charts
- Calculating simple daily percentage change and cumulative return
- Resampling daily data to a monthly period and calculating simple monthly percentage change and cumulative return
- Analyzing the distribution of returns using histograms, Q-Q plots, and boxplots
- Determining the correlation of daily returns across multiple stocks and market indexes, including creating heatmaps and scatter plots of correlations

- Calculating and visualizing correlation
- Calculating and visualizing risk relative to expected returns
- Determining the rolling correlation of returns
- Computating least-squares regression of returns
- Analyzing the performance of stocks relative to the S&P 500 index

Notebook setup

The workbook and examples will all require the following code to execute and format output. It is similar to the previous chapters but also includes matplotlib imports to support many of the graphics that will be created and some modifications to fit data to the page in the text:

```
In [1]:
   import pandas as pd
   import pandas.io.data
   import numpy as np
   import datetime
   import matplotlib.pyplot as plt
   %matplotlib inline

   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', 78)
   pd.set_option('precision', 3)
```

Obtaining historical stock and index data

The examples will use two sets of data obtained from Yahoo! Finance. The first one is a series of stock values for several stocks over the calendar years 2012–2014. The second set of data is the S&P 500 average over the same period. Note that although we are using data from a fixed period in time, the adjusted close values tend to change slightly over time, so there may be slight differences in output when you run the code as compared to what is in the text.

Fetching historical stock data from Yahoo!

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 three different sectors: technology, airlines, and soft drinks. The purpose of this is to demonstrate deriving 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.

Historical stock quotes can be retrieved from Yahoo! using the `DataReader` class. The following command will obtain the historical quotes for Microsoft (MSFT) for the entirety of 2012-2014:

```
In [2]:
   start = datetime.date(2012, 1, 1)
   end = datetime.date(2014, 12, 31)
   msft = pd.io.data.DataReader('MSFT', "yahoo", start, end)
   msft[:5]


Out[2]:
                Open   High    Low  Close    Volume  Adj Close
   Date
   2012-01-03  26.55  26.96  26.39  26.77  64731500      24.42
   2012-01-04  26.82  27.47  26.78  27.40  80516100      25.00
   2012-01-05  27.38  27.73  27.29  27.68  56081400      25.25
   2012-01-06  27.53  28.19  27.53  28.11  99455500      25.64
   2012-01-09  28.05  28.10  27.72  27.74  59706800      25.31
```

To effectively compare this data, we will want to pull the historical quotes for each stock and store them all in a single `DataFrame`, which we can use as a source for the various analyses. To facilitate this, we will start with the following function that will get the quotes for a list of stock tickers and return all the results in a single `DataFrame`, which is indexed by `Ticker` and then `Date`:

```
In [3]:
   def get(tickers, start, end):
       def data(ticker):
```

```
            return pd.io.data.DataReader(ticker, 'yahoo', start, end)
        datas = map(data, tickers)
        return pd.concat(datas, keys=tickers, names=['Ticker','Date'])
```

Using this function, we can now load the data for all of our stocks:

```
In [4]:
   tickers = ['AAPL','MSFT','GE','IBM','AA','DAL','UAL', 'PEP', 'KO']
   all_data = get(tickers, start, end)
   all_data[:5]


Out[4]:
                        Open   High    Low  Close    Volume  Adj Close
   Ticker Date
   AAPL   2012-01-03 409.40 412.50 409.00 411.23  75555200      55.41
          2012-01-04 410.00 414.68 409.28 413.44  65005500      55.71
          2012-01-05 414.95 418.55 412.67 418.03  67817400      56.33
          2012-01-06 419.77 422.75 419.22 422.40  79573200      56.92
          2012-01-09 425.50 427.75 421.35 421.73  98506100      56.83
```

Fetching index data from Yahoo!

One set of examples will demonstrate the correlation of the various stocks against the S&P 500 average. To do this, we need to retrieve this data. This can also be retrieved from Yahoo! Finance and `DataReader`, but using the `^GSPC` symbol. The following command reads this historical data and stores it in `sp500_all`:

```
In [5]:
   sp_500 = pd.io.data.DataReader("^GSPC", "yahoo", start, end)
   sp_500[:5]


Out[5]:
                   Open    High     Low    Close      Volume  Adj Close
   Date
   2012-01-03 1258.86 1284.62 1258.86  1277.06  3943710000    1277.06
   2012-01-04 1277.03 1278.73 1268.10  1277.30  3592580000    1277.30
   2012-01-05 1277.30 1283.05 1265.26  1281.06  4315950000    1281.06
   2012-01-06 1280.93 1281.84 1273.34  1277.81  3656830000    1277.81
   2012-01-09 1277.83 1281.99 1274.55  1280.70  3371600000    1280.70
```

Visualizing financial time-series data

One of the best ways to determine patterns and relationships in financial data is to create visualizations of the information. We will examine a number of common financial visualizations and how to create them before diving into the various analyses.

Plotting closing prices

The closing price of a stock can be easily plotted with matplotlib for either a single stock or multiple stocks on the same graph. We have already pulled down all the historical data for our stocks, so to visualize the closing prices, we will need to extract those values and pass and plot them with `.plot()`.

Most of the examples will focus on the adjusted closing price instead of the close price as this takes into account splits and dividends and reflects a continuous change in the value of each stock. To facilitate the use of this field, we can extract just the adjusted close value for each stock into its own pandas object.

This happens to be very simple because of the way we organized it when it was retrieved. We first extract the `Adj Close` column and then reset the index to move the dates into a column:

```
In [6]:
   # reset the index to make everything columns
   just_closing_prices = all_data[['Adj Close']].reset_index()
   just_closing_pricesp[:5]


Out[6]:
     Ticker       Date  Adj Close
   0   AAPL 2012-01-03      55.41
   1   AAPL 2012-01-04      55.71
   2   AAPL 2012-01-05      56.33
   3   AAPL 2012-01-06      56.92
   4   AAPL 2012-01-09      56.83
```

We moved the dates into a column because we now want to pivot `Date` into the index and each `Ticker` value into a column:

```
In [7]:
   daily_close_px = just_closing_prices.pivot('Date', 'Ticker',
                                              'Adj Close')
```

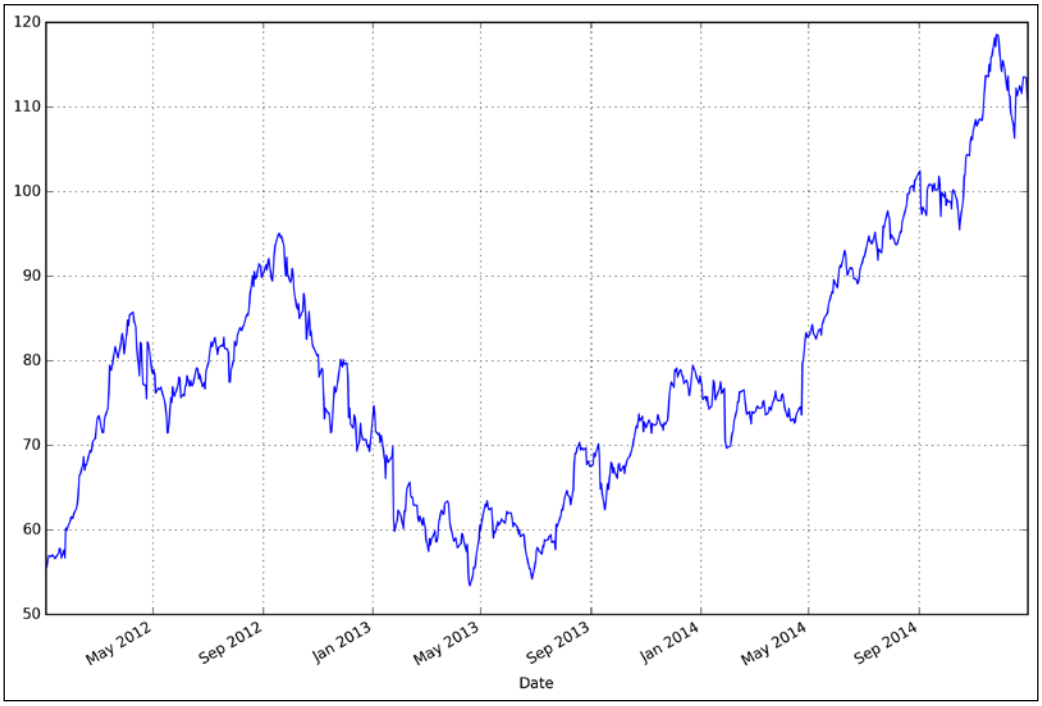
```
daily_close_px[:5]
```

```
Out[7]:
```

Ticker	AA	AAPL	DAL	...	MSFT	PEP	UAL
Date				...			
2012-01-03	8.91	55.63	7.93	...	24.60	60.85	18.90
2012-01-04	9.12	55.93	7.90	...	25.17	61.16	18.52
2012-01-05	9.03	56.55	8.22	...	25.43	60.68	18.39
2012-01-06	8.84	57.14	8.21	...	25.83	59.92	18.21
2012-01-09	9.10	57.05	8.17	...	25.49	60.24	17.93

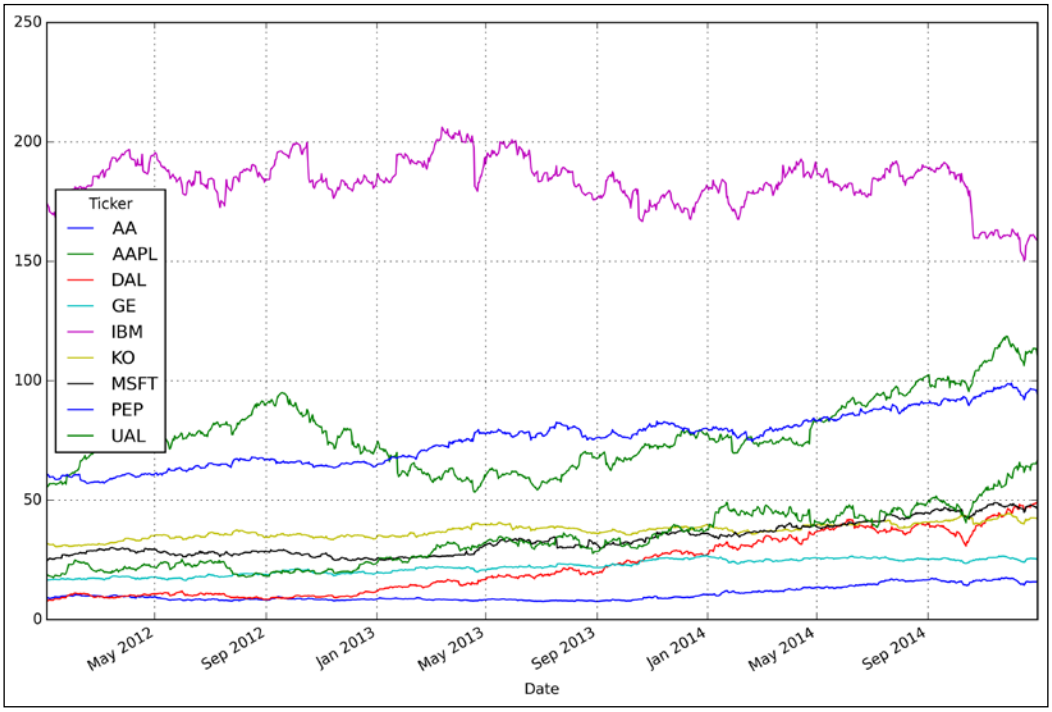
Using this `DataFrame`, we can easily plot a single stock's closing price by selecting the specific column and calling `.plot()`. The following command plots AAPL:

```
In [8]:
_ = daily_close_px['AAPL'].plot(figsize=(12,8));
```



All the close prices can also be easily plotted against each other simply by calling `.plot()` on the entire `DataFrame`:

```
In [9]:
   _ = daily_close_px.plot(figsize=(12,8));
```

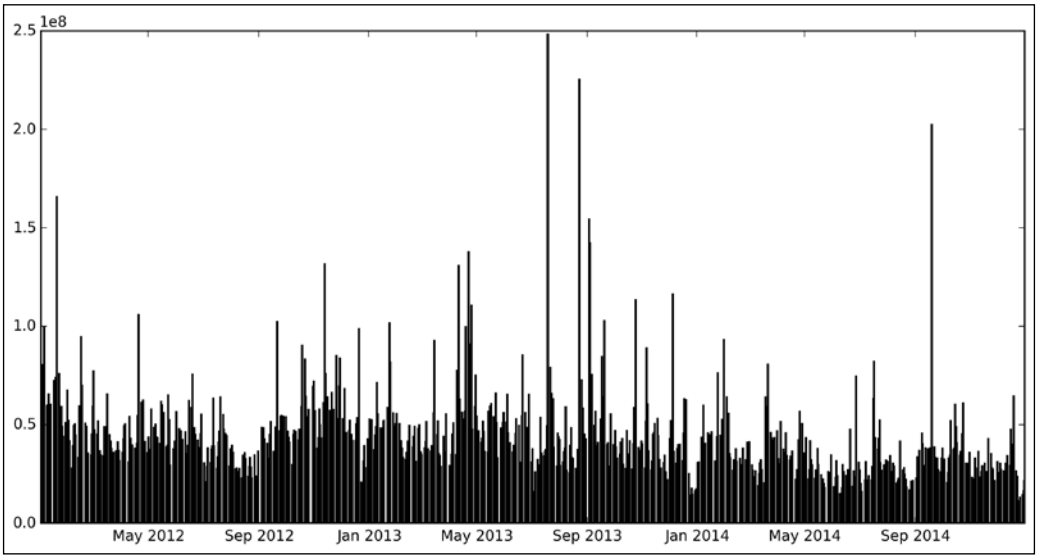


Plotting volume-series data

Stock trading volume data is normally plotted using matplotlib bar charts. This is made almost embarrassingly easy using pandas and the `.bar()` function. The following command plots the volume for `MSFT`:

```
In [10]:
   msftV = all_data.Volume.loc['MSFT']
```

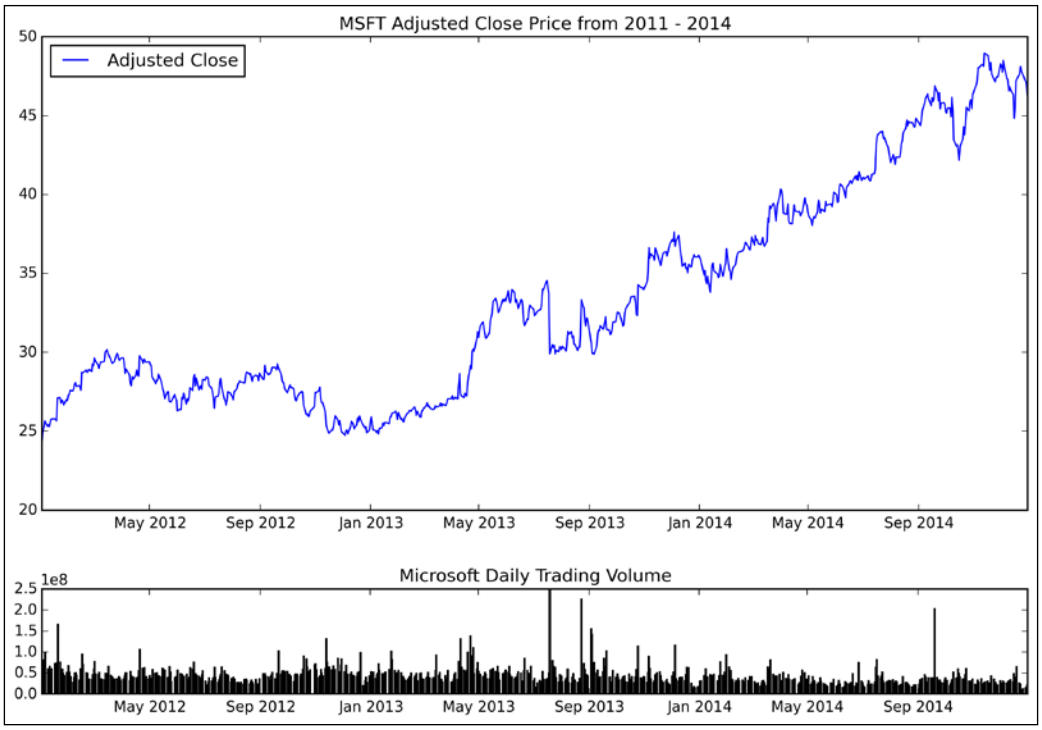
```
   plt.bar(msftV.index, msftV)
   plt.gcf().set_size_inches(12,6)
```



Combined price and volumes

A common type of financial graph plots a stock trading volume relative to its closing price. The following command constructs this combined chart:

```
In [11]:
   top = plt.subplot2grid((4,4), (0, 0), rowspan=3, colspan=4)
   top.plot(daily_close_px.index,
            daily_close_px['MSFT'],
            label='Adjusted Close')
   plt.title('MSFT Adjusted Close Price from 2011 - 2014')
   plt.legend(loc=2)
   bottom = plt.subplot2grid((4,4), (3,0), rowspan=1, colspan=4)
   bottom.bar(msftV.index, msftV)
   plt.title('Microsoft Daily Trading Volume')
   plt.gcf().set_size_inches(12,8)
   plt.subplots_adjust(hspace=0.75)
```



Plotting candlesticks

The open-high-low-close plots, often referred to as candlestick charts, are a type of chart used to illustrate movements in the price of a financial instrument over time. These charts generally consist of a thin vertical line for each unit of time that represents the range of the price during that unit of time and then overlying the thin line is a thicker bar that represents the spacing between the open and close prices. From these charts, it is easy to get a visual feel for the overall movement of the price not only over the entire duration of the chart but also how much the price varies during each unit of measurement.

To demonstrate the process of creating candlestick charts, we will utilize the data for MSFT from the month of December 2014, plotting each day's data as a separate candlestick.

We will also demonstrate the process of selecting specific dates for the *x* axis labels as displaying labels for all 31 days of the month will be too cluttered. The out chart will display labels only for the Mondays during the month.

Additionally, we will format the labels in the `MMM DD` format, where `MMM` represents a three-character month code (in this case, always Dec), and `DD` will be a two-digit date. As an example, the label for Monday December 15 will be Dec 15.

The first step we will perform is to select the subset of data for MSFT in Dec 15 from our `DataFrame` of adjusted close values. We have pivoted both `Ticker` and `Date` into the index, so we will use chained calls to `.loc` to first retrieve only the rows for MSFT and then a slice using a partial date specification to extract only rows for `2014-12`:

```
In [12]:
   subset = all_data.loc['MSFT'].loc['2014-12':'2014-12'] \
                    .reset_index()
   subset[:5]


Out[12]:
           Date   Open   High    ...      Close    Volume  Adj Close
   0 2014-12-01  47.88  48.78    ...      48.62  31191600      48.28
   1 2014-12-02  48.84  49.05    ...      48.46  25743000      48.12
   2 2014-12-03  48.44  48.50    ...      48.08  23534800      47.74
   3 2014-12-04  48.39  49.06    ...      48.84  30320400      48.49
   4 2014-12-05  48.82  48.97    ...      48.42  27313400      48.08
```

We reset the index to move `Date` into a column as, in the end, the date for the charting function needs to be in a column. The process is complicated in that the date-formatting functions we will use do not use the same representation for a date that pandas uses. We will, therefore, need to convert the values in our `Date` column into that representation and add them as a new column to our set of data.

The representation of a date required by the date formatter is a floating point number representing the number of days since the `0001-01-01` universal time plus `1`.

You can find out more about this date representation and the formatting of labels for dates at `http://matplotlib.org/api/dates_api.html`.

We can convert dates to this representation by first converting our pandas date to `pydatetime` and then using the `matplotlib.date2num` function to convert once more into the representation needed for the matplotlib label formatter. The following command will use the `.apply()` method of the `DataFrame` to convert each value in the `Date` column to this representation and add it as the new column `date_num`:

```
In [13]:
   import matplotlib.dates as mdates
   subset['date_num'] = subset['Date'] \
       .apply(lambda date: mdates.date2num(date.to_pydatetime()))
   subset[:5]
Out[13]:
          Date   Open   High    ...      Volume  Adj Close  date_num
   0 2014-12-01  47.88  48.78    ...    31191600      48.28    735568
   1 2014-12-02  48.84  49.05    ...    25743000      48.12    735569
   2 2014-12-03  48.44  48.50    ...    23534800      47.74    735570
   3 2014-12-04  48.39  49.06    ...    30320400      48.49    735571
   4 2014-12-05  48.82  48.97    ...    27313400      48.08    735572
```

Our data is almost ready for use in drawing our candlestick chart. But unfortunately, the `candlestick_ohlc` function does not know how to work with `DataFrame` objects, so we must convert our data to another format. Specifically, we need to provide this function with a list of tuples, where each tuple consists of the `date_num`, `Open`, `High`, `Low`, and `Close`). We can organize the required data this way with the following command:

```
In [14]:
   subset_as_tuples = [tuple(x) for x in subset[['date_num',
                                                 'Open',
                                                 'High',
                                                 'Low',
                                                 'Close']].values]
   subset_as_tuples

Out[14]:
   [(735568.0,
     47.880000000000003,
     48.780000000000001,
     47.710000000000001,
     48.619999999999997),
    (735569.0,
     48.840000000000003,
     49.049999999999997,
     48.200000000000003,
```

```
     48.460000000000001),
    (735570.0, 48.439999999999998, 48.5, 47.810000000000002,
   48.079999999999998),
    (735571.0,
     48.390000000000001,
     49.060000000000002,
     48.200000000000003,
     48.840000000000003),
    (735572.0, 48.82, 48.969999999999999, 48.380000000000003,
   48.420000000000002)]
```

The input data for our chart is now ready. To set up the chart to know how we would like to format the *x* axis labels, we need to do two things. The first thing to do is create an instance of `DateFormatter` and configure it to format the dates as we want, as shown here:

```
In [15]:
   from matplotlib.dates import DateFormatter
   week_formatter = DateFormatter('%b %d')  # e.g., Jan 12
```

We also need to let the chart know how to select which data points on the *x* axis that we would like to label. We do this by creating an instance of `WeekdayLocator` and initializing it with the constant `MONDAY`:

```
In [16]:
   from matplotlib.dates import (WeekdayLocator, MONDAY)
   mondays = WeekdayLocator(MONDAY) # major ticks on the mondays
```

We are now all set to draw the chart with the following command:

```
In [17]:
   plt.figure(figsize(12,8))
   fig, ax = plt.subplots()
   ax.xaxis.set_major_locator(mondays)
   ax.xaxis.set_major_formatter(week_formatter)
   from matplotlib.finance import candlestick_ohlc
   candlestick_ohlc(ax, subset_as_tuples, width=0.6,
                    colorup='g',colordown='r')
```

Here's a final note on the chart that we have created; we specified colors for the candlesticks to represent whether the close price finished the day above or below the open price. If it finished above the open price, we had an overall gain during the day, and we denote this with green. In the other scenario, we use red to denote that we had a loss for the day.

Fundamental financial calculations

There are a number of analyses and data conversions commonly used to analyze the performance of historical stock quotes. These calculations generally relate to either analyzing the rate of return from an investment in a stock over a daily or monthly basis or how multiple stocks perform relative to each other or a market index. The calculations could also relate to determining the riskiness of an investment in a stock relative to others. We will now look at all of these operations using our previously collected stock data.

Calculating simple daily percentage change

The simple daily percentage change (without dividends and other factors) is the amount of percentage change in the value of a stock over a single day of trading. It is defined by the following formula:

$$r_t = \frac{p_t}{p_{t-1}} - 1$$

Using this formula, the following command calculates the percentage change for AA between `2014-01-04` and `2014-01-05`:

```
In [18]:
   AA_p_t0 = daily_close_px.iloc[0]['AA']  #Pt-1
   AA_p_t1 = daily_close_px.iloc[1]['AA']  #Pt
   r_t1 = AA_p_t1 / AA_p_t0 - 1 # returns
   AA_p_t0, AA_p_t1, r_t1


Out[18]:
   (8.91, 9.12, 0.023569023569023573)
```

AA moved from `8.91` to `9.12` between those two days for an increase of 2.4 percent. We can use this result to determine the correctness of applying this formula to an entire `DataFrame`.

There are several ways of calculating the simple daily return across an entire `DataFrame`. One means is by using slicing. The following command uses the trick of dividing a slice of the `DataFrame` that excludes the first row by the values sliced to exclude the last value:

```
In [19]:
   dpc_1 = daily_close_px.iloc[1:] / \
           daily_close_px.iloc[:-1].values - 1
   dpc_1.ix[:,'AA':'AAPL'][:5]


Out[19]:
   Ticker          AA   AAPL
   Date
   2012-01-04   0.024   0.005
   2012-01-05  -0.010   0.011
```

```
2012-01-06 -0.021  0.010
2012-01-09  0.029 -0.002
2012-01-10  0.001  0.004
```

At first glance, you might wonder how this works. This example benefits from the fact that we are dividing a slice of the values in a `DataFrame` by a slice of the values in a two-dimensional array, which doesn't do an alignment of values as the denominator does not have an index (it's not a pandas object at all).

We can visualize this using the following two commands:

```
In [20]:
   price_matrix_minus_day1 = daily_close_px.iloc[1:]
   price_matrix_minus_day1[:5]


Out[20]:
   Ticker        AA   AAPL   DAL  ...     MSFT    PEP    UAL
   Date                           ...
   2012-01-04  9.10  55.71  7.89  ...    25.00  60.75  18.52
   2012-01-05  9.02  56.33  8.20  ...    25.25  60.28  18.39
   2012-01-06  8.83  56.92  8.19  ...    25.64  59.52  18.21
   2012-01-09  9.09  56.83  8.15  ...    25.31  59.83  17.93
   2012-01-10  9.10  57.03  8.14  ...    25.40  59.77  17.48

In [21]:
   daily_close_px.iloc[:-1].values

Out[21]:
   array([[   8.89,  55.41,    7.92, ...,  24.42,   60.44,   18.9 ],
          [   9.1 ,  55.71,    7.89, ...,  25.  ,   60.75,   18.52],
          ...,
          [  15.79, 113.46,   48.68, ...,  47.11,   96.09,   65.22],
          [  15.82, 112.08,   49.13, ...,  46.69,   95.32,   66.05]])
```

So, when we do the division (and subsequently subtract `1` from the division), pandas matches the row/column in the `DataFrame` to the row/column in the array. As an example, `2012-01-14/AA` is *9.1 / 8.89 – 1 = 0.024*, which matches with our result.

Another way to perform this calculation is by performing a shift of the values using the pandas `.shift()` method:

```
In [22]:
   dpc_2 = daily_close_px / daily_close_px.shift(1) - 1
   dpc_2.ix[:,0:2][:5]


Out[22]:
   Ticker          AA   AAPL
   Date
   2012-01-03     NaN    NaN
   2012-01-04   0.024  0.005
   2012-01-05  -0.009  0.011
   2012-01-06  -0.021  0.010
   2012-01-09   0.029 -0.002
```

Note that this has the same values as the previous example but also includes `2012-01-03` with the `NaN` values.

This process performs alignment as `.shift()` moves the values along the axis and results in a pandas object instead of a list of values.

Probably the easiest way to do this is with the built-in `.pct_change()` method of a pandas `Series` or `DataFrame`. This calculation is actually so commonly performed in pandas and finance that it was baked into the library:

```
In [23]:
   daily_pct_change = daily_close_px.pct_change()
   daily_pct_change.ix[:,0:2][:5]


Out[23]:
   Ticker          AA   AAPL
   Date
   2012-01-03     NaN    NaN
   2012-01-04   0.024  0.005
   2012-01-05  -0.009  0.011
   2012-01-06  -0.021  0.010
   2012-01-09   0.029 -0.002
```

The last thing we will want to do here is set the `NaN` values to `0`. This is not strictly required, and the examples will work without it, but it is a good practice:

```
In [24]:
   daily_pct_change.fillna(0, inplace=True)
   daily_pct_change.ix[:5,:5]


Out[24]:
   Ticker          AA   AAPL    DAL     GE    IBM
   Date
   2012-01-03   0.000  0.000  0.000  0.000  0.000
   2012-01-04   0.024  0.005 -0.004  0.011 -0.004
   2012-01-05 -0.009  0.011  0.039 -0.001 -0.005
   2012-01-06 -0.021  0.010 -0.001  0.005 -0.011
   2012-01-09   0.029 -0.002 -0.005  0.011 -0.005
```

Calculating simple daily cumulative returns

The cumulative daily rate of return is useful to determine the value of an investment at regular intervals after investment. This is calculated from the daily percentage change values by multiplying (1 + the current day's percentage change) with the cumulative product of all of the previous values. This is represented by the following formula:

$$i_i = (1 + r_t) \cdot i_{t-1}\,, \quad i_0 = 1$$

This is actually calculated very succinctly using the following code and the `.cumprod()` method:

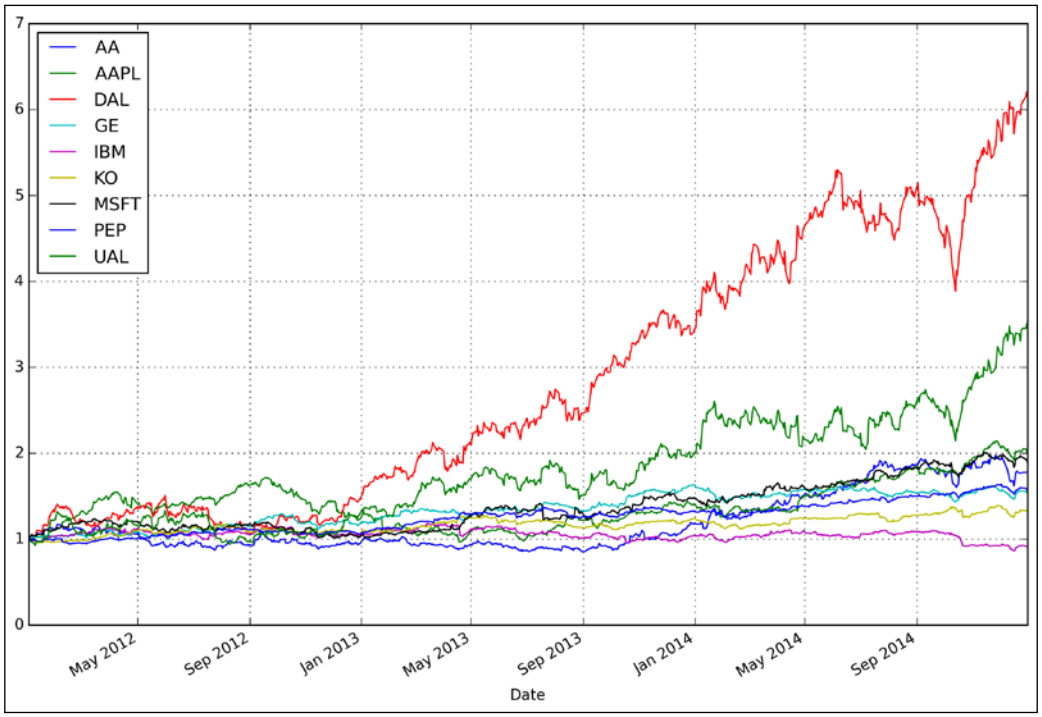
```
In [25]:
   cum_daily_return = (1 + daily_pct_change).cumprod()
   cum_daily_return.ix[:,:2][:5]


Out[25]:
   Ticker          AA   AAPL    DAL     GE    IBM
   Date
   2012-01-03   0.000  0.000  0.000  0.000  0.000
   2012-01-04   0.024  0.005 -0.004  0.011 -0.004
   2012-01-05 -0.009  0.011  0.039 -0.001 -0.005
```

```
2012-01-06 -0.021  0.010 -0.001  0.005 -0.011
2012-01-09  0.029 -0.002 -0.005  0.011 -0.005
```

This informs us that the value of $1 invested in AA on `2012-01-03` would be worth $1.772 on `2014-12-31`.

We can plot the cumulative returns to see how the different stocks compare. This gives a nice view of how the stocks will change your investment over time and how they perform relative to each other:

```
In [26]:
   cum_daily_return.plot(figsize=(12,8))
   plt.legend(loc=2);
```



Analyzing the distribution of returns

If you want to get a feel for the difference in distribution of the daily returns for a particular stock, you can plot the returns using several common visualizations, including:

- Histograms
- Q-Q plots
- Box and whisker plots

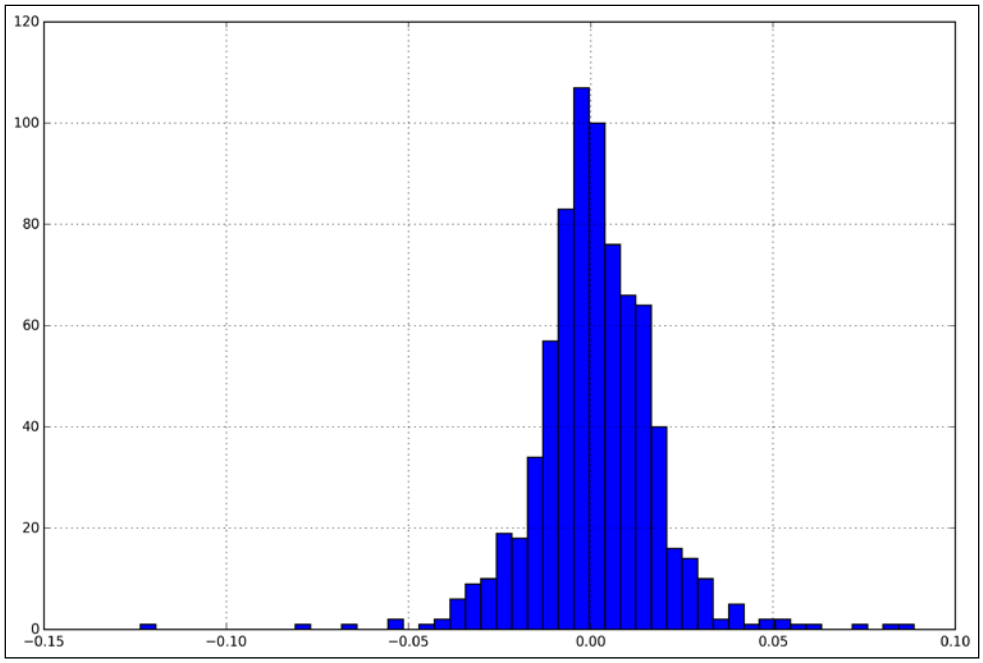
Histograms

Histograms give you an overall feel for the distribution of returns. In general, return distributions are approximately normal in shape, demonstrating a familiar bell curve shape.

Histograms can be generated using the `.hist()` method of a pandas `Series`. The method can be supplied with a number of different parameters, of which one of the most important is the number of bins that the data is to be lumped into. We will use `50` bins, which gives a good feel for the distribution of daily changes across three years of data.

The following command shows the distribution of the daily returns for `AAPL`:

```
In [27]:
   aapl = daily_pct_change['AAPL']
   aapl.hist(bins=50, figsize=(12,8));
```



This chart tells us several things. First, most of the daily changes center around **0.0**, and there is a small amount of skew to the left, but the data appears fairly symmetrical and normally distributed.

If we use the `.describe()` method on this data, we will get some useful summary statistics that describe the histogram:

```
In [28]:
   aapl.describe()


Out[28]:
   count    754.000
   mean       0.001
   std        0.017
   min       -0.124
   25%       -0.007
   50%        0.001
   75%        0.011
   max        0.089
   Name: AAPL, dtype: float64
```

Using this information, some of our conclusions from the histogram can be rationalized. The mean of the distributions is very close to `0.0`, being `0.001`. The standard deviation is `0.017`. The percentiles tell us that 25 percent of the points fall below `-0.007`, 50 percent below `0.001`, and 75 percent below `0.011`.

We can provide parameters to `.describe()` to further specify the percentiles that we would like to calculate. The following command asks the method to give us the breakdown at the `2.5`, `50`, and `97.5` percentiles:

```
In [29]:
   aapl.describe(percentiles=[0.025, 0.5, 0.975])


Out[29]:
   count    754.000
   mean       0.001
   std        0.017
   min       -0.124
   2.5%      -0.032
   50%        0.001
```

```
97.5%       0.032
max         0.089
Name: AAPL, dtype: float64
```

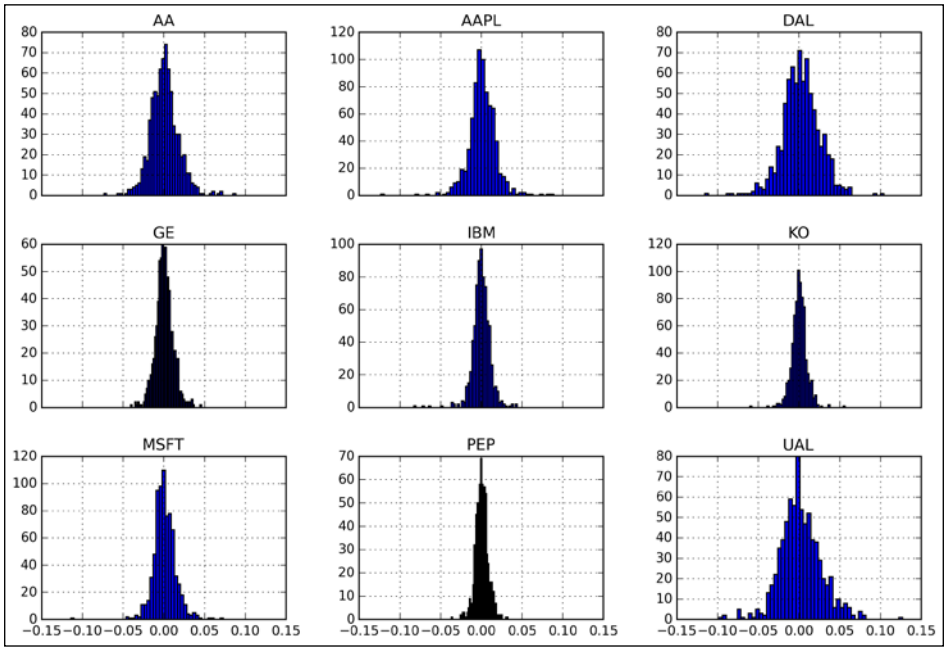
This range of percentiles is commonly used to formulate a 95 percent confidence interval. If our return distribution is perfectly normally distributed (with an equal distribution of gains and losses), then we would expect our 2.5 percent value to be -1.95996 times the standard deviation less than the mean, and the 97.5 percent value to be 1.95996 times the standard deviation above the mean.

Manually calculating these, we get the 2.5 percent value as -0.032 and the 97.5 percent value as 0.034. These are roughly equivalent, again giving us a good feeling that this stock has an equal distribution of gains and losses.

And, statistically speaking, this range of values gives us the 95 percent confidence interval, which tells us that over the last three years, the daily return on 95 percent of the days will fall within -0.032 percent and 0.032 percent.

To compare the return distributions of more than one stock using histograms, we can visualize these distributions on all stocks in a single visual by creating a matrix of histograms. As demonstrated here, pandas allows us to do this very simply:

```
In [30]:
   daily_pct_change.hist(bins=50, sharex=True, figsize=(12,8));
```



The labels on the axis are a bit squished together. That is fine as it is the relative shapes of the histograms that are the most important. The `sharex=True` parameter tells pandas to ensure a common range of *x* axis values on all of the histograms, which facilitates our comparison of the overall distributions.

Using this chart, we get a feel for the difference in performance of these nine stocks during this time. Stocks with a wider interval have higher fluctuation in returns and, hence, are more volatile. Stocks where the curve is skewed demonstrate a propensity to have either larger (skewed right) or smaller rates (skewed left) of return during the period of measurement.

Q-Q plots

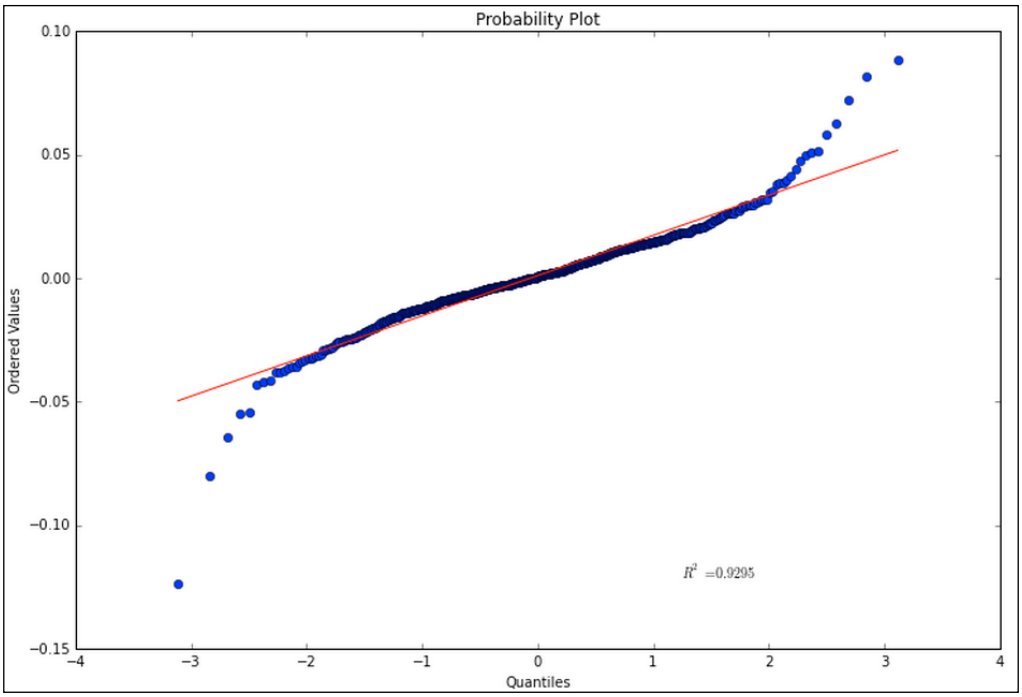
A Q-Q plot, short for Quantile-Quantile plot, is a probability plot comparing two probability distributions by plotting their quantiles against each other. We can use a Q-Q plot of the returns of a stock compared to a normal distribution to get a feel of how close our returns are to a normal distribution. We can get an idea of this from histograms, but a Q-Q plot gives a much better representation.

We can create a Q-Q plot of our returns using the `probplot()` function of `scipy.stats`.

You will need to ensure you have `scipy` installed in your Python environment to generate these plots.

As an example, we can plot the returns of AAPL against a sequence of random normal values. We can generate this plot using the following command that generates a Q-Q plot of our distribution date compared to a normal distribution:

```
In [31]:
   import scipy.stats as stats
   f = plt.figure(figsize=(12,8))
   ax = f.add_subplot(111)
   stats.probplot(aapl, dist='norm', plot=ax)
   plt.show();
```



We will not get into detailed analysis of Q-Q plots in this book. For more details, I recommend `http://en.wikipedia.org/wiki/Q-Q_plot` and `http://stats.stackexchange.com/questions/101274/how-to-interpret-a-qq-plot`.

A distribution of data in a Q-Q plot would show perfect correspondence to a normal distribution if all of the blue dots fell exactly along the red line and the slope of the red line would be 1.0 (representing perfect correlation and an `R^2` value of `1.0`). Our returns are correlated at a level of 0.9295, which is representative of a very high degree of correlation.

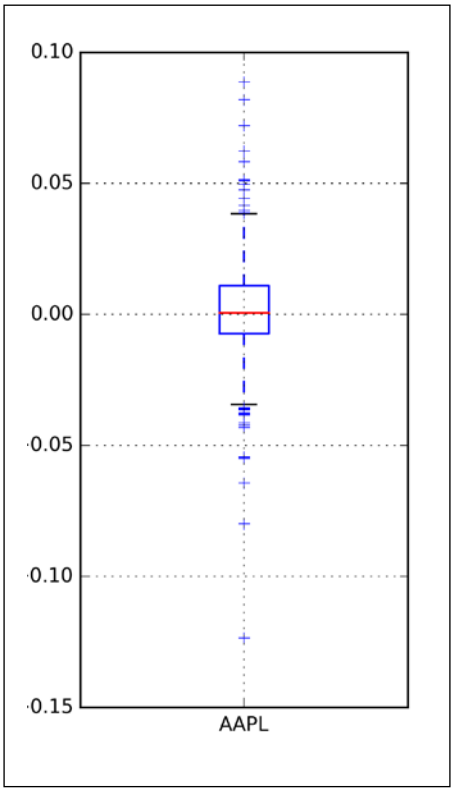
Between quantiles -2 and +2, most of our data is very close to being perfectly correlated. This range is also very close to our 95 percent confidence interval (just slightly wider, which actually means higher confidence). It is outside this range that we begin to see differences in the levels of correlation of the distribution with what appears to be a similar amount of skew along both tails but perhaps with a little more towards the negative.

Box-and-whisker plots

A box plot is a convenient way to graphically depict groups of data through their quartiles. The box portion of the plot represents the range from the low quantile to the high quantile, and the box is split by a line that represents the median value. A box plot may also have lines extending out from both sides of the box, which represent the amount of variability outside of the upper and lower quartiles. These are often referred to as whiskers, hence the use of the term box-and-whisker plot.

To demonstrate this, the following command creates a box-and-whisker plot for the AAPL daily returns:

```
In [32]:
   daily_pct_change[['AAPL']].plot(kind='box', figsize=(3,6));
```

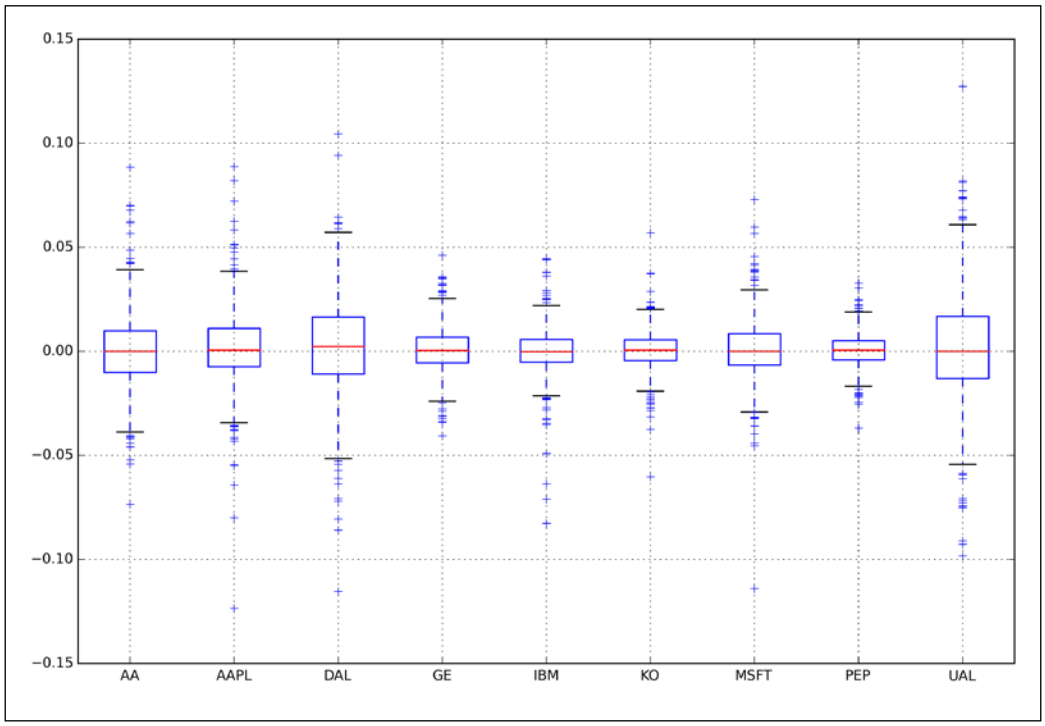


The box in the chart represents the range of the values in the 25 percent (Q1) and 75 percent (Q3) quartiles. The red line is the median value.

The dashed lines extend out to what is referred to as an IQR of 1.5, where the **inter-quartile range** (**IQR**) is defined as Q3–Q1. In this case, the IQR is *1.5 * (0.011-(-0.007)) = 0.027*. Hence, we have a line at *0.011 + 0.027 = 0.038* and another at *-0.007-0.027 = -0.034*. These values represent an amount where sample values beyond are considered outliers. Those outliers are then individually plotted along the vertical to give an idea of their values and quantities.

These become particularly useful when we align them next to each other to compare the distributions of multiple datasets. To demonstrate this, the following command does this for the returns of all of our stocks:

```
In [33]:
   daily_pct_change.plot(kind='box', figsize=(12,8));
```



This plot gives us a very good comparison of the performance of these stocks over this period of time. The wider the box, the higher the variability and the risk. The closer the median line to either side of the box or the longer a whisker is than the other, the greater the skew in the distribution.

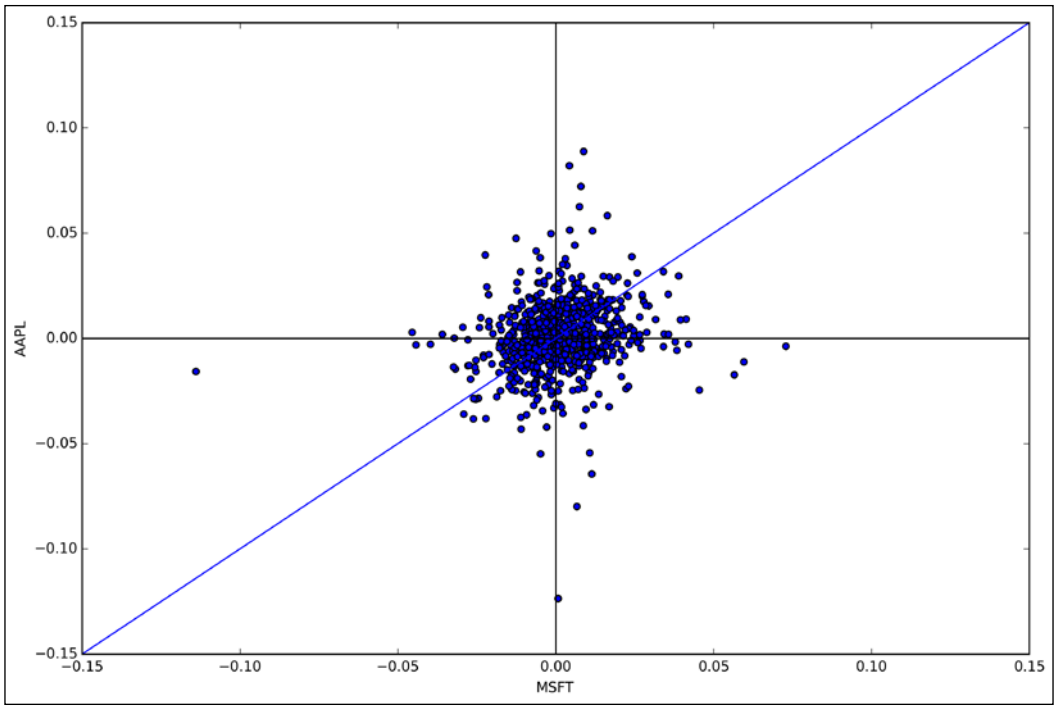
Comparison of daily percentage change between stocks

A scatter plot is also a very effective means of being able to visually determine the relationship between the rate of percentage change in prices between two stocks. To demonstrate this, we will use the following function that will plot the values in two series relative to each other:

```
In [34]:
   def render_scatter_plot(data, x_stock_name,
                           y_stock_name, xlim=None, ylim=None):
       fig = plt.figure(figsize=(12,8))
       ax = fig.add_subplot(111)
       ax.scatter(data[x_stock_name], data[y_stock_name])
       if xlim is not None: ax.set_xlim(xlim)
       ax.autoscale(False)
       ax.vlines(0, -10, 10)
       ax.hlines(0, -10, 10)
       ax.plot((-10, 10), (-10, 10))
       ax.set_xlabel(x_stock_name)
       ax.set_ylabel(y_stock_name)
```

The following graph shows the relationship between the daily percentage change of `MSFT` and `AAPL`:

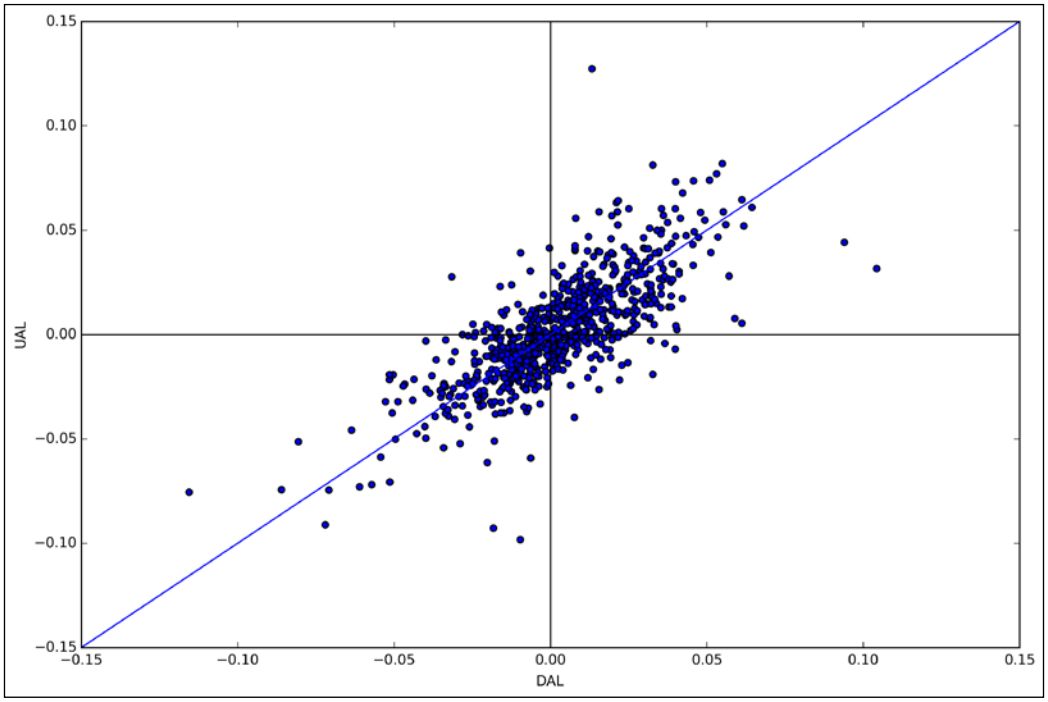
```
In [35]:
   limits = [-0.15, 0.15]
   render_scatter_plot(daily_pct_change, 'MSFT', 'AAPL', xlim=limits)
```



In this plot, excluding several outliers, this cluster appears to demonstrate a small amount of correlation between the two stocks as the linear correlation would seem to be closer to horizontal (slope = 0, that is, no correlation) than a perfect diagonal. As we have seen, an actual correlation actually shows the correlation to be 0.236 (the slope of the regression line), which backs up our visual analysis.

This can be compared to the relationship between `DAL` and `UAL`, which shows very high correlation:

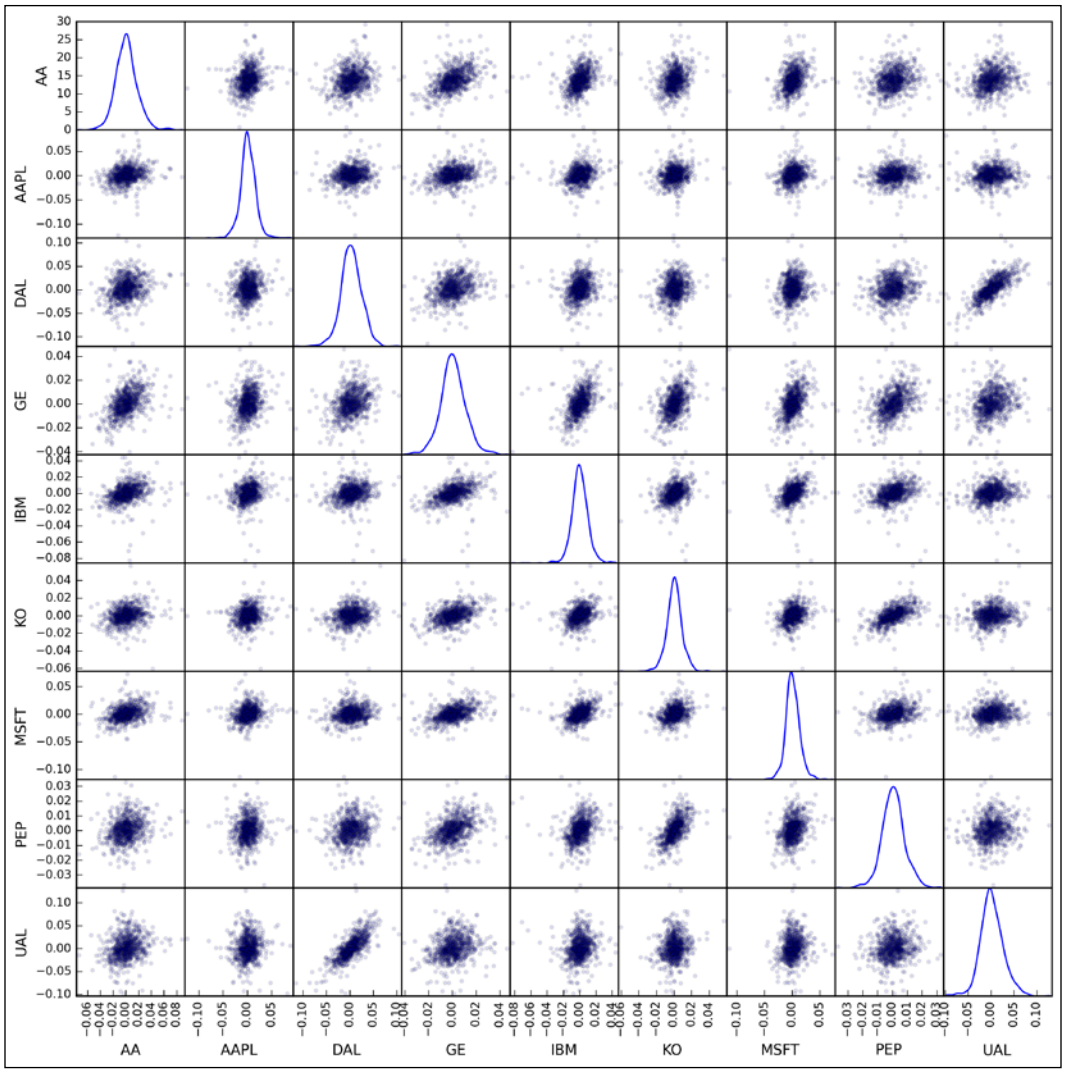
```
In [36]:
   render_scatter_plot(daily_pct_change, 'DAL', 'UAL', xlim=limits)
```



This is supported with an actual correlation that is calculated to be 0.76.

It is not required to draw every graph independently to compare all relationships. The very useful scatter matrix graph provided by pandas plots the scatters for all combinations of stocks that gives a very easy means of eyeballing all combinations. The use of `alpha=0.1` adds transparency to the points on the graph, which helps with small graphs with many overlapping points, as shown here:

```
In [37]:
   # all stocks against each other, with a KDE in the diagonal
   _ = pd.scatter_matrix(daily_pct_change, diagonal='kde', alpha=0.1,
                         figsize=(12,12));
```



The diagonal is a kernel density estimation graph, which estimates the distribution and, in simple terms, represents a continuous histogram of the relationships.

Moving windows

A number of functions are provided to compute moving (also known as rolling) statistics, where the function computes the statistic on a window of data represented by a particular period of time and then slides the window across the data by a specified interval, continually calculating the statistic as long as the window falls first within the dates of the time-series.

With the following functions, pandas provides direct support for rolling windows:

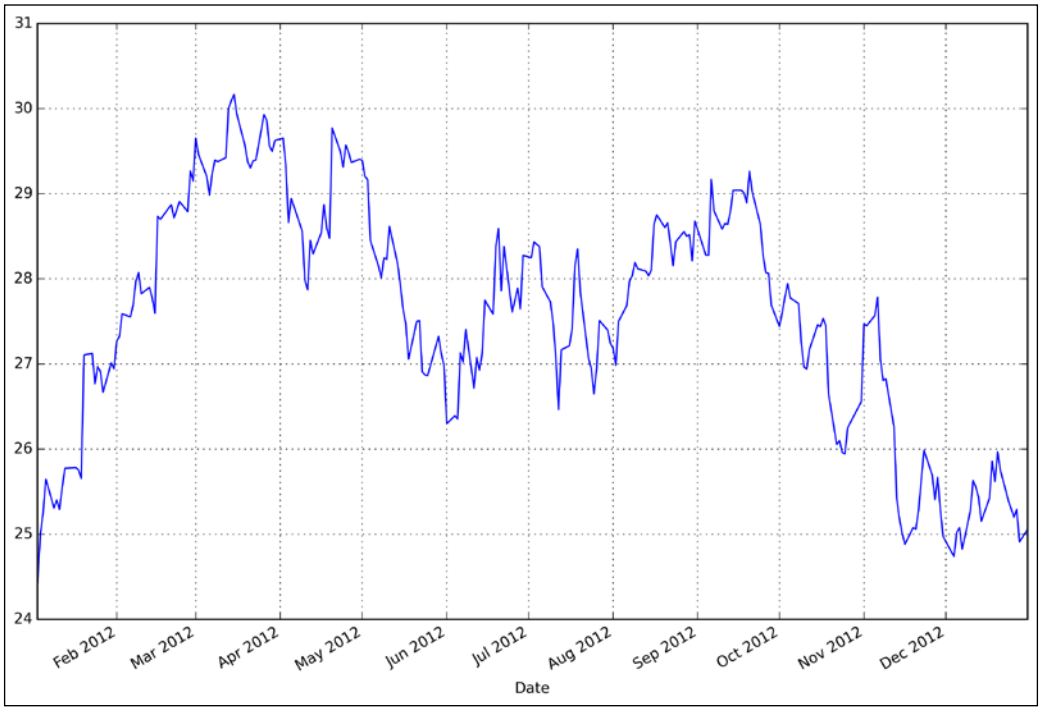
Function	Description
`rolling_mean`	This is the mean of the values in the window
`rolling_std`	This is the standard deviation of the values in the window
`rolling_var`	This is the variance of values
`rolling_min`	This is the minimum of the values in the window
`rolling_max`	This is maximum of the values in the window
`rolling_cov`	This is the covariance of values
`rolling_quantile`	This is the moving window score at the percentile/ sample quantile
`rolling_corr`	This is the correlation of the values in the window
`rolling_median`	This is the median of the values in the window
`rolling_sum`	This is the sum of the values in the window
`rolling_apply`	This is the application of a user function to the values in the window
`rolling_count`	This is the number of non-NaN values in a window
`rolling_skew`	This is the skewedness of the values in the window
`rolling_kurt`	This is the kurtosis of the 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 this, we will calculate a rolling window on the adjusted close values for MSFT for the year 2012. The following command extracts the raw values for 2012 and plots them to gives us an idea of the shape of the data:

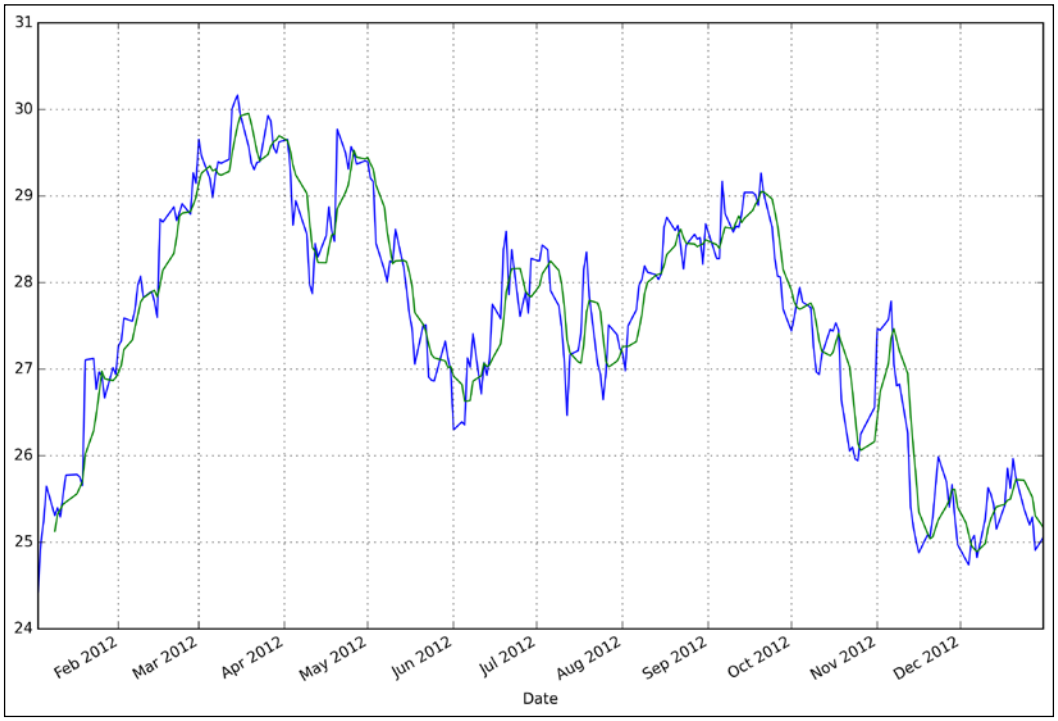
```
In [38]:
   msftAC = msft['2012']['Adj Close']
   msftAC[:5]
```

```
Out[38]:
   Date
   2012-01-03    24.42
   2012-01-04    25.00
   2012-01-05    25.25
   2012-01-06    25.64
   2012-01-09    25.31
   Name: Adj Close, dtype: float64

In [39]:
   sample = msftAC['2012']
   sample.plot(figsize=(12,8));
```



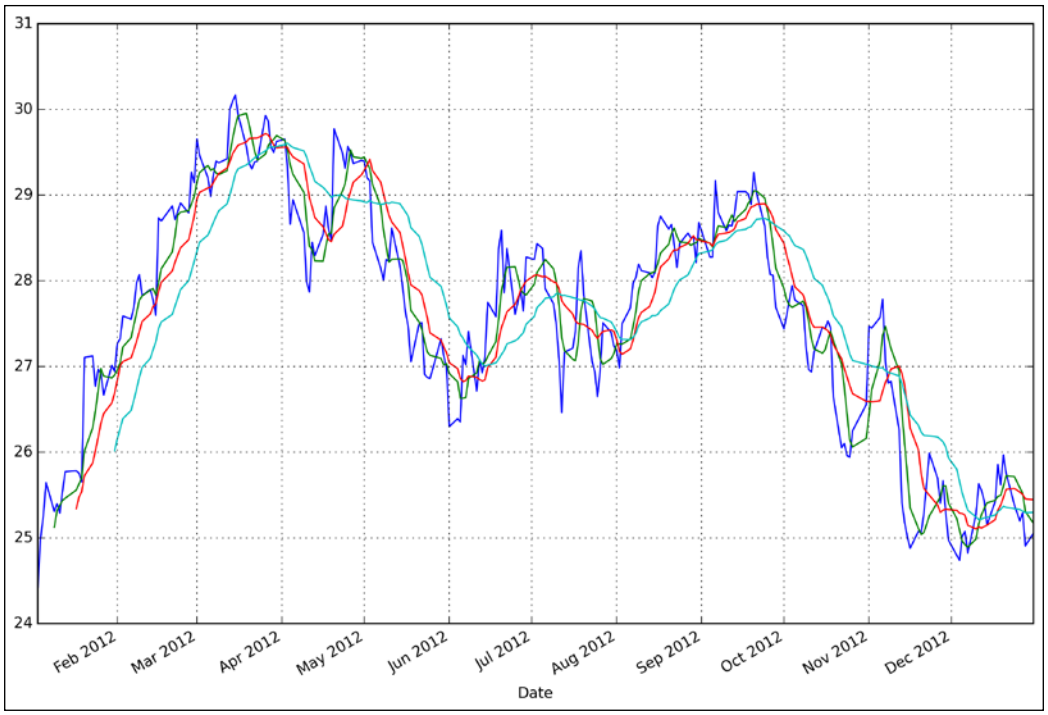
The following command calculates the rolling mean with a window of `5` periods and plots it against the raw data:

```
In [40]:
   sample.plot(figsize=(12,8))
   pd.rolling_mean(sample, 5).plot(figsize=(12,8));
```



From this, it can be seen how the `pd.rolling_mean` function provides a smoother representation of the underlying data. A larger window smoothens out the variance but at the cost of accuracy. We can see how this gets smoother as the window size is increased. The following command plots the rolling mean of window size `5`, `10`, and `20` periods against the raw data:

```
In [41]:
   sample.plot(figsize=(12,8))
   pd.rolling_mean(sample, 5).plot(figsize=(12,8))
   pd.rolling_mean(sample, 10).plot(figsize=(12,8))
   pd.rolling_mean(sample, 20).plot(figsize=(12,8));
```

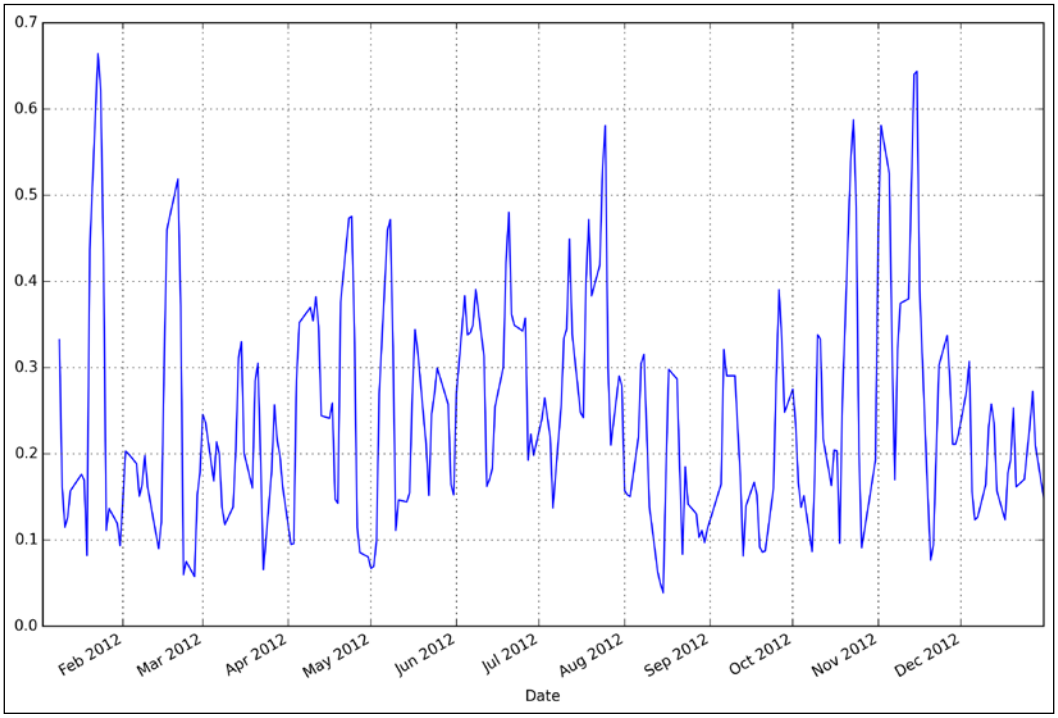


Note that the larger the window, the more the data missing at the beginning of the curve. A window of size *n* requires *n* data points before the measure can be calculated, hence the gap in the beginning of the plot.

Any function can be applied via a rolling window using the `pd.rolling_apply` function. The supplied function will be passed an array of values in the window and should return a single value. Then pandas will combine these results into a time-series.

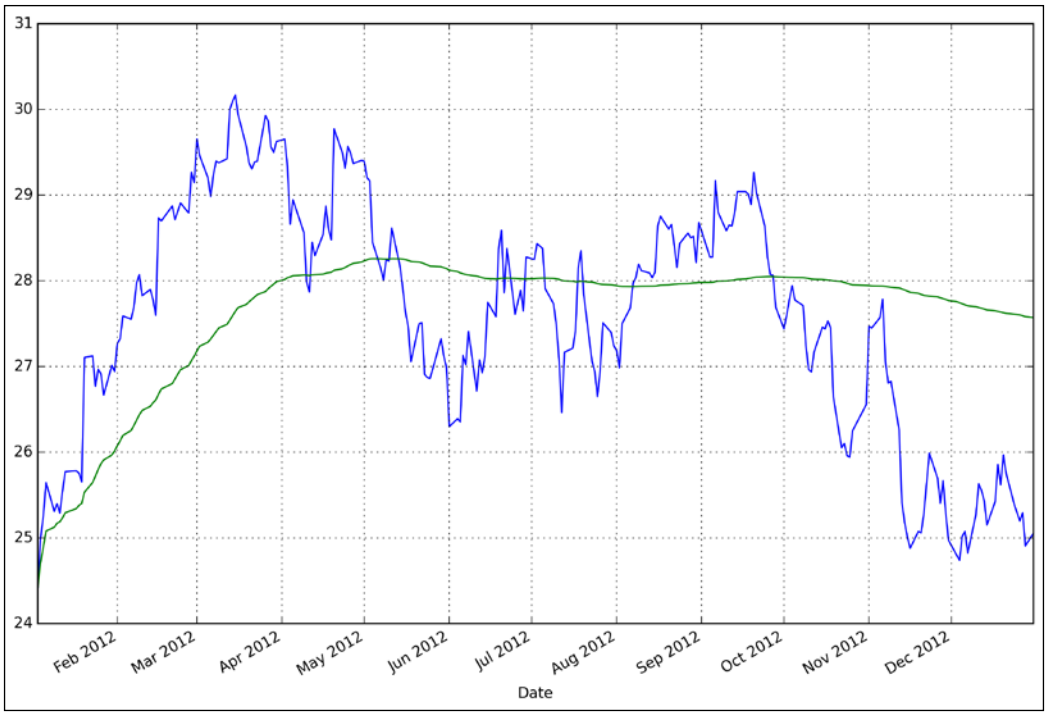
To demonstrate this, the following command calculates the mean average deviation, which gives a feel of how far on average all the values in the sample are from the overall mean:

```
In [42]:
   mean_abs_dev = lambda x: np.fabs(x - x.mean()).mean()
   pd.rolling_apply(sample, 5, mean_abs_dev).plot(figsize=(12,8));
```



An expanding window mean can be calculated using a slight variant of the `pd.rolling_mean` function that repeatedly calculates the mean by always starting with the first value in the time-series, and for every iteration, increasing the window size by one. An expanding window mean will be more stable (less responsive) than a rolling window because greater the size of the window, the less the impact of the next value:

```
In [43]:
   expanding_mean = lambda x: pd.rolling_mean(x, len(x),
                                              min_periods=1)
   sample.plot()
   pd.expanding_mean(sample).plot();
```



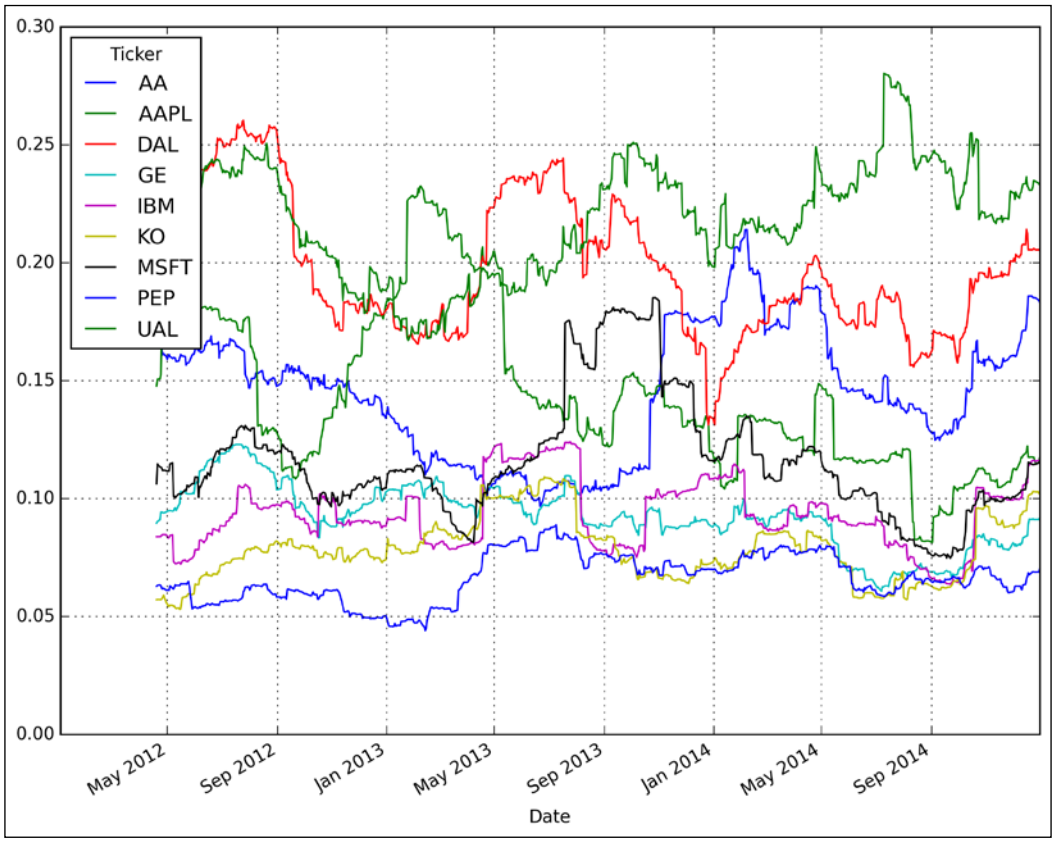
Volatility calculation

The volatility of a stock is a measurement of the change in variance in the returns of a stock over a specific period of time. It is common to compare the volatility of a stock with another stock to get a feel for which may have less risk or to a market index to examine the stock's volatility in the overall market. Generally, the higher the volatility, the riskier the investment in that stock, which results in investing in one over another.

Volatility is calculated by taking a rolling window standard deviation on the percentage change in a stock. The size of the window affects the overall result. The wider the 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 upon the data sampling frequency. Fortunately, pandas makes this very easy to modify interactively.

As a demonstration, the following command calculates the volatility of all the stock in our sample with a window of 75 days:

```
In [44]:
   min_periods = 75
   vol = pd.rolling_std(daily_pct_change, min_periods) * \
                        np.sqrt(min_periods)
   vol.plot(figsize=(10, 8));
```



The lines higher on the chart represent overall higher volatility and hence represent a riskier investment. PEP seems to have the lowest overall volatility, while it appears that UAL has the highest.

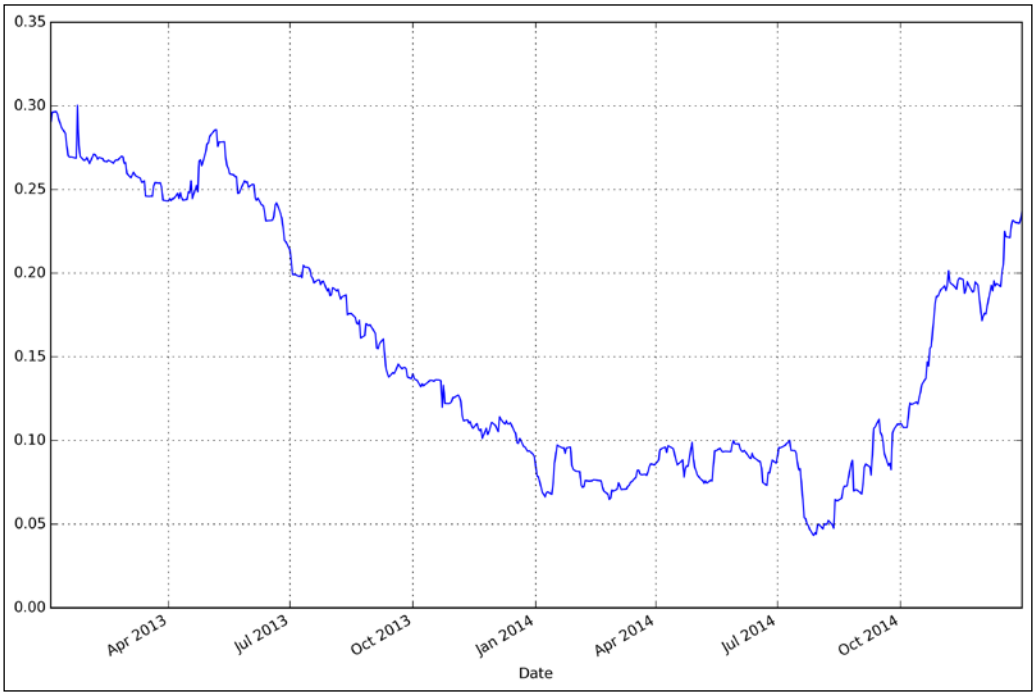
Rolling correlation of returns

We previously examined the calculation of the overall correlation between two stocks over a time period (3 years in our case). This can also be performed using rolling windows to demonstrate how the correlation has changed over time:

```
In [45]:
   rolling_corr = pd.rolling_corr(daily_pct_change['AAPL'],
                                  daily_pct_change['MSFT'],
                                  window=252).dropna()
   rolling_corr[251:] #first 251 are NaN

Out[45]:
   Date
   2014-01-02    0.08
   2014-01-03    0.08
   2014-01-06    0.07
   2014-01-07    0.07
   2014-01-08    0.07
                 ...
   2014-12-24    0.23
   2014-12-26    0.23
   2014-12-29    0.23
   2014-12-30    0.23
   2014-12-31    0.24
   dtype: float64
```

We can visualize this change in correlation over time as follows:

```
In [46]:
   rolling_corr.plot(figsize=(12,8));
```



Least-squares regression of returns

The correlations that we have examined up until this point show the relationship of the change in daily return from two investments. They do not capture the change in the volatility between two investments. This can be calculated using least-squares regression using the pandas `ols()`, the ordinary least-squares function.

The following command calculates this on the returns of `AAPL` and `MSFT`:

```
In [47]:
   model = pd.ols(y=daily_pct_change['AAPL'],
                  x={'MSFT': daily_pct_change['MSFT']},
                  window=250)
   model


Out[47]:
```

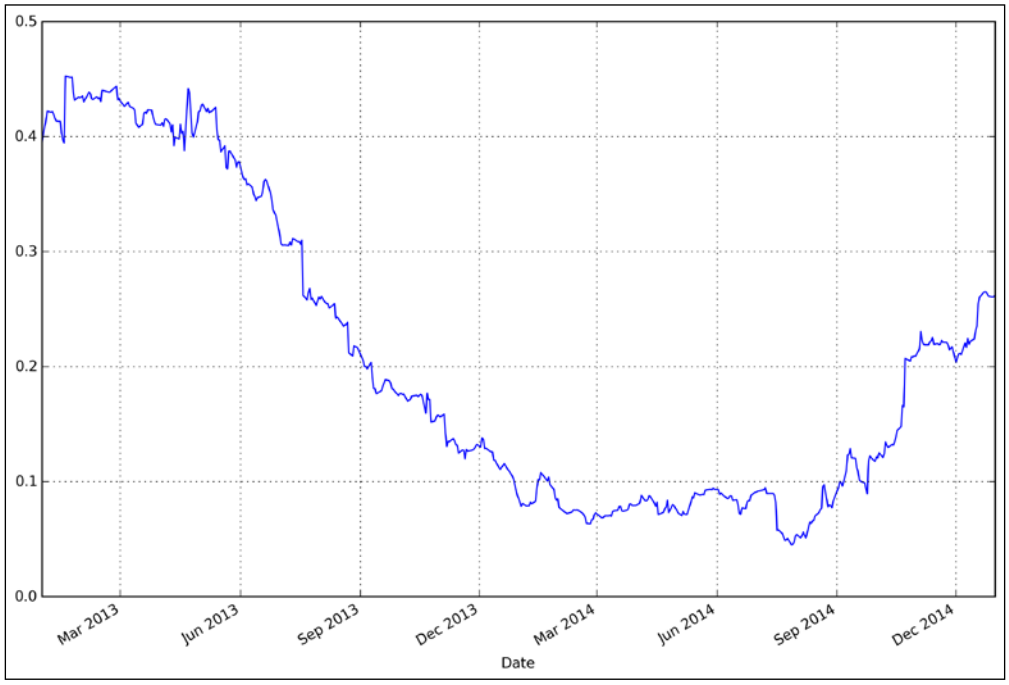
```
------------------Summary of Regression Analysis------------------

Formula: Y ~ <MSFT> + <intercept>

Number of Observations:         250
Number of Degrees of Freedom:   2

R-squared:         0.0535
Adj R-squared:     0.0497

Rmse:              0.0132

F-stat (1, 248):    14.0223, p-value:     0.0002

Degrees of Freedom: model 1, resid 248

-----------------Summary of Estimated Coefficients----------------
 Variable   Coef    Std Err    t-stat    p-value    CI 2.5%    CI 97.5%
-----------------------------------------------------------------------
     MSFT 0.2617     0.0699     3.74     0.0002     0.1247      0.3987
intercept 0.0013     0.0008     1.56     0.1193    -0.0003      0.0030
-------------------------End of Summary-----------------------------
```

The beta from the resulting model gives us an idea of the relationship between the changes in volatility of the two stocks over the period:

```
In [48]:
   model.beta[0:5] # what is the beta?

Out[48]:
                MSFT  intercept
   Date
   2012-12-31  0.394      0.001
   2013-01-02  0.407      0.001
   2013-01-03  0.413      0.001
   2013-01-04  0.421      0.001
   2013-01-07  0.420      0.001
```

Also, this can be easily plotted, as shown here:

```
In [49]:
   _ = model.beta['MSFT'].plot(figsize=(12, 8)); # plot the beta
```



Comparing stocks to the S&P 500

The analyses until this point have been performed only between stocks. It is often useful to perform some of these against a market index such as the S&P 500. This will give a sense of how those stocks compare to movements in the overall market.

At the beginning of the chapter, we loaded the S&P 500 data for the same time period as the other stocks. To perform comparisons, we can perform the same calculations to derive the daily percentage change and cumulative returns on the index:

```
In [50]:
   sp_500_dpc = sp_500['Adj Close'].pct_change().fillna(0)
   sp_500_dpc[:5]
```

```
Out[50]:
   Date
   2012-01-03    0.000
   2012-01-04    0.000
   2012-01-05    0.003
   2012-01-06   -0.003
   2012-01-09    0.002
   Name: Adj Close, dtype: float64
```

We can concatenate the index calculations in the results of the calculations of the stocks. This will let us easily compare the overall set of stocks and index calculations:

```
In [51]:
   dpc_all = pd.concat([sp_500_dpc, daily_pct_change], axis=1)
   dpc_all.rename(columns={'Adj Close': 'SP500'}, inplace=True)
   dpc_all[:5]


Out[51]:
                  SP500    AA  AAPL  ...   MSFT   PEP   UAL
   Date                               ...
   2012-01-03  0.00e+00  0.00  0.00  ...   0.00  0.00  0.00
   2012-01-04  1.88e-04  0.02  0.01  ...   0.02  0.01 -0.02
   2012-01-05  2.94e-03 -0.01  0.01  ...   0.01 -0.01 -0.01
   2012-01-06 -2.54e-03 -0.02  0.01  ...   0.02 -0.01 -0.01
   2012-01-09  2.26e-03  0.03 -0.00  ...  -0.01  0.01 -0.02
```

Now, we calculate the cumulative daily returns with the following command:

```
In [52]:
   cdr_all = (1 + dpc_all).cumprod()
   cdr_all[:5]


Out[52]:
               SP500    AA  AAPL  ...   MSFT   PEP   UAL
   Date                            ...
   2012-01-03      1  1.00  1.00  ...   1.00  1.00  1.00
   2012-01-04      1  1.02  1.01  ...   1.02  1.01  0.98
   2012-01-05      1  1.01  1.02  ...   1.03  1.00  0.97
```

```
   2012-01-06      1  0.99  1.03  ...   1.05  0.98  0.96
   2012-01-09      1  1.02  1.03  ...   1.04  0.99  0.95
```

Also, we will calculate the correlation of the daily percentage change values as follows:

```
In [53]:
   dpc_corrs = dpc_all.corr()
   dpc_corrs

Out[53]:
          SP500    AA  AAPL  ...   MSFT   PEP   UAL
   SP500   1.00  0.60  0.41  ...   0.54  0.52  0.32
   AA      0.60  1.00  0.24  ...   0.31  0.23  0.22
   AAPL    0.41  0.24  1.00  ...   0.19  0.09  0.06
   DAL     0.42  0.25  0.14  ...   0.15  0.17  0.76
   GE      0.73  0.46  0.24  ...   0.34  0.38  0.24
   IBM     0.53  0.31  0.21  ...   0.36  0.26  0.12
   KO      0.53  0.23  0.16  ...   0.27  0.56  0.14
   MSFT    0.54  0.31  0.19  ...   1.00  0.28  0.13
   PEP     0.52  0.23  0.09  ...   0.28  1.00  0.13
   UAL     0.32  0.22  0.06  ...   0.13  0.13  1.00

   [10 rows x 10 columns]
```

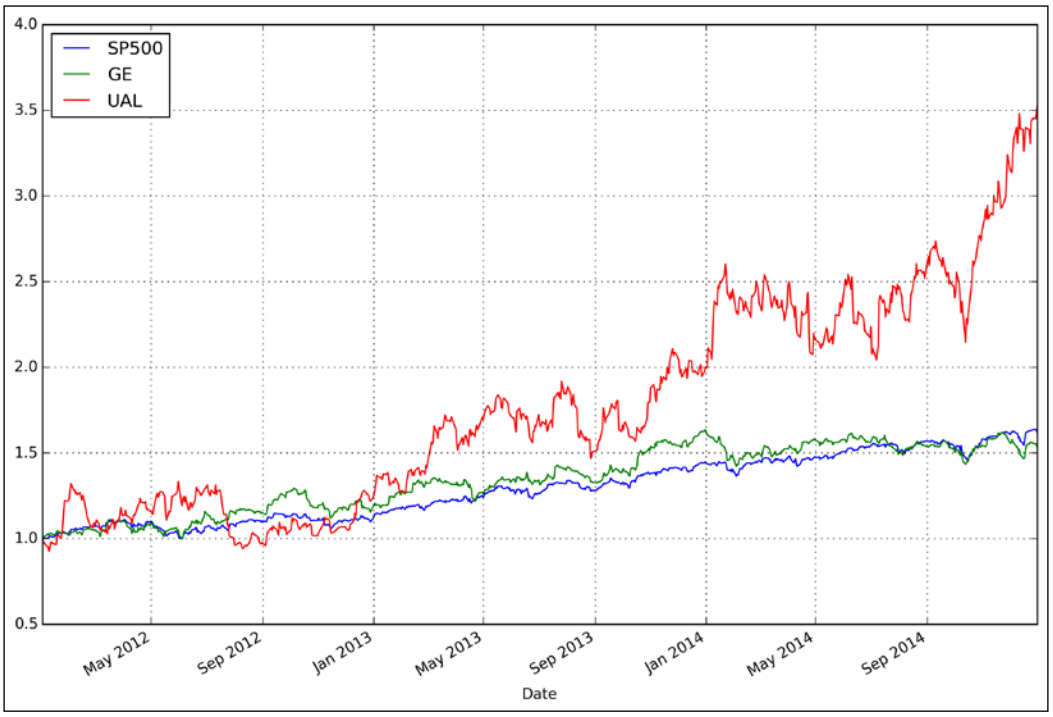
Of interest in the correlations is each stock relative to the S&P 500. We can extract this with the following command:

```
In [54]:
   dpc_corrs.ix['SP500']

Out[54]:
   SP500    1.00
   AA       0.60
   AAPL     0.41
   DAL      0.42
   GE       0.73
  IBM      0.53
   KO       0.53
```

```
MSFT      0.54
PEP       0.52
UAL       0.32
Name: SP500, dtype: float64
```

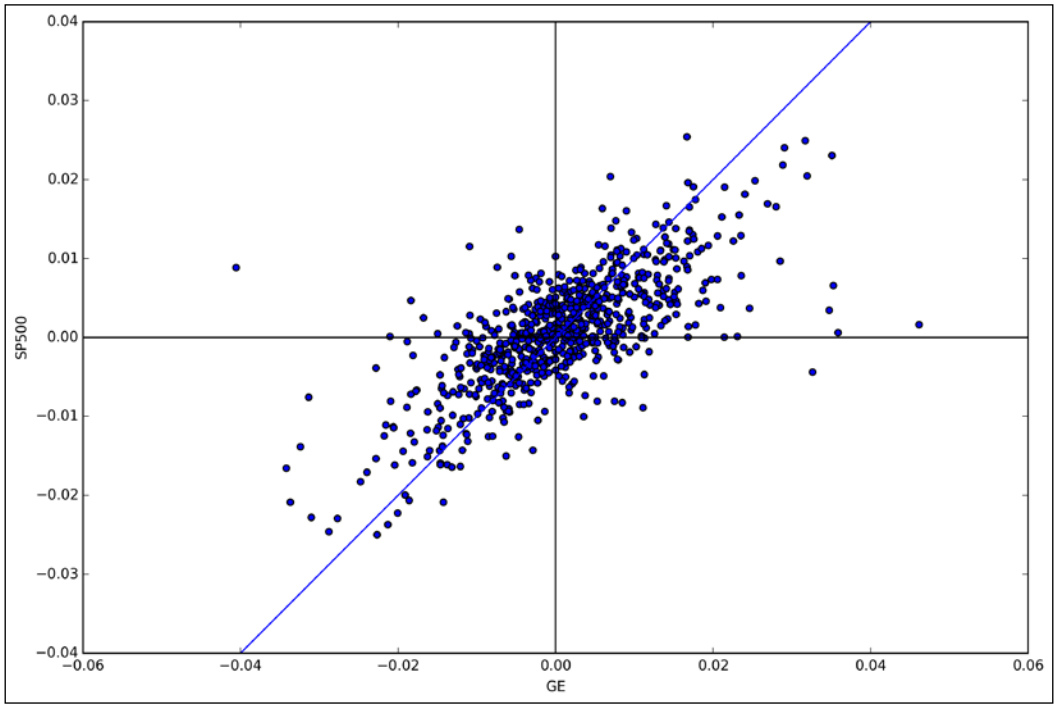
GE shows that it moved in the most similar way to the S&P 500, while UAL showed that it moved the least like the index. A plot of the returns shows this, as GE indeed follows right along the S&P 500, which was quite a good investment relative to the S&P 500 after March 2013:

```
In [55]:
   _ = cdr_all[['SP500', 'GE', 'UAL']].plot(figsize=(12,8));
```



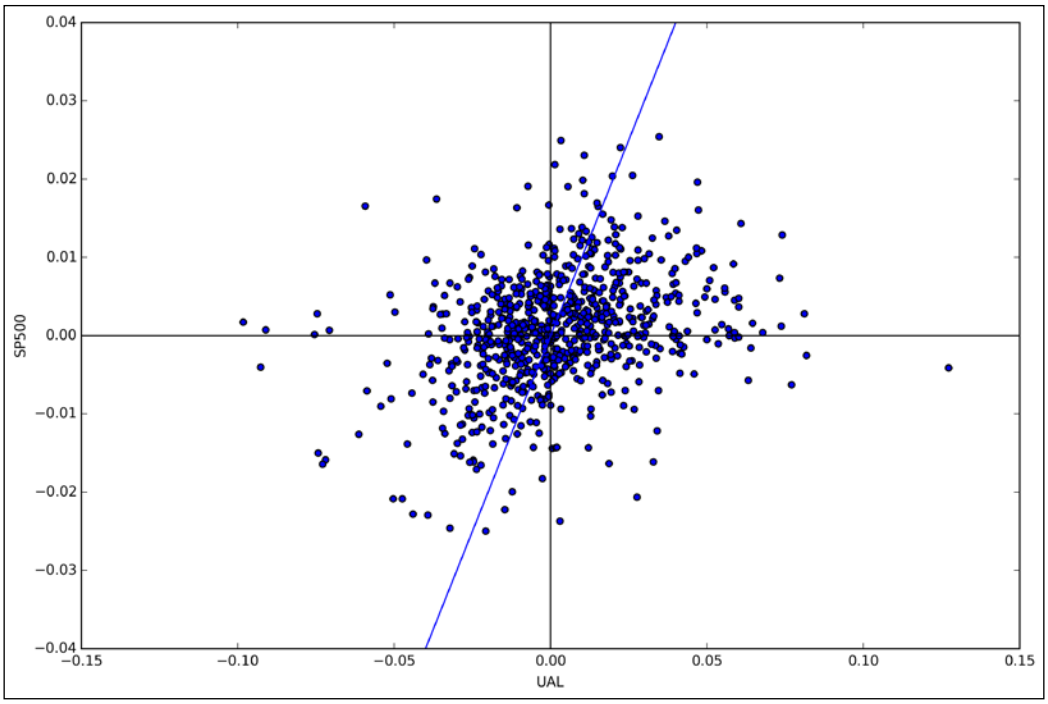
We can examine these conclusions with scatter plots of both against the S&P 500, as shown here:

```
In [56]:
   render_scatter_plot(dpc_all, 'GE', 'SP500')
   plt.savefig('5104_05_23.png', bbox_inches='tight', dpi=300)
```



Here's another plot:

```
In [57]:
   render_scatter_plot(dpc_all, 'UAL', 'SP500')
```



This shows that GE is fairly tightly correlated to the S&P 500. UAL has a much more distributed cluster of points around the origin, which supports that it has a lot less correlation.

Summary

In this chapter, we examined various means of performing the analysis of stock and index data. We started with loading historical quotes from Yahoo! Finance, moved on to how to create various visualizations for this data, and then to performing various financial analyses that are common for analyzing stock market data.

We focused in this chapter purely upon the analysis of historical data. We neither made any attempts to use this data to predict the future and to make decisions on trade execution, nor did we look at the management of portfolios, where historical data can be used to calculate optimal portfolios. We will come to both of these chapters later in the book, where the techniques in this chapter will be leveraged to help with those concepts.

6
Trading Using Google Trends

Several years ago, a paper titled *Quantifying Trading Behavior in Financial Markets Using Google Trends* was published in Scientific Reports. This paper asked this question: "Is it possible to predict efficient trading strategies based upon the frequency of certain words in Google searches?"

The authors went through a number of interesting steps involved in gathering data and performing analyses. They derived a set of financial keywords from the Financial Times website that they thought were good words for examining patterns of search for financial information.

With this, they built a robust set of keywords using Google Sets (which is now defunct). Using those keywords, they collected Google Trends data on those words over a period of years and defined a trading execution plan to buy or sell based upon the changes in the search history on all of their terms. They ran the strategies for all the keywords and ranked the value of their investments for each search phrase.

What was their conclusion? It appears to be very likely that this can be used to beat random investment in the S&P 500 index. This result is perhaps debatable, but the process itself is an interesting one to attempt using pandas. It demonstrates the bringing in of historical data from multiple sources and using the statistical analysis of one stream of data to make decisions on investing and valuating a portfolio based on another stream.

In this chapter, we will investigate by gathering much of the data that they collected and in reproducing their results as closely as possible (and we will be very close to their results). All of the steps that they performed can be simply reproduced using pandas. Together, they provide an excellent example of social data collection and how it can be applied to make money. They also present a very interesting introduction to using pandas to develop trading strategies, which this book will now turn its attention to and run with for its remainder.

We will go through the following topics in detail:

- A brief summary of Quantifying Trading Behavior in Financial Markets Using Google Trends
- Retrieving trend data from Google Trends
- Obtaining Dow Jones Index data from Quandl
- Generating trade orders
- Calculating investment results
- Conclusions

For your reference, the paper is available at `http://www.nature.com/srep/2013/130425/srep01684/full/srep01684.html`.

Notebook setup

The workbook and examples will all require the following code to execute and format output. It is similar to the previous chapters but also includes matplotlib imports to support many of the graphics that will be created, several options to fit data to the page in the text, the CSV (comma separated value) framework and the RE (regular expression) framework. Here's the code I am talking about:

```
In [1]:
   import pandas as pd
   import numpy as np
   import datetime as dt
   import matplotlib.pyplot as plt
   import pandas.io.data as web

   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', 78)
   pd.set_option('precision', 6)

   %matplotlib inline
```

A brief on Quantifying Trading Behavior in Financial Markets Using Google Trends

The authors of this paper state that financial markets are a prime target for investigating the prediction of market movements based upon the social habits of people searching for and gathering information to gain a competitive advantage in order to capture opportunities for personal financial gain.

They go on to investigate whether search query data from Google Trends can historically be used to provide insights into the information gathering process that leads up to making trading decisions in the stock market.

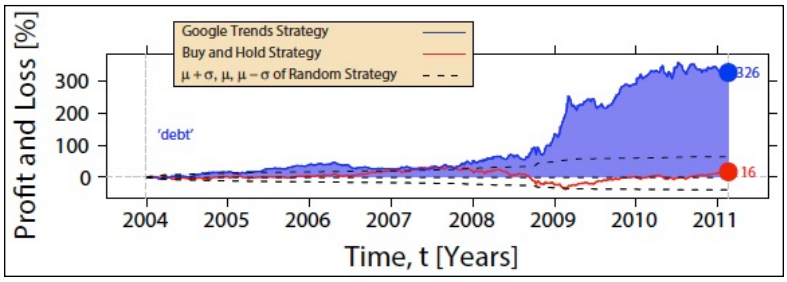
The authors gather data from Google Trends and **Dow Jones Industrial Average** (**DJIA**) for the period of `2004-01-01` to `2011-02-28`. They seed the process with some financial terms (specifically the term, debt) that can yield a bias towards the search for financial results. They take the initial set of terms and then build a larger set of terms using Google Sets to suggest more search terms based upon the seed terms. They decide upon using 98 different search terms and analyze trading decisions made upon the volume of the searches on each of those terms.

They use weekly DJIA closing values on the first trading day of the week, usually Monday but occasionally Tuesday. Google Trends data is reported on a Sunday through Saturday interval.

To be able to relate search behavior in Google to market movements, the authors correlate the volumes of search terms relative to the movement of the DJIA. The authors then propose a trading strategy, where, if the average number of searches for a term on Google has increased at the end of a three-week window, then there should be an upturn in the market the following week. Therefore, a trader should take a long position, transferring all current holdings into the newly identified position. Given the property prediction of market gains, a profit will be made over that one week period due to the increase in the value of the investment.

If the number of searches at the end of the three-week period has decreased from the previous three-week average, then we should go short and sell our holdings at the end of the first day of the next trading week, and then buy them back at the beginning of the next week. If the market moves down during this period, then we will have profited.

The authors also present an analysis of the performance of their strategy. This analysis is based upon the search term "debt" and how their position increased based upon their strategy. This is represented in the following graph by the solid blue line and shows they produced 326 percent:



The dashed lines represent the standard deviation of the cumulative return for a strategy that involves buying and selling financial instruments in an uncorrelated and random manner, and the results are derived from the simulation of 10,000 realizations of the random strategy. Their conclusion is that there is a significantly large enough difference in the results of their Google Trends strategy over the random strategies to determine that that is validity to their assertion.

They do neglect transaction fees, stating that their strategy only involves 104 transactions per year but could have an effect on the results if taken into account. They state that they ignored the transaction fees as their goal is to determine the overall effectiveness of relating social data to market movement to gain advantage as a trader.

In our analysis in this chapter, we will proceed with gathering Google Trends data for the search term "debt" for the same period of time as in the paper, along with DJIA data, and set up a model for replicating this investment strategy. Our goal is not to validate their research but to be able to learn various concepts and their implementation in Python with pandas. From this, you will learn valuable skills to obtain and relate data from disparate data feeds, model a trading strategy, and use a trading back-tester to evaluate the effectiveness of the strategies.

Data collection

Our goal will be to create a `DataFrame`, which contains both the authors' DJIA and Google Trends data combined with data that we also collect dynamically from the Web for each. We will check that our data conforms to what they had collected, and then we will use our data to simulate trades based upon their algorithm.

The data used in the study is available on the Internet. I have included it in the examples for the text. But we will also dynamically collect this information to demonstrate those processes using pandas. We will perform the analysis both on the data provided by the authors as well as our freshly collected data.

Unfortunately, but definitely not uncommon in the real world, we will also run into several snags in data collection that we need to work around. First, Yahoo! no longer provides DJIA data, so we can't fetch that data with the `DataReader` class of pandas. We will get around Yahoo! Finance no longer providing DJIA data using a web-based service named Quandl, which is a good service to also introduce to a reader of this text.

Second, Google Sets, used by the authors to derive their search terms, is now defunct—having been turned off by Google. That is actually disappointing, but we are just going to model results on the single search term "debt", which the authors claim had the best results.

Third, access to Google Trends data is, for lack of a better description, wonky. I will provide a `.csv` file that we will use, but we will also take time to discuss dynamically downloading the data.

The data from the paper

The data from the paper is available on the Internet, but I have also included it in the data folder of the code samples. It can be loaded into pandas using the following command:

```
In [2]:
   paper = pd.read_csv('PreisMoatStanley2013.dat',
                       delimiter=' ',
                       parse_dates=[0,1,100,101])
   paper[:5]

Out[2]:
     Google Start Date Google End Date     arts  banking        ... \
   0        2004-01-04      2004-01-10  0.95667  0.19333        ...
   1        2004-01-11      2004-01-17  0.97000  0.20333        ...
   2        2004-01-18      2004-01-24  0.92667  0.19667        ...
   3        2004-01-25      2004-01-31  0.95000  0.19667        ...
   4        2004-02-01      2004-02-07  0.89333  0.20333        ...
```

```
        water    world   DJIA Date  DJIA Closing Price
   0  1.91333  4.83333  2004-01-12            10485.18
   1  1.93333  4.76667  2004-01-20            10528.66
   2  1.89333  4.60000  2004-01-26            10702.51
   3  1.92000  4.53333  2004-02-02            10499.18
   4  1.88667  4.53333  2004-02-09            10579.03

   [5 rows x 102 columns]
```

The data from the paper is a single file containing all of the DJIA data combined with a normalization of the search volume for each of their 98 keywords. Each keyword used for a search is represented as a column.

We want to extract from each row the values in the debt column, the Google Trends Week End date, and the closing price and date for the DJIA. We can do this with the following pandas code:

```
In [3]:
   data = pd.DataFrame({'GoogleWE': paper['Google End Date'],
                        'debt': paper['debt'].astype(np.float64),
                        'DJIADate': paper['DJIA Date'],
                        'DJIAClose': paper['DJIA Closing Price']
                        .astype(np.float64)})
   data[:5]

Out[3]:
     DJIAClose   DJIADate   GoogleWE     debt
   0  10485.18 2004-01-12 2004-01-10  0.21000
   1  10528.66 2004-01-20 2004-01-17  0.21000
   2  10702.51 2004-01-26 2004-01-24  0.21000
   3  10499.18 2004-02-02 2004-01-31  0.21333
   4  10579.03 2004-02-09 2004-02-07  0.20000
```

The paper's Google Trends data has been normalized relative to all of their resulting searches. We will see the raw values when we get this data on our own. The important thing with this data is not actually the value but the change in value over time, which can be used to represent the relative change in search volumes for the given period.

Gathering our own DJIA data from Quandl

It has historically been possible to retrieve DJIA data from Yahoo! Finance using the pandas `DataReader` class. Unfortunately, Yahoo! has stopped providing DJIA data, so we need an alternative to get this data. We can retrieve this data from Quandl (`https://www.quandl.com/`). Quandl is a provider of datasets specifically related to quantitative analysis. An account can be created for free, and they provide API-based access to their data. They also provide client libraries for multiple languages, including Python, Java, and C#. In *Chapter 1*, *Getting Started with pandas Using Wakari.io*, we added their library to our Python environment. Here's the command for this discussion:

```
In [4]:
   import Quandl
   djia = Quandl.get("YAHOO/INDEX_DJI",
                     trim_start='2004-01-01',
                     trim_end='2011-03-05')
```

Alternatively, you can load this data from a file provided with the code packet for the text:

```
In [5]:
   # djia = pd.read_csv("djia.csv", index_col=0)
```

The following command gives us an overview of the data that was retrieved. It is a set of daily variables from the DJIA between and including the specified dates:

```
In [6]:
   djia[:3]


Out[6]:
                  Open    High     Low   Close  Volume  Adjusted Close
   Date
   2004-01-02 10452.7 10527.0 10384.3 10409.9 1688900         10409.9
   2004-01-05 10411.9 10544.1 10411.9 10544.1 2212900         10544.1
   2004-01-06 10543.9 10549.2 10499.9 10538.7 1914600         10538.7
```

We would now like to merge the `Close` values in this data into our `DataFrame`. We will want to do this by aligning our dates with the data in the `DJIADate` column. We also want to drop all of the days the data of which does not align. We can do this simply with a pandas merge. To perform this, we first need to extract the `Close` values and move the dates from the index to a column, as shown here:

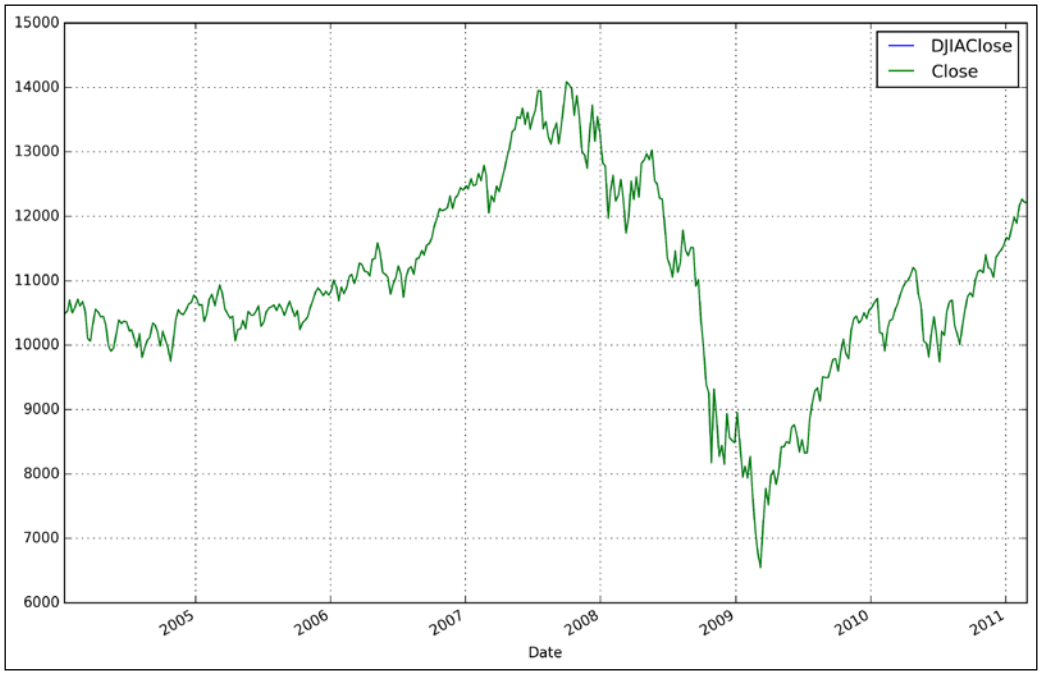
```
In [7]:
   djia_closes = djia['Close'].reset_index()
   djia_closes[:3]


Out[7]:
           Date    Close
   0 2004-01-02  10409.9
   1 2004-01-05  10544.1
   2 2004-01-06  10538.7
```

Now, we will create a new `DataFrame` object with the two datasets merged based upon the `DJIADate` and `Date` columns from the two respective `DataFrame` objects (we drop the `DJIADate` column from the result as it is redundant and set `Date` to be the index), as follows:

```
In [8]:
   data = pd.merge(data, djia_closes,
                   left_on='DJIADate', right_on='Date')
   data.drop(['DJIADate'], inplace=True, axis=1)
   data = data.set_index('Date')
   data[:3]


Out[8]:
              DJIAClose   GoogleWE  debt    Close
   Date
   2004-01-12  10485.18 2004-01-10  0.21  10485.2
   2004-01-20  10528.66 2004-01-17  0.21  10528.7
   2004-01-26  10702.51 2004-01-24  0.21  10702.5
```

Upon examining this data, it is evident that there is a fairly good match between the DJIA closing prices. If we plot both series next to each other, we will see that they are practically identical. The following is the command to plot the data:

```
In [17]:
   data[['DJIAClose', 'Close']].plot(figsize=(12,8));
```



We can also check the statistics of the differences in the values, as shown here:

```
In [10]:
   (data['DJIAClose']-data['Close']).describe()

Out[10]:
   count    371.00000
   count    373.00000
   mean      -0.00493
   std        0.03003
   min       -0.05000
   25%       -0.03000
```

```
50%       -0.01000
75%        0.02000
max        0.04000
dtype: float64
```

The overall differences appear to be well less than one-tenth of a point and seem likely to be just from rounding errors.

One final check can examine the correlation of the two series of data:

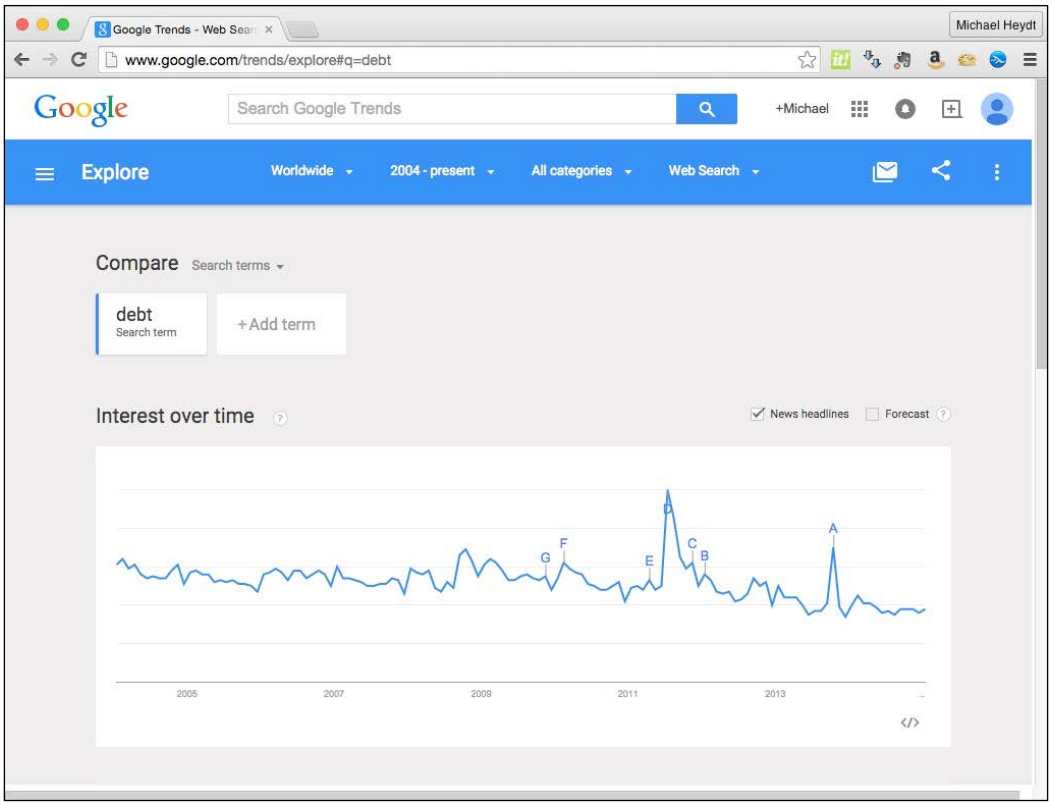
```
In [11]:
   data[['DJIAClose', 'Close']].corr()


Out[11]:
              DJIAClose  Close
   DJIAClose          1      1
   Close              1      1
```

There is a perfect positive correlation. With this summarizing performed, we can have strong confidence that our data is prepared properly.

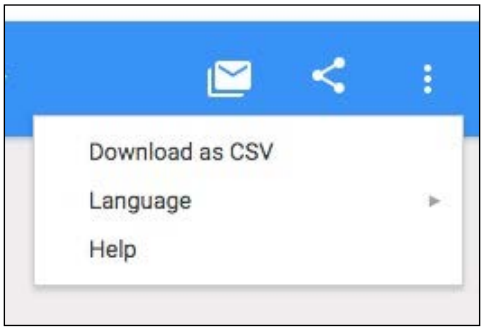
Google Trends data

The authors provided their own version of Google Trends data for the term "debt". This is convenient, but we want to get our own Google Trends data. Unfortunately, there does not currently appear to be any API access to this data. But there are several ways that we can go about retrieving it. One way is to use the mechanize framework to automate a web-crawling process. Another way, which is what we will do, is to use the web portal to download the CSV for the data we want.

You can search in your browser for any term and get the associated trend data at `http://www.google.com/trends/`. The following command demonstrates the result of searching for the term `debt`:



This is pretty but not usable in our pandas application. Fortunately, if we go to the options button, we see the option **Download as CSV**, as shown in the following screenshot:



You can download this for yourself. The data is also provided in the `data` folder of the samples for the book. The name of the file is `trends_report_debt.csv`. The following command shows the contents of the first few lines of the file:

```
In [12]:
   !head trends_report_debt.csv
   # type trends_report_debt.csv # on windows

   Web Search interest: debt
   United States; Jan 2004 - Feb 2011

   Interest over time
   Week,debt
   2004-01-04 - 2004-01-10,63
   2004-01-11 - 2004-01-17,60
   2004-01-18 - 2004-01-24,61
   2004-01-25 - 2004-01-31,63
   2004-02-01 - 2004-02-07,61
```

This is not a particularly friendly CSV file, and we need to do a little bit of processing to extract the data properly. The following command reads the file and selects trend data in the range of dates we are working with:

```
In [13]:
   from StringIO import StringIO
   with open("trends_report_debt.csv") as f:
       data_section = f.read().split('\n\n')[1]
       trends_data = pd.read_csv(
           StringIO(data_section),
           header=1, index_col='Week',
           converters={
               'Week': lambda x: pd.to_datetime(x.split(' ')[-1])
           }
       )
   our_debt_trends = trends_data['2004-01-01':'2011-02-28'] \
                                 .reset_index()
```

```
   our_debt_trends[:5]
Out[13]:
           Week  debt
   0 2004-01-10    63
   1 2004-01-17    60
   2 2004-01-24    61
   3 2004-01-31    63
   4 2004-02-07    61
```

The numbers do not represent an actual count of the number of searches. It is simply a number provided by Google that you can use to compare with the other periods in the dataset to get a sense of how the volume changes. I'm sorry about the fact that they keep the good data to themselves, but there is enough here for us to work with.

We can start by combining this data into our dataset and check how well they conform. We will do the same as before and use `pd.merge()`. This time, we will join on the `GoogleWE` column on the left and the `Week` column on the right. The following command performs the merge, renames debt columns, and moves the indexes around:

```
In [14]:
   final = pd.merge(data.reset_index(), our_debt_trends,
                    left_on='GoogleWE', right_on='Week',
                    suffixes=['_P', '_O'])
   final.drop('Week', inplace=True, axis=1)
   final.set_index('Date', inplace=True)
   final[:5]


Out[14]:
               DJIAClose   GoogleWE    debtP    Close   debtO
   Date
   2004-01-12   10485.18 2004-01-10  0.21000  10485.2      63
   2004-01-20   10528.66 2004-01-17  0.21000  10528.7      60
   2004-01-26   10702.51 2004-01-24  0.21000  10702.5      61
   2004-02-02   10499.18 2004-01-31  0.21333  10499.2      63
   2004-02-09   10579.03 2004-02-07  0.20000  10579.0      61
```

We can create a new `DataFrame` with the normalized trend data from both the paper and our trend data (indexed by `GoogleWE`) and check to see how closely our trend data correlates with that used in the paper:

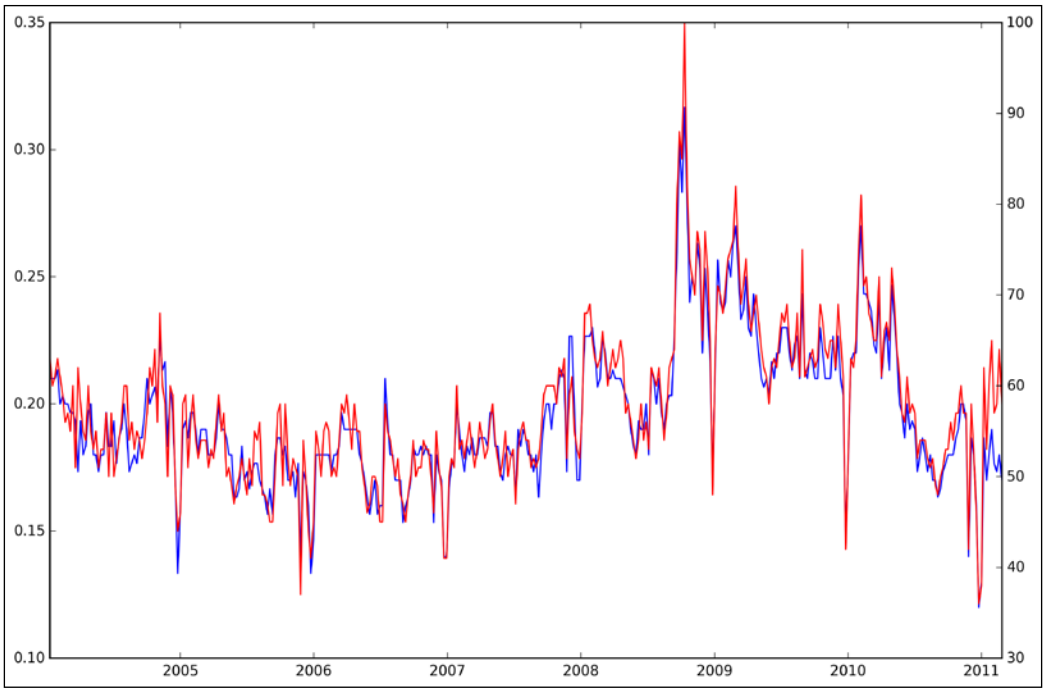
```
In [15]:
   combined_trends = final[['GoogleWE', 'debtP', 'debtO']] \
                           .set_index('GoogleWE')
   combined_trends[:5]


Out[15]:
                     DebtP      debtO
   GoogleWE
   2004-01-10     0.21000         63
   2004-01-17     0.21000         60
   2004-01-24     0.21000         61
   2004-01-31     0.21333         63
   2004-02-07     0.20000         61
```

A correlation between these series shows that they are highly correlated. There is some difference as the data retrieved from Google is constantly renormalized and will cause small differences in the trend data that was captured earlier:

```
In [16]:
   combined_trends.corr()


Out[16]:
              debtP    debtO
   debtP   1.00000  0.95766
   debtO   0.95766  1.00000
```

Plotting these two against each other, we can see they are very closely correlated:

```
In [17]:
   fig, ax1 = plt.subplots(figsize=(12,8))
   ax1.plot(combined_trends.index,
            combined_trends.debtP, color='b')
   ax2 = ax1.twinx()
   ax2.plot(combined_trends.index,
            combined_trends.debtO, color='r')
   plt.show()
```



Generating order signals

In the trading strategy that we will define, we want to be able to decide whether there is enough movement in the volume of searches on `debt` to go and execute a trade in the market that will make us a profit. The paper defines this threshold as though there is a higher search volume at the end of a Google Trends week than in the previous three-week average of the search volume, and then we will go short. If there is a decline, we will go long the following week.

The first thing we will need to do is reorganize our data by moving the `GoogleWE` dates into the index. We are going to make our decisions based upon these week-ending dates and use the `Close` price in an associated record as the basis for our trade as that price represents the `Close` price at the beginning of the next week. We also drop the `DJIAClose` column as it is redundant with `Close`:

```
In [18]:
   base = final.reset_index().set_index('GoogleWE')
   base.drop(['DJIAClose'], inplace=True, axis=1)
   base[:3]
```

```
Out[18]:
                    Date  debtP    Close  debtO
   GoogleWE
   2004-01-10 2004-01-12   0.21  10485.2     63
   2004-01-17 2004-01-20   0.21  10528.7     60
   2004-01-24 2004-01-26   0.21  10702.5     61
```

We now need to calculate the moving average of the previous three weeks for each week. This is easily performed with pandas, and the following command will compute the moving average for both the trends provided in the paper and the data we just collected from Google Trends:

```
In [19]:
   base['PMA'] = pd.rolling_mean(base.debtP.shift(1), 3)
   base['OMA'] = pd.rolling_mean(base.debtO.shift(1), 3)
   base[:5]


Out[19]:
                    Date    debtP    Close  debtO      PMA       OMA
   GoogleWE
   2004-01-10 2004-01-12  0.21000  10485.2     63      NaN       NaN
   2004-01-17 2004-01-20  0.21000  10528.7     60      NaN       NaN
   2004-01-24 2004-01-26  0.21000  10702.5     61      NaN       NaN
   2004-01-31 2004-02-02  0.21333  10499.2     63  0.21000  61.33333
   2004-02-07 2004-02-09  0.20000  10579.0     61  0.21111  61.33333
```

The code shifts the calculated values by one week. This is because we need the three previous weeks' rolling mean at each week. Not shifting would include the current week in the average, and we want to make decisions on the prior three.

We now need to make a decision on how to execute based upon this information. This is referred to as generating order signals. The current organization of the data makes this simple to perform as we need to simply subtract the moving average from the current trend value for each week. If there is a decrease, we assign `1` as a value, and we assign `-1` as a value in the opposite situation:

```
In [20]:
   base['signal0'] = 0 # default to 0
   base.loc[base.debtP > base.PMA, 'signal0'] = -1
   base.loc[base.debtP < base.PMA, 'signal0'] = 1
   base['signal1'] = 0
```

```
base.loc[base.debtO > base.OMA, 'signal1'] = -1
base.loc[base.debtO < base.OMA, 'signal1'] = 1
base[['debtP', 'PMA', 'signal0', 'debtO', 'OMA', 'signal1']]
```

```
Out[20]:
                debtP      PMA  signal0  debtO       OMA  signal1
GoogleWE
2004-01-10  0.21000      NaN        0     63       NaN        0
2004-01-17  0.21000      NaN        0     60       NaN        0
2004-01-24  0.21000      NaN        0     61       NaN        0
2004-01-31  0.21333  0.21000       -1     63  61.33333       -1
2004-02-07  0.20000  0.21111        1     61  61.33333        1
...             ...      ...      ...    ...       ...      ...
2011-01-29  0.19000  0.17889       -1     65  58.33333       -1
2011-02-05  0.17667  0.18000        1     57  59.33333        1
2011-02-12  0.17333  0.18222        1     58  60.66667        1
2011-02-19  0.18000  0.18000        1     64  60.00000       -1
2011-02-26  0.17000  0.17667        1     58  59.66667        1

[373 rows x 6 columns]
```

The trade signals based on our data are very similar but have slight differences due to the difference in the normalization of the data.

Computing returns

Every week, we will reinvest the entirety of our portfolio. Because of this, the return on the investment over the week will be reflected simply by the percentage change in the DJIA between the close of the first Monday and the close of the following Monday, but with the factor taken into account on whether we went short or long.

We have already accounted for going short or long using `-1` or `1` for the signal, respectively. Now, we just need to calculate the percentage change, shift it by one week back in time, and multiply it by the signal value. We shift the percentage change back one week as we want to multiply the signal value for the current week by the next percentage change from the next week:

```
In [21]:
   base['PctChg'] = base.Close.pct_change().shift(-1)
   base[['Close', 'PctChg', 'signal0', 'signal1']][:5]

Out[21]:
                Close   PctChg  signal0  signal1
   GoogleWE
   2004-01-10  10485.2  0.00415        0        0
   2004-01-17  10528.7  0.01651        0        0
   2004-01-24  10702.5 -0.01900        0        0
   2004-01-31  10499.2  0.00760       -1       -1
   2004-02-07  10579.0  0.01285        1        1
```

To calculate the returns gained each week, we simply multiply the signal value by the percentage change (we will do this for both signals for both sets of trends):

```
In [22]:
   base['ret0'] = base.PctChg * base.signal0
   base['ret1'] = base.PctChg * base.signal1
   base[['Close', 'PctChg', 'signal0', 'signal1',
         'ret0', 'ret1']][:5]

Out[22]:
                Close   PctChg  signal0  signal1     ret0     ret1
   GoogleWE
   2004-01-10  10485.2  0.00415        0        0  0.00000  0.00000
   2004-01-17  10528.7  0.01651        0        0  0.00000  0.00000
   2004-01-24  10702.5 -0.01900        0        0 -0.00000 -0.00000
   2004-01-31  10499.2  0.00760       -1       -1 -0.00760 -0.00760
   2004-02-07  10579.0  0.01285        1        1  0.01285  0.01285
```

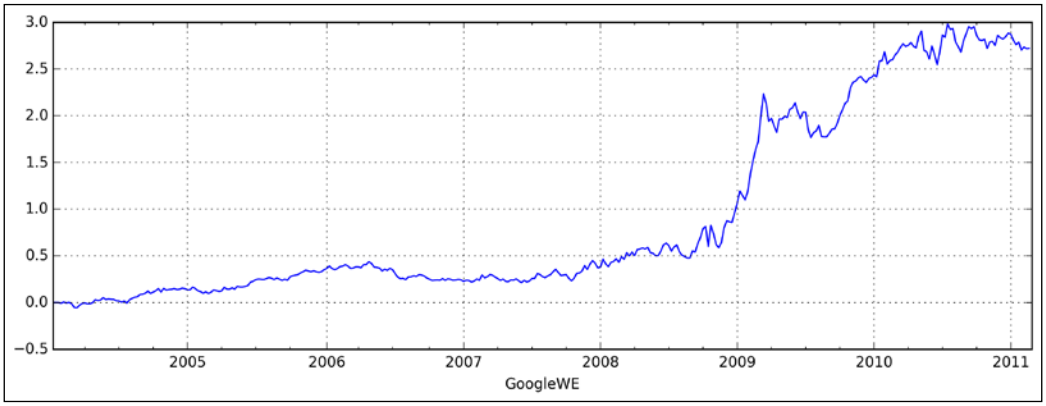
Cumulative returns and the result of the strategy

We now have all the weekly returns based upon our strategy. We can calculate the overall net percentage return of the investments at the end by applying the cumulative product of `1 + base.ret0` (the return of each week) and then subtracting `1` from the cumulative product:

```
In [23]:
   base['cumret0'] = (1 + base.ret0).cumprod() - 1
   base['cumret1'] = (1 + base.ret1).cumprod() - 1
   base[['cumret0', 'cumret1']]


Out[23]:
               cumret0  cumret1
   GoogleWE
   2004-01-10  0.00000  0.00000
   2004-01-17  0.00000  0.00000
   2004-01-24  0.00000  0.00000
   2004-01-31 -0.00760 -0.00760
   2004-02-07  0.00515  0.00515
   ...             ...      ...
   2011-01-29  2.70149  0.84652
   2011-02-05  2.73394  0.86271
   2011-02-12  2.71707  0.85430
   2011-02-19  2.72118  0.85225
   2011-02-26      NaN      NaN
   [373 rows x 2 columns]
```

At the end of our run of this strategy, we can see that we have made a profit. We now plot the returns based on the data from the paper:

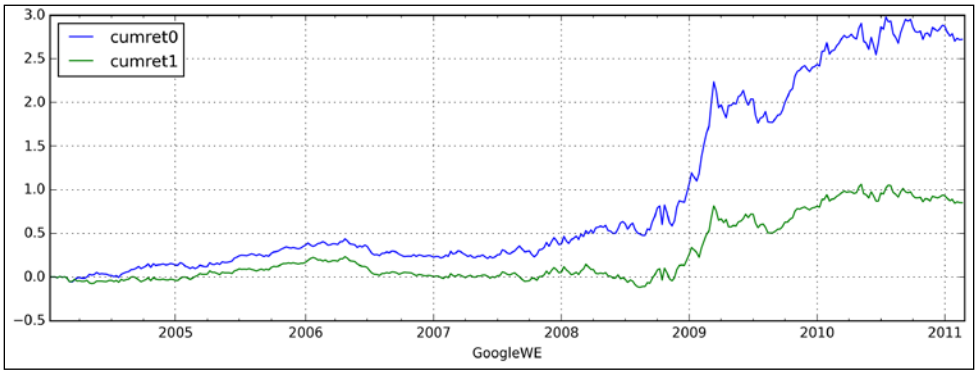
```
In [24]:
base['cumret0'].plot(figsize=(12,4));
```



This shows that our implementation of the strategy produces a very similar result to that published in the paper. We only obtained a 271 percent increase in value compared to their stated return of 326 percent, but the curve almost identically follows the same path as theirs. This suggests that our strategy executes similarly to theirs although there must be some slight differences in the calculations of some of the decisions. The important thing is you learned a number of concepts in pandas.

When we used our own data from Google Trends, we still had gains but not to the extent they received using their trend data. We now plot the two sets of data using the following command:

```
In [25]:
    data[['cumret0', 'cumret1']].plot(figsize=(12,4));
```



Although the trend data we received from Google followed very similar paths, small differences in the data can be seen when examining the order signals that will make the accumulation of returns more modest. This is likely to be because there is an overall sensitivity around where the total change in volume of the search term is very close to `0`, but the strategy still decides to execute one way or the other and causes returns to not grow as rapidly. But again, we are not analyzing the correctness of their results but seeing whether we can replicate the process and decision making using pandas.

Summary

In this chapter, we took an in-depth look at collecting a type of social data and using it to see whether we could identify trends in the data that can be correlated with market movements in order to gain an advantage over the general movement of the market. We did this by reproducing results from a published paper, which concludes that it is possible. We were able to reproduce very similar results and you learned a general process of analyzing data and making decisions on trading in the market.

The best part of this is that during this process, you saw that pandas provides a very robust framework for financial time-series analysis as well as for the analysis of simple social data. You learned how to work with multiple time-series that have different frequencies and how to manipulate them to be able to have frequencies that can be aligned to be able to apply decisions made in one to execution in another. This included various concepts such as frequency conversion, grouping by year and day of week, generating signals based upon data, and shifting calculations back and forth to align properly to relate data at different periods to each other with simple pandas formulas.

But there is also a lot that we did not cover. Our strategy was based purely on historical data and can lead to a look-ahead bias. It also did not cover the effects that our trades may have on the actual market. It did not factor in transaction costs. Perhaps most significantly, we did not perform the simulation of alternative strategies.

In the upcoming chapters, we will dive into each of these concepts (and more). In the next chapter on algorithmic trading, we will start to look at more elaborate strategies for investing in the market where we do not have perfect knowledge, need to learn on the fly, and make decisions based on imperfect data.

7
Algorithmic Trading

In this chapter, we will examine how to use pandas and a library known as **Zipline** to develop automated trading algorithms. Zipline (`http://www.zipline.io/`) is a Python-based algorithmic trading library. It provides event-driven approximations of live-trading systems. It is currently used in production as the trading engine that powers Quantopian (`https://www.quantopian.com/`), a free, community-centered platform for collaborating on the development of trading algorithms with a web browser.

We previously simulated trading based on a historical review of social and stock data, but these examples were naive in that they glossed over many facets of real trading, such as transaction fees, commissions, and slippage, among many others. Zipline provides robust capabilities to include these factors in the trading model.

Zipline also provides a facility referred to as backtesting. Backtesting is the ability to run an algorithm on historical data to determine the effectiveness of the decisions made on actual market data. This can be used to vet the algorithm and compare it to others in an effort to determine the best trading decisions for your situation.

We will examine three specific and fundamental trading algorithms: simple crossover, dual moving average crossover, and pairs trade. We will first look at how these algorithms operate and make decisions, and then we will actually implement these using Zipline and execute and analyze them on historical data.

This chapter will cover the following topics in detail:

- The process of algorithmic trading
- Momentum and mean-reversion strategies
- Moving averages and their significance in automated decision making
- Simple and exponentially weighted moving averages
- Common algorithms used in algorithmic trading

- Crossovers, including simple and dual moving average crossovers
- Pairs trading strategies
- Implementing dual moving crossover and pairs trading algorithms in Zipline

Notebook setup

The Notebook and examples will all require the following code to execute and format output. Later in the chapter, we will import the Zipline package but only after first discussing how to install it in your Python environment:

```
In [1]:
   import pandas as pd
   import pandas.io.data as web
   import numpy as np
   from datetime import datetime
   import matplotlib.pyplot as plt
   %matplotlib inline

   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', 78)
   pd.set_option('precision', 6)
```

The process of algorithmic trading

Algorithmic trading is the use of an automated system to execute trades in a market. These trades are executed in a predetermined manner using one or more algorithms and without human interaction. In this chapter, we will examine several common trading algorithms, along with tools that you can use in combination with pandas to determine the effectiveness of your trading algorithms.

Financial markets move in cycles. Proper identification of the movement of the market can lead to opportunities for profit by making appropriate and timely buys or sells of financial instruments. There are two broad categories for predicting movement in the market, which we will examine in this chapter: momentum strategies and mean-reversion strategies.

Momentum strategies

In momentum trading, trading focuses on stocks that are moving in a specific direction on high volume, measuring the rate of change in price changes. It is typically measured by continuously computing price differences at fixed time intervals. Momentum is a useful indicator of the strength or weakness of the price although it is typically more useful during rising markets as they occur more frequently than falling markets; therefore, momentum-based prediction gives better results in a rising market.

Mean-reversion strategies

Mean reversion is a theory in trading that prices and returns will eventually move back towards the mean of the stock or of another historical average, such as the growth of the economy or an industry average. When the market price is below the average price, a stock is considered attractive for purchase as it is expected that the price will rise and, hence, a profit can be made by buying and holding the stock as it rises and then selling at its peak. If the current market price is above the mean, the expectation is the price will fall and there is potential for profit in shorting the stock.

Moving averages

Whether using a momentum or mean-reversion strategy for trading, the analyses will, in one form or another, utilize moving averages of the closing price of stocks. We have seen these before when we looked at calculating a rolling mean. We will now examine several different forms of rolling means and cover several concepts that are important to use in order to make trading decisions based upon how one or more means move over time:

- Simple moving average
- Exponential moving average

Simple moving average

A moving average is a technical analysis technique that smooths price data by calculating a constantly updated average price. This average is taken over a specific period of time, ranging from minutes, to days, weeks, and months. The period selected depends on the type of movement of interest, such as making a decision on short-term, medium-term, or long-term investment.

Moving averages give us a means to relate the price data to determine a trend indicator. A moving average does not predict price direction but instead gives us a means of determining the direction of the price with a lag, which is the size of the window.

In financial markets, a moving average can be considered support in a rising market and resistance in a falling market.

For more info on support and resistance, visit `http://www.investopedia.com/articles/technical/061801.asp`.

To demonstrate this, take a look at the closing price of MSFT for 2014 related to its 7-day, 30-day, and 120-day rolling means during the same period:

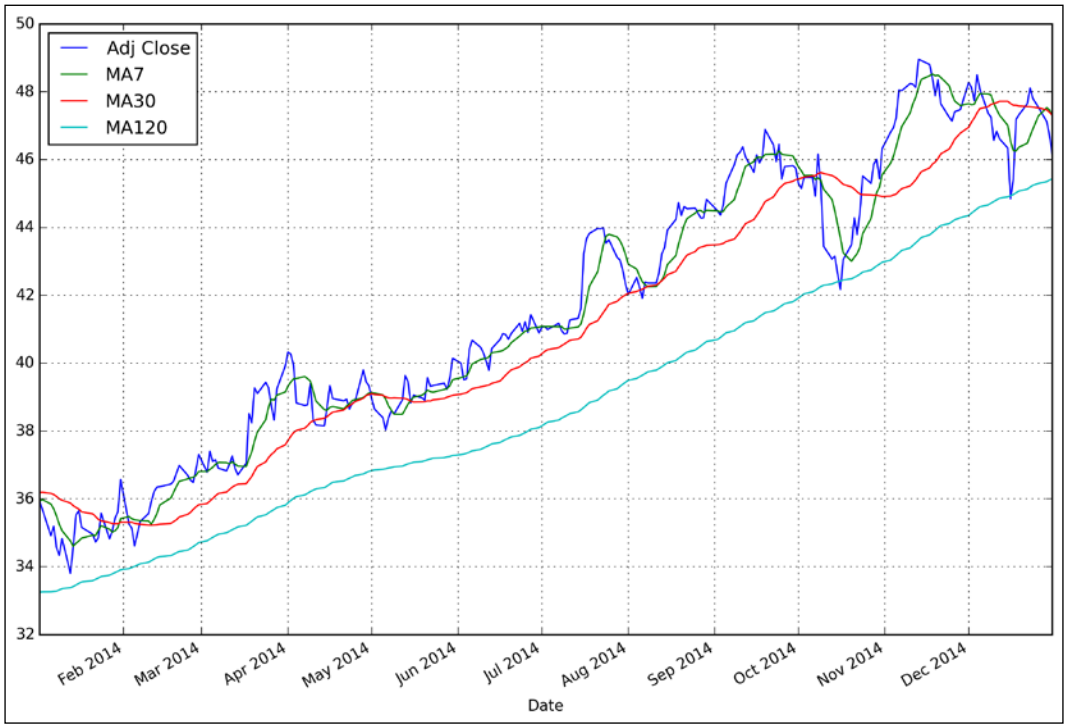
```
In [2]:
   msft = web.DataReader("MSFT", "yahoo",
                         datetime(2000, 1, 1),
                         datetime(2014, 12, 31))
   msft[:5]


Out[2]:
                  Open    High     Low   Close    Volume Adj Close
   Date
   2000-01-03  117.38  118.62  112.00  116.56  53228400     41.77
   2000-01-04  113.56  117.12  112.25  112.62  54119000     40.36
   2000-01-05  111.12  116.38  109.38  113.81  64059600     40.78
   2000-01-06  112.19  113.88  108.38  110.00  54976600     39.41
   2000-01-07  108.62  112.25  107.31  111.44  62013600     39.93
```

Now, we can calculate the rolling means using `pd.rolling_mean()`:

```
In [3]:
   msft['MA7'] = pd.rolling_mean(msft['Adj Close'], 7)
   msft['MA30'] = pd.rolling_mean(msft['Adj Close'], 30)
   msft['MA90'] = pd.rolling_mean(msft['Adj Close'], 90)
   msft['MA120'] = pd.rolling_mean(msft['Adj Close'], 120)
```

Then, we plot the price versus various rolling means to see this concept of support:

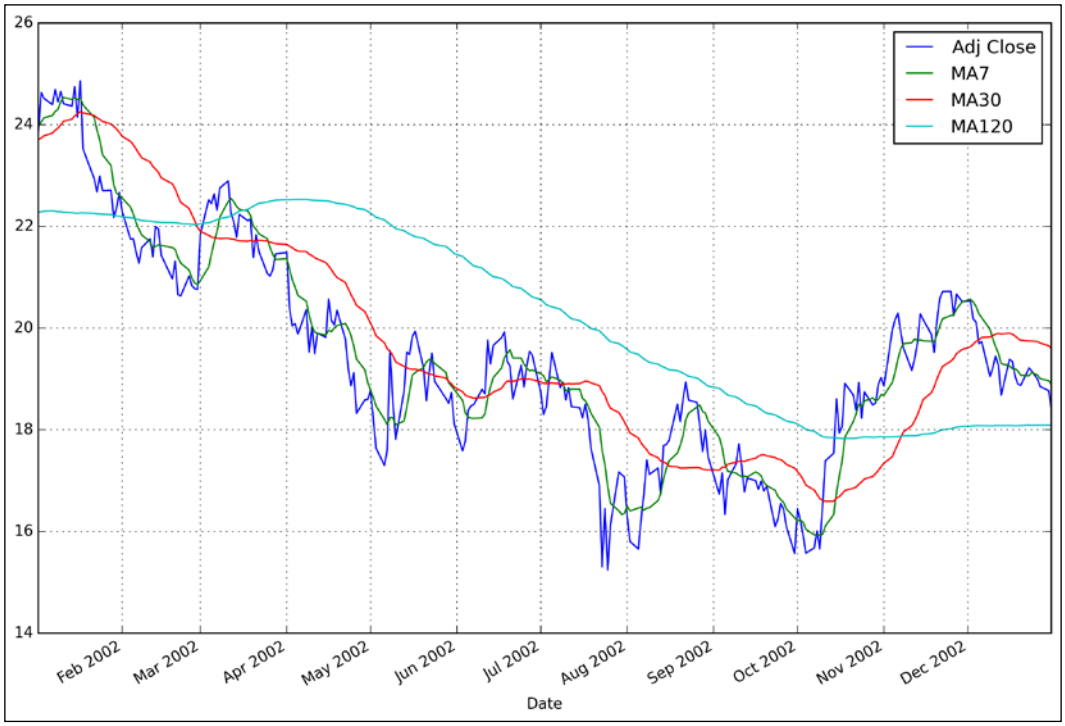
```
In [4]:
   msft['2014'][['Adj Close', 'MAm7',
                 'MA30', 'MA120']].plot(figsize=(12,8));
```



The price of MSFT had a progressive rise over 2014, and the 120-day rolling mean has functioned as a floor/support, where the price bounces off this floor as it approaches it. The longer the window of the rolling mean, the lower and smoother the floor will be in an uptrending market.

Contrast this with the price of the stock in 2002, when it had a steady decrease in value:

```
In [5]:
   msft['2014'][['Adj Close', 'MA7',
                 'MA30', 'MA120']].plot(figsize=(12,8));
```



In this situation, the 120-day moving average functions as a ceiling for about 9 months. This ceiling is referred to as resistance as it tends to push prices down as they rise up towards this ceiling.

The price does not always respect the moving average. In both of these cases, the prices have crossed over the moving average, and, at times, it has reversed its movement slightly before or just after crossing the average.

In general, though, if the price is above a particular moving average, then it can be said that the trend for that stock is up relative to that average and when the price is below a particular moving average, the trend is down.

The means of calculating the moving average used in the previous example is considered a **simple moving average** (**SMA**). The example demonstrated calculated the 7, 30, and 120 SMA values.

While valuable and used to form the basis of other technical analyses, simple moving averages have several drawbacks. They are listed as follows:

- The shorter the window used, the more the noise in the signal feeds into the result
- Even though it uses actual data, it is lagging behind it by the size of the window
- It never reaches the peaks or valleys of the actual data as it is smoothing the data
- It does not tell you anything about the future
- The average calculated at the end of the window can be significantly skewed by the values earlier in the window that are significantly skewed from the mean

To help address some of these concerns, it is common to instead use an exponentially weighted moving average.

Exponentially weighted moving average

Exponential moving averages reduce the lag and effect of exceptional values early in a window by applying more weight to recent prices. The amount of weighting applied to the most recent price depends on the number of periods in the moving average and how the exponential function is formulated.

In general, the weighted moving average is calculated using the following formula:

$$y_t = \frac{\sum_{i=0}^{t} w_i x_{t-i}}{\sum_{i=0}^{t} w_i}$$

In the preceding formula, x_t is the input and y_t is the result.

The EW functions in pandas support two variants of exponential weights: The default, `adjust=True`, uses the following weights:

$$w_i = (1-\alpha)^i w_i = (1-\alpha)^i$$

When `adjust=False` is specified, moving averages are calculated using the following formula:

$$y_0 = x_0$$

The preceding formula is followed by this formula:

$$y_t = (1-\alpha)y_{t-1} + \alpha x_t$$

This is equivalent to using weights:

$$w_i = \begin{cases} \alpha(1-\alpha)^i & if\ i < t \\ (1-\alpha)^i & if\ i = t. \end{cases}$$

However, instead of dealing with these formulas as described, pandas takes a slightly different approach to specifying the weighting. Instead of specifying an alpha between 0 and 1, pandas attempts to make the process less abstract by letting you specify alpha in terms of either *span*, *center of mass*, or *half life*:

$$\alpha = \begin{cases} \frac{2}{s+1} & s = span \\ \frac{1}{1+c} & c = center\ of\ mass \\ 1 - exp^{\frac{\log 0.5}{h}} & h = half\ life \end{cases}$$

One must specify precisely one of the three values to the `pd.ewma()` function at which point pandas will use the corresponding formulation for alpha.

As an example, a span of 10 corresponds to what is commonly referred to as a 10-day exponentially weighted moving average. The following command demonstrates the calculation of the percentage weights that will be used for each data point in a 10-span EWMA (`alpha=0.18181818`):

```
In [6]:
   periods = 10
   alpha = 2.0/(periods +1)
   factors = (1-alpha) ** np.arange(1, 11)
   sum_factors = factors.sum()
   weights = factors/sum_factors
   weights


Out[6]:
   array([ 0.21005616,  0.17186413,  0.14061611,  0.11504954,
   0.09413145, 0.07701664,  0.06301361,  0.05155659,  0.04218267,
   0.03451309])
```

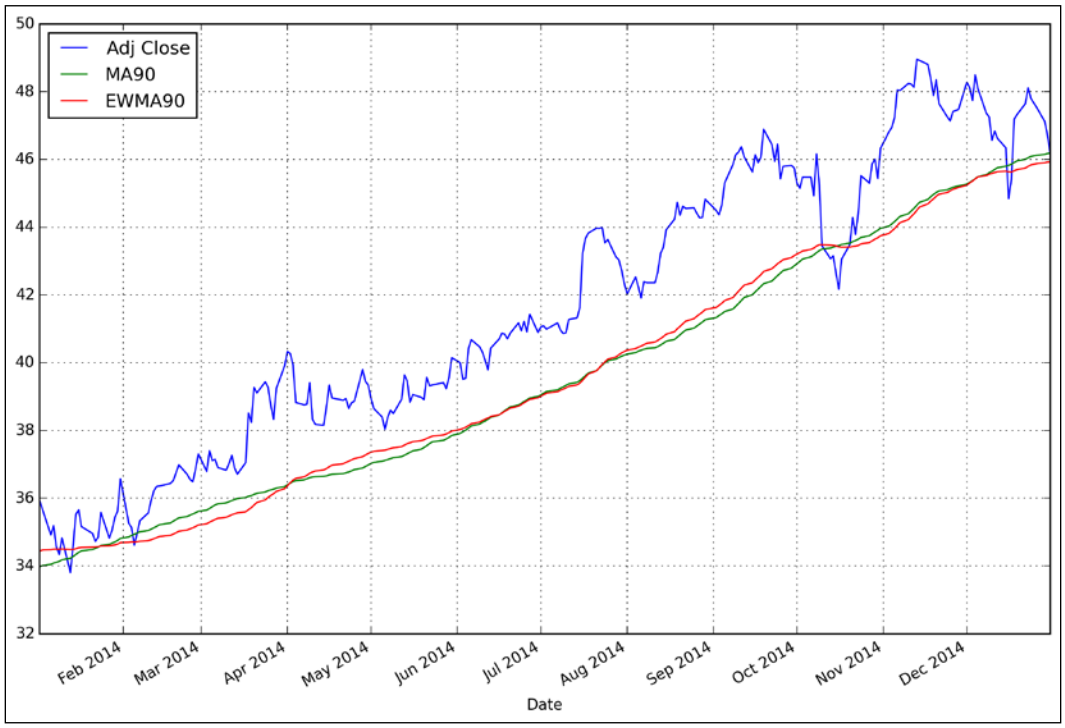
The most recent value is weighted at 21 percent of the result, and this decreases by a factor (1-alpha) across all the points, and the total of these weights is equal to 1.0.

The center of mass option specifies the point where half of the number of weights would be on each side of the center of mass. In the case of a 10-period span, the center of mass is `5.5`. Data points 1, 2, 3, 4, and 5 are on one side, and 6, 7, 8, 9, and 10 are on the other. The actual weight is not taken into account—just the number of items.

The half-life specification specifies the period of time for the percentage of the weighting factor to become half of its value. For the 10-period span, the half-life value is `3.454152`. The first weight is `0.21`, and we would expect that to reduce to `0.105` just under halfway between points 4 and 5 (`1+3.454152=4.454152`). These values are `0.115` and `0.094`, and `0.105` is indeed between the two.

The following example demonstrates how the exponential weighted moving average differs from a normal moving average. It calculates both kinds of averages for a 90-day window and plots the results:

```
In [7]:
   span = 90
   msft_ewma = msft[['Adj Close']].copy()
   msft_ewma['MA90'] = pd.rolling_mean(msft_ewma, span)
   msft_ewma['EWMA90'] = pd.ewma(msft_ewma['Adj Close'],
                                 span=span)
   msft_ewma['2014'].plot(figsize=(12, 8));
```



The exponential moving averages exhibit less lag, and, therefore, are more sensitive to recent prices and price changes. Since more recent values are favored, they will turn before simple moving averages, facilitating decision making on changes in momentum.

Comparatively, a simple moving average represents a truer average of prices for the entire time period. Therefore, a simple moving average may be better suited to identify the support or resistance level.

Technical analysis techniques

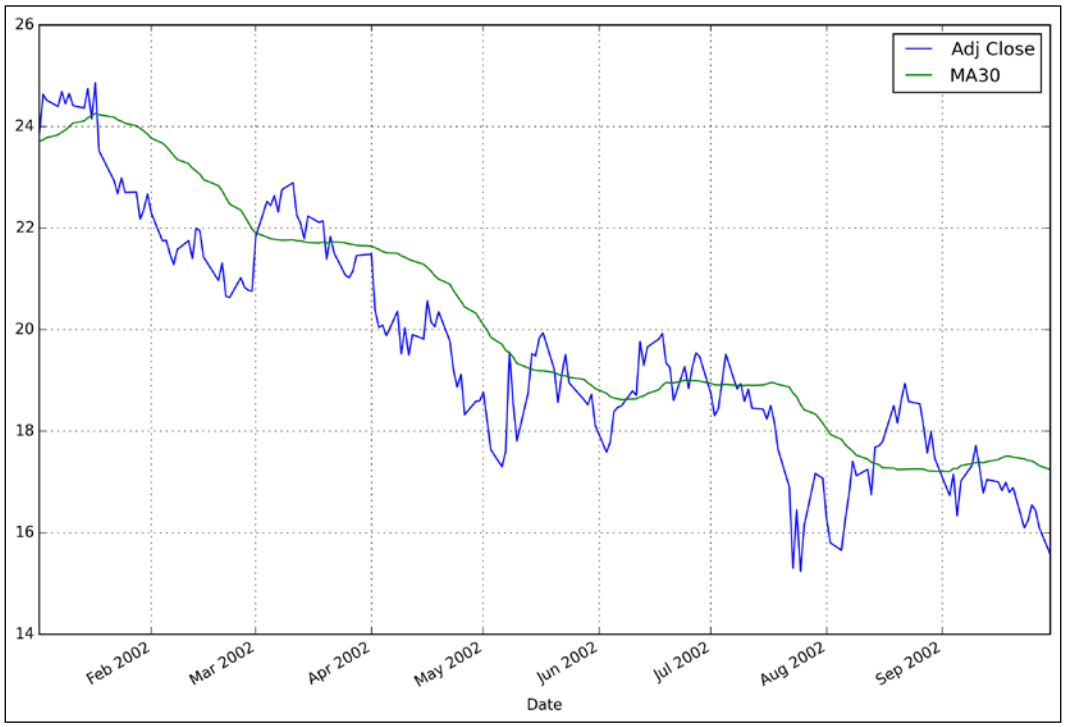
We will now cover two categories of technical analysis techniques, which utilize moving averages in different ways to be able to determine trends in market movements and hence give us the information needed to make potentially profitable transactions. We will examine how this works in this section, and in the upcoming section on Zipline, we will see how to implement these strategies in pandas and Zipline.

Crossovers

A crossover is the most basic type of signal for trading. The simplest form of a crossover is when the price of an asset moves from one side of a moving average to the other. This crossover represents a change in momentum and can be used as a point of making the decision to enter or exit the market.

The following command exemplifies several crossovers in the Microsoft data:

```
In [8]:
   msft['2002-1':'2002-9'][['Adj Close',
                            'MA30']].plot(figsize=(12,8));
```

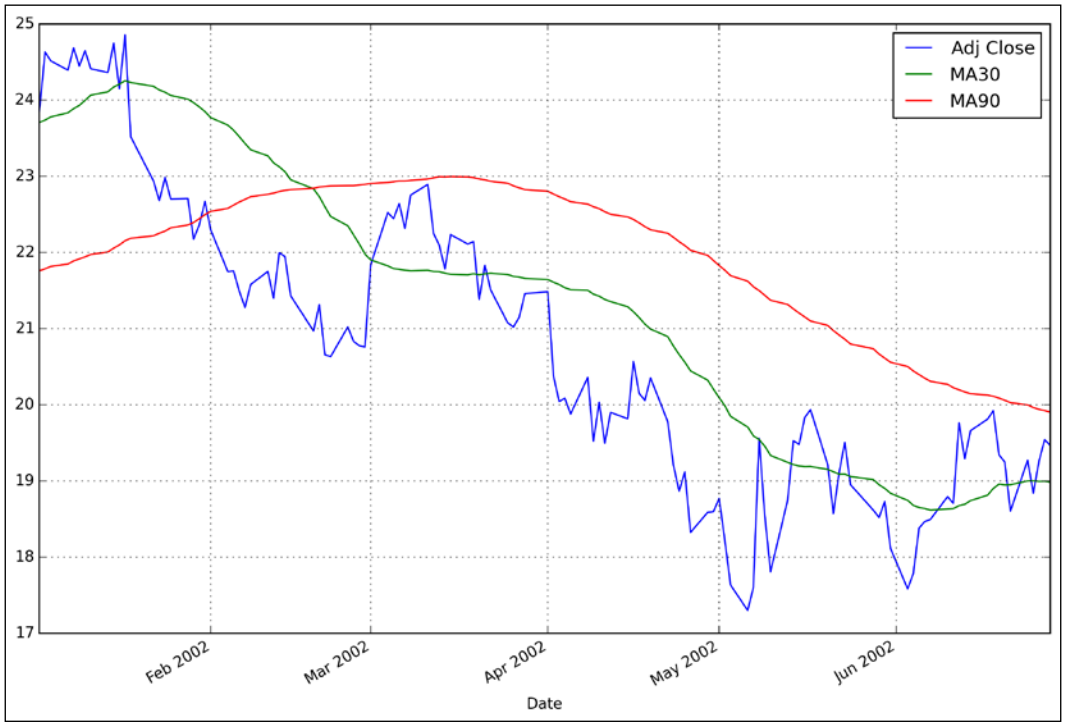


As an example, the cross occurring on July 09, 2002, is a signal of the beginning of a downtrend and would likely be used to close out any existing long positions. Conversely, a close above a moving average, as shown around August 13, may suggest the beginning of a new uptrend and a signal to go short on the stock.

A second type of crossover, referred to as a dual moving average crossover, occurs when a short-term average crosses a long-term average. This signal is used to identify that momentum is shifting in the direction of the short-term average. A buy signal is generated when the short-term average crosses the long-term average and rises above it, while a sell signal is triggered by a short-term average crossing long-term average and falling below it.

To demonstrate this, the following command shows MSFT for January 2002 through June 2002. There is one crossover of the 30- and 90-day moving averages with the 30-day crossing moving from above to below the 90-day average. This is a significant signal of the downswing of the stock during upcoming intervals:

```
In [9]:
   msft['2002-1':'2002-6'][['Adj Close', 'MA30', 'MA90']
                           ].plot(figsize=(12,8));
```



Pairs trading

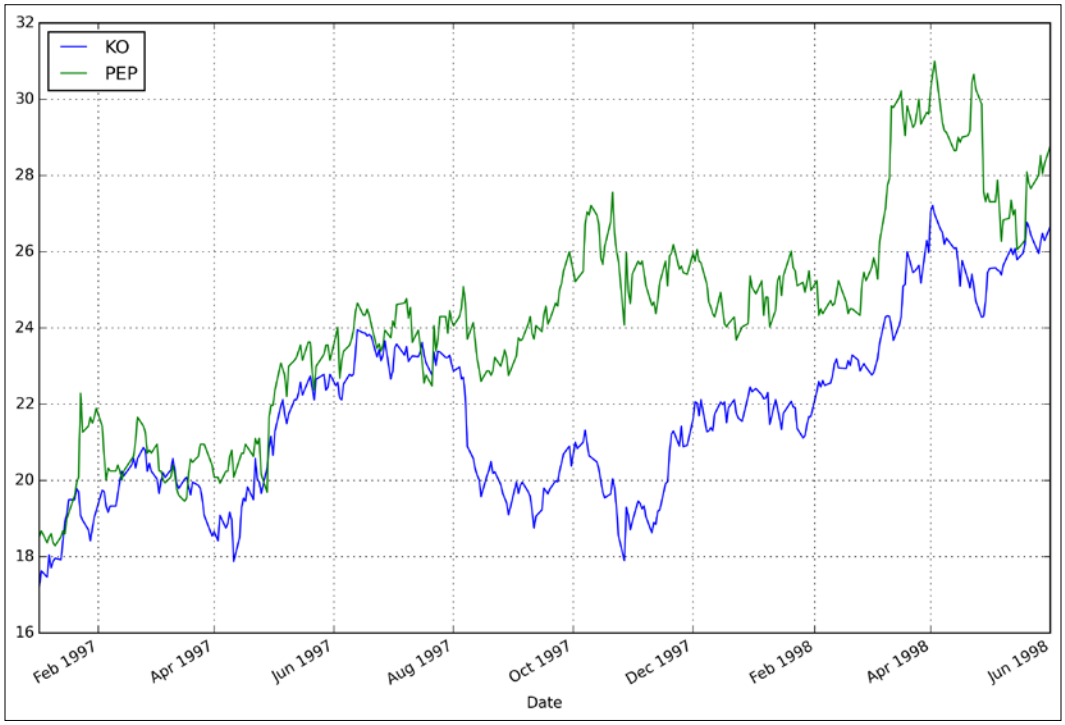
Pairs trading is a strategy that implements a statistical arbitrage and convergence. The basic idea is that, as we have seen, prices tend to move back to the mean. If two stocks can be identified that have a relatively high correlation, then the change in the difference in price between the two stocks can be used to signal trading events if one of the two moves out of correlation with the other.

If the change in the spread between the two stocks exceeds a certain level (their correlation has decreased), then the higher-priced stock can be considered to be in a short position and should be sold as it is assumed that the spread will decrease as the higher-priced stock returns to the mean (decreases in price as the correlation returns to a higher level). Likewise, the lower-priced stock is in a long position, and it is assumed that the price will rise as the correlation returns to normal levels.

This strategy relies on the two stocks being correlated as temporary reductions in correlation by one stock making either a positive or negative move. This is based upon the effects on one of the stocks that outside of shared market forces. This difference can be used to our advantage in an arbitrage by selling and buying equal amounts of each stock and profiting as the two prices move back into correlation. Of course, if the two stocks move into a truly different level of correlation, then this might be a losing situation.

Coca-Cola (**KO**) and **Pepsi** (**PEP**) are a canonical example of pairs-trading as they are both in the same market segment and are both likely to be affected by the same market events, such as the price of the common ingredients.

As an example, the following screenshot shows the price of Pepsi and Coca-Cola from January 1997 through June 1998 (we will revisit this series of data later when we implement pairs trading):



These prices are generally highly correlated during this period, but there is a marked change in correlation that starts in August 1997 and seems to take until the end of the year to move back into alignment. This is a situation where pairs trading can give profits if identified and executed properly.

Algo trading with Zipline

Zipline is a very powerful tool with many options, most of which we will not be able to investigate in this book. It makes creating trading algorithms and their simulation on historical data very easy (but there is still some creativity required).

Zipline provides several operational models. One allows the execution of Python script files via the command line. We will exclusively use a model where we include Zipline into our pandas application and request it to run our algorithms.

To do this, we will need to implement our algorithms and instruct Zipline on how to run them. This is actually a very simple process, and we will walk through implementing three algorithms of increasing complexity: buy apple, dual moving average crossover, and pairs trade.

The algorithms that we will implement have been discussed earlier: the dual moving average crossover and the pairs trading mean-reversion algorithm. We will, however, start with a very simple algorithm, buy apple, which will be used to demonstrate the overall process of how to create an algorithm as well as to show many of the things that Zipline handles automatically.

The three examples we will examine are available as part of this distribution, but we will examine them in detail. They have been modified to work exclusively within an IPython environment using pandas and to implement several of the constructs inherent in the examples in a manner that is better for understanding in the context of this book.

Algorithm – buy apple

Trading algorithms in Zipline are implemented in several manners. The technique we will use is creating a subclass of Zipline, that is, the `TradingAlgorithm` class and run the simulation within IPython with the Zipline engine.

The tracing is implemented as a static variable and the initialize method is called by Zipline as a static method to set up trading simulation. Also, initialize is called by Zipline prior to the completion of the call to `super()`, so to enable tracing, the member must be initialized before the call to `super()`.

The following is a simple algorithm for trading AAPL that is provided with the Zipline examples, albeit modified to be in a class, and run in IPython. Then, print some additional diagnostic code to trace how the process is executing in more detail:

```
In [11]:
   class BuyApple(zp.TradingAlgorithm):
       trace=False

       def __init__(self, trace=False):
           BuyApple.trace = trace
           super(BuyApple, self).__init__()

       def initialize(context):
           if BuyApple.trace: print("---> initialize")
           if BuyApple.trace: print(context)
           if BuyApple.trace: print("<--- initialize")

       def handle_data(self, context):
           if BuyApple.trace: print("---> handle_data")
           if BuyApple.trace: print(context)
           self.order("AAPL", 1)
           if BuyApple.trace: print("<-- handle_data")
```

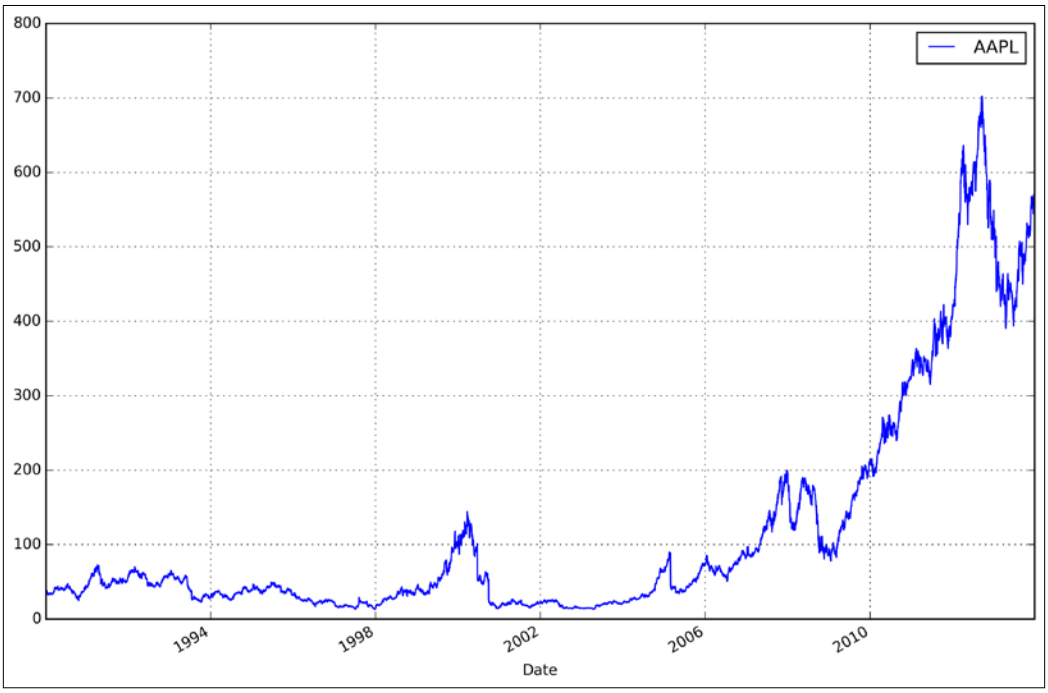
Trading simulation starts with the call to the static `.initialize()` method. This is your opportunity to initialize the trading simulation. In this sample, we do not perform any initialization other than printing the context for examination.

The implementation of the actual trading is handled in the override of the `handle_data` method. This method will be called for each day of the trading simulation. It is your opportunity to analyze the state of the simulation provided by the context and make any trading actions you desire. In this example, we will buy one share of AAPL regardless of how AAPL is performing.

The trading simulation can be started by instantiating an instance of `BuyApple()` and calling that object's `.run` method, thereby passing the base data for the simulation, which we will retrieve from Zipline's own method for accessing data from Yahoo! Finance:

```
In [12]:
   import zipline.utils.factory as zpf
   data = zpf.load_from_yahoo(stocks=['AAPL'],
```

```
                                indexes={},
                                start=datetime(1990, 1, 1),
                                end=datetime(2014, 1, 1),
                                adjusted=False)
    data.plot(figsize=(12,8));
```



Our first simulation will purposely use only one week of historical data so that we can easily keep the output to a nominal size that will help us to easily examine the results of the simulation:

```
In [13]:
    result = BuyApple().run(data['2000-01-03':'2000-01-07'])
    ---> initialize
    BuyApple(
        capital_base=100000.0
        sim_params=
    SimulationParameters(
        period_start=2006-01-01 00:00:00+00:00,
        period_end=2006-12-31 00:00:00+00:00,
```

```
    capital_base=100000.0,
    data_frequency=daily,
    emission_rate=daily,
    first_open=2006-01-03 14:31:00+00:00,
    last_close=2006-12-29 21:00:00+00:00),
    initialized=False,
    slippage=VolumeShareSlippage(
    volume_limit=0.25,
    price_impact=0.1),
    commission=PerShare(cost=0.03, min trade cost=None),
    blotter=Blotter(
    transact_partial=(VolumeShareSlippage(
    volume_limit=0.25,
    price_impact=0.1), PerShare(cost=0.03, min trade cost=None)),
    open_orders=defaultdict(<type 'list'>, {}),
    orders={},
    new_orders=[],
    current_dt=None),
    recorded_vars={})
<--- initialize
---> handle_data
BarData({'AAPL': SIDData({'volume': 1000, 'sid': 'AAPL',
'source_id': 'DataFrameSource-fc37c5097c557f0d46d6713256f4eaa3',
'dt': Timestamp('2000-01-03 00:00:00+0000', tz='UTC'), 'type': 4,
'price': 111.94})})
<-- handle_data
---> handle_data
 [2015-04-16 21:53] INFO: Performance: Simulated 5 trading days
out of 5.
 [2015-04-16 21:53] INFO: Performance: first open: 2000-01-03
14:31:00+00:00
 [2015-04-16 21:53] INFO: Performance: last close: 2000-01-07
21:00:00+00:00


BarData({'AAPL': SIDData({'price': 102.5, 'volume': 1000, 'sid':
'AAPL', 'source_id': 'DataFrameSource-
fc37c5097c557f0d46d6713256f4eaa3', 'dt': Timestamp('2000-01-04
00:00:00+0000', tz='UTC'), 'type': 4})})
```

```
   <-- handle_data
   ---> handle_data
   BarData({'AAPL': SIDData({'price': 104.0, 'volume': 1000, 'sid':
   'AAPL', 'source_id': 'DataFrameSource-
   fc37c5097c557f0d46d6713256f4eaa3', 'dt': Timestamp('2000-01-05
   00:00:00+0000', tz='UTC'), 'type': 4})})
   <-- handle_data
   ---> handle_data
   BarData({'AAPL': SIDData({'price': 95.0, 'volume': 1000, 'sid':
   'AAPL', 'source_id': 'DataFrameSource-
   fc37c5097c557f0d46d6713256f4eaa3', 'dt': Timestamp('2000-01-06
   00:00:00+0000', tz='UTC'), 'type': 4})})
   <-- handle_data
   ---> handle_data
   BarData({'AAPL': SIDData({'price': 99.5, 'volume': 1000, 'sid':
   'AAPL', 'source_id': 'DataFrameSource-
   fc37c5097c557f0d46d6713256f4eaa3', 'dt': Timestamp('2000-01-07
   00:00:00+0000', tz='UTC'), 'type': 4})})
   <-- handle_data
```

The context in the `initialize` method shows us some parameters that the simulation will use during its execution. The context also shows that we start with a base capitalization of `100000.0`. There will be a commission of $0.03 assessed for each share purchased.

The context is also printed for each day of trading. The output shows us that Zipline passes the price data for each day of AAPL. We do not utilize this information in this simulation and blindly purchase one share of AAPL.

The result of the simulation is assigned to the result variable, which we can analyze for detailed results of the simulation on each day of trading. This is a `DataFrame` where each column represents a particular measurement during the simulation, and each row represents the values of those variables on each day of trading during the simulation.

We can examine a number of the variables to demonstrate what Zipline was doing during the processing. The orders variable contains a list of all orders made during the day. The following command gets the orders for the first day of the simulation:

```
In [14]:
   result.iloc[0].orders


Out[14]:
   [{'amount': 1,
```

```
     'commission': None,
     'created': Timestamp('2000-01-03 00:00:00+0000', tz='UTC'),
     'dt': Timestamp('2000-01-03 00:00:00+0000', tz='UTC'),
     'filled': 0,
     'id': 'dccb19f416104f259a7f0bff726136a2',
     'limit': None,
     'limit_reached': False,
     'sid': 'AAPL',
     'status': 0,
     'stop': None,
     'stop_reached': False}]
```

This tells us that Zipline placed an order in the market for one share of AAPL on `2000-01-03`. The order filled the value `0`, which means that this trade has not yet been executed in the market.

On the second day of trading, Zipline reports that two orders were made:

```
In [15]:
   result.iloc[1].orders

Out[15]:
    [{'amount': 1,
     'commission': 0.03,
     'created': Timestamp('2000-01-03 00:00:00+0000', tz='UTC'),
     'dt': Timestamp('2000-01-04 00:00:00+0000', tz='UTC'),
     'filled': 1,
     'id': 'dccb19f416104f259a7f0bff726136a2',
     'limit': None,
     'limit_reached': False,
     'sid': 'AAPL',
     'status': 1,
     'stop': None,
     'stop_reached': False},
    {'amount': 1,
     'commission': None,
     'created': Timestamp('2000-01-04 00:00:00+0000', tz='UTC'),
     'dt': Timestamp('2000-01-04 00:00:00+0000', tz='UTC'),
     'filled': 0,
     'id': '1ec23ea51fd7429fa97b9f29a66bf66a',
```

```
    'limit': None,
    'limit_reached': False,
    'sid': 'AAPL',
    'status': 0,
    'stop': None,
    'stop_reached': False}]
```

The first order listed has the same ID as the order from day one. This tells us that this represents that same order, and we can see this from the filled key, which is now `1` and from the fact that this order has been filled in the market.

The second order is a new order, which represents our request on the second day of trading, which will be reported as filled at the start of day two.

During the simulation, Zipline keeps track of the amount of cash we have (capital) at the start and end of the day. As we purchase stocks, our cash is reduced. Starting and ending cash is represented by the `starting_cash` and `ending_case` variables of the result.

Zipline also accumulates the total value of the purchases of stock during the simulation. This value is represented in each trading period using the `ending_value` variable of the result.

The following command shows us the running values for `ending_cash` and `ending_value`, along with `ending_value`:

```
In [16]:
   result[['starting_cash', 'ending_cash', 'ending_value']]


Out[16]:
                          starting_cash   ending_cash  ending_value
   2000-01-03 21:00:00   100000.00000  100000.00000           0.0
   2000-01-04 21:00:00   100000.00000   99897.46999         102.5
   2000-01-05 21:00:00    99897.46999   99793.43998         208.0
   2000-01-06 21:00:00    99793.43998   99698.40997         285.0
   2000-01-07 21:00:00    99698.40997   99598.87996         398.0
```

Ending cash represents the amount of cash (capital) that we have to invest at the end of the given day. We made an order on day one for one share of the apple, but since the transaction did not execute until the next day, we still have our starting seed at the end of the day. But on day two, this will execute at the value reported at the close of day one, which is `111.94`. Hence, our `ending_cash` is reduced by `111.94` for one share and also deducted is the $0.03 for the commission resulting in `9987.47`.

At the end of day two, our `ending_value`, that is, our position in the market, is `102.5` as we have accumulated one share of AAPL, and it closed at `102.5` on day two.

We did not print `starting_cash` and `starting_value` as this will always be equal to our initial capitalization of `100000.0` and a portfolio value of `0.0` as we have not yet bought any securities.

While investing, we would be interested in the overall value of our portfolio, which, in this case, would be the value of our on-hand cash + our position in the market. This can be easily calculated:

```
In [17]:
   pvalue = result.ending_cash + result.ending_value
   pvalue

Out[17]:
   2000-01-03 21:00:00    100000.00000
   2000-01-04 21:00:00     99999.96999
   2000-01-05 21:00:00    100001.43998
   2000-01-06 21:00:00     99983.40997
   2000-01-07 21:00:00     99996.87996
   dtype: float64
```

There is also a convenient shorthand to retrieve this result:

```
In [18]:
   result.portfolio_value

Out[18]:
   2000-01-03 21:00:00    100000.00000
   2000-01-04 21:00:00     99999.96999
   2000-01-05 21:00:00    100001.43998
   2000-01-06 21:00:00     99983.40997
   2000-01-07 21:00:00     99996.87996
   Name: portfolio_value, dtype: float64
```

In a similar vein, we can also calculate the daily returns on our investment using `.pct_change()`:

```
In [19]:
   result.portfolio_value.pct_change()
```

```
Out[19]:
   2000-01-03 21:00:00            NaN
   2000-01-04 21:00:00   -3.00103e-07
   2000-01-05 21:00:00    1.46999e-05
   2000-01-06 21:00:00   -1.80297e-04
   2000-01-07 21:00:00    1.34722e-04
   Name: portfolio_value, dtype: float64
```

This is actually a column of the results from the simulation, so we do not need to actually calculate it:

```
In [20]:
   result['returns']


Out[20]:
   2000-01-03 21:00:00            NaN
   2000-01-04 21:00:00   -3.00103e-07
   2000-01-05 21:00:00    1.46999e-05
   2000-01-06 21:00:00   -1.80297e-04
   2000-01-07 21:00:00    1.34722e-04
   Name: portfolio_value, dtype: float64
```

Using this small trading interval, we have seen what type of calculations Zipline performs during each period. Now, let's run this simulation over a longer period of time to see how it performs. The following command runs the simulation across the entire year 2000:

```
In [21]:
   result_for_2000 = BuyApple().run(data['2000'])


Out[21]:
    [2015-02-15 05:05] INFO: Performance: Simulated 252 trading days
     out of 252.
    [2015-02-15 05:05] INFO: Performance: first open: 2000-01-03
     14:31:00+00:00
    [2015-02-15 05:05] INFO: Performance: last close: 2000-12-29
     21:00:00+00:00
```

The following command shows us our cash on hand and the value of our investments throughout the simulation:

```
In [22]:
   result_for_2000[['ending_cash', 'ending_value']]
```

```
Out[22]:
                          ending_cash  ending_value
   2000-01-03 21:00:00  100000.00000          0.00
   2000-01-04 21:00:00   99897.46999        102.50
   2000-01-05 21:00:00   99793.43998        208.00
   2000-01-06 21:00:00   99698.40997        285.00
   2000-01-07 21:00:00   99598.87996        398.00
   ...                           ...           ...
   2000-12-22 21:00:00   82082.91821       3705.00
   2000-12-26 21:00:00   82068.19821       3643.12
   2000-12-27 21:00:00   82053.35821       3687.69
   2000-12-28 21:00:00   82038.51820       3702.50
   2000-12-29 21:00:00   82023.60820       3734.88

   [252 rows x 2 columns]
```

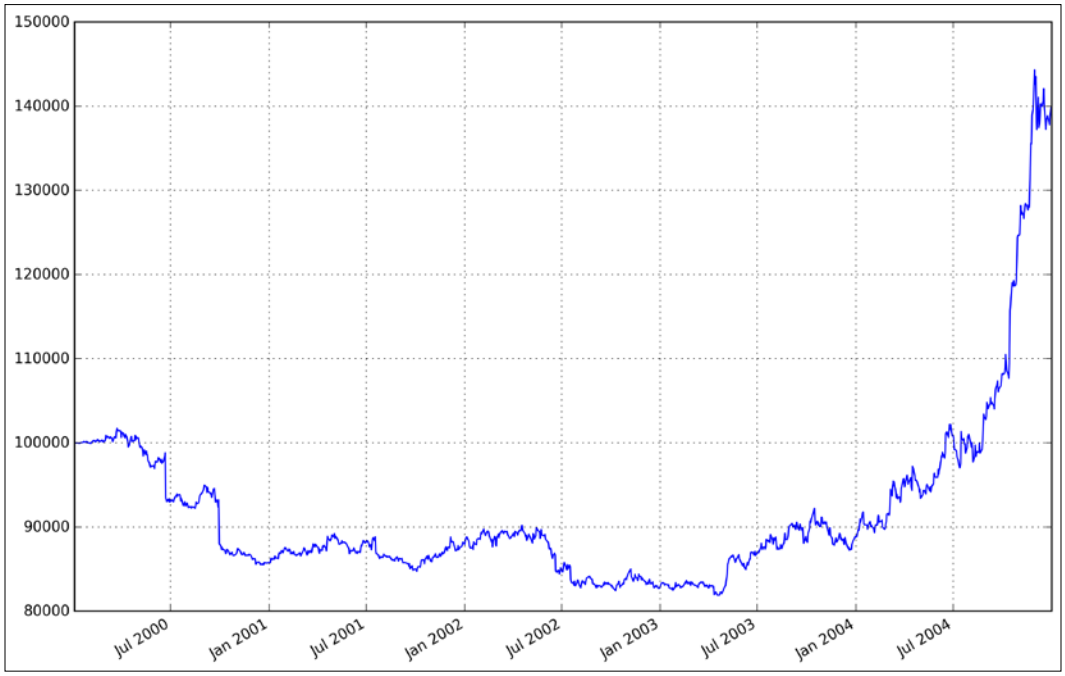
The following command visualizes our overall portfolio value during the year 2000:

```
In [23]:
   result_for_2000.portfolio_value.plot(figsize=(12,8));
```

Our strategy has lost us money over the year 2000. AAPL generally trended downward during the year, and simply buying every day is a losing strategy.

The following command runs the simulation over 5 years:

```
In [24]:
   result = BuyApple().run(data['2000':'2004']).portfolio_value
   result.plot(figsize=(12,8));
    [2015-04-16 22:52] INFO: Performance: Simulated 1256 trading days
     out of 1256.
    [2015-04-16 22:52] INFO: Performance: first open: 2000-01-03
     14:31:00+00:00
    [2015-04-16 22:52] INFO: Performance: last close: 2004-12-31
     21:00:00+00:00
```



Hanging in with this strategy over several more years has paid off as AAPL had a marked upswing in value starting in mid-2013.

Algorithm – dual moving average crossover

We now analyze a dual moving average crossover strategy. This algorithm will buy apple once its short moving average crosses its long moving average. This will indicate upward momentum and a buy situation. It will then begin selling shares once the averages cross again, which will represent downward momentum.

We will load data for AAPL for 1990 through 2014, but we will only use the data from 1990 through 2001 in the simulation:

```
In [25]:
   sub_data = data['1990':'2002-01-01']
   sub_data.plot();
```

The following class implements a double moving average crossover where investments will be made whenever the short moving average moves across the long moving average. We will trade only at the cross, not continuously buying or selling until the next cross. If trending down, we will sell all of our stock. If trending up, we buy as many shares as possible up to 100. The strategy will record our buys and sells in extra data returned from the simulation:

```
In [26]:
```

```
class DualMovingAverage(zp.TradingAlgorithm):
    def initialize(context):
        # we need to track two moving averages, so we will set
        # these up in the context the .add_transform method
        # informs Zipline to execute a transform on every day
        # of trading

        # the following will set up a MovingAverge transform,
        # named short_mavg, accessing the .price field of the
        # data, and a length of 100 days
        context.add_transform(zp.transforms.MovingAverage,
                              'short_mavg', ['price'],
                              window_length=100)

        # and the following is a 400 day MovingAverage
        context.add_transform(zp.transforms.MovingAverage,
                              'long_mavg', ['price'],
                              window_length=400)

        # this is a flag we will use to track the state of
        # whether or not we have made our first trade when the
        # means cross.  We use it to identify the single event
        # and to prevent further action until the next cross
        context.invested = False

    def handle_data(self, data):
        # access the results of the transforms
        short_mavg = data['AAPL'].short_mavg['price']
        long_mavg = data['AAPL'].long_mavg['price']

        # these flags will record if we decided to buy or sell
        buy = False
        sell = False

        # check if we have crossed
        if short_mavg > long_mavg and not self.invested:
            # short moved across the long, trending up
            # buy up to 100 shares
            self.order_target('AAPL', 100)
            # this will prevent further investment until
```

```
            # the next cross
            self.invested = True
            buy = True # records that we did a buy
        elif short_mavg < long_mavg and self.invested:
            # short move across the long, trending down
            # sell it all!
            self.order_target('AAPL', -100)
            # prevents further sales until the next cross
            self.invested = False
            sell = True # and note that we did sell

        # add extra data to the results of the simulation to
        # give the short and long ma on the interval, and if
        # we decided to buy or sell
        self.record(short_mavg=short_mavg,
                    long_mavg=long_mavg,
                    buy=buy,
                    sell=sell)
```

We can now execute this algorithm by passing it data from 1990 through 2001, as shown here:

```
In [27]:
   results = DualMovingAverage().run(sub_data)
    [2015-02-15 22:18] INFO: Performance: Simulated 3028 trading days
     out of 3028.
    [2015-02-15 22:18] INFO: Performance: first open: 1990-01-02
     14:31:00+00:00
    [2015-02-15 22:18] INFO: Performance: last close: 2001-12-31
     21:00:00+00:00
```

To analyze the results of the simulation, we can use the following function that creates several charts that show the short/long means relative to price, the value of the portfolio, and the points at which we made buys and sells:

```
In [28]:
   def analyze(data, perf):
       fig = plt.figure()
       ax1 = fig.add_subplot(211,  ylabel='Price in $')
       data['AAPL'].plot(ax=ax1, color='r', lw=2.)
       perf[['short_mavg', 'long_mavg']].plot(ax=ax1, lw=2.)

       ax1.plot(perf.ix[perf.buy].index, perf.short_mavg[perf.buy],
```

```
            '^', markersize=10, color='m')
    ax1.plot(perf.ix[perf.sell].index, perf.short_mavg[perf.sell],
            'v', markersize=10, color='k')

    ax2 = fig.add_subplot(212, ylabel='Portfolio value in $')
    perf.portfolio_value.plot(ax=ax2, lw=2.)

    ax2.plot(perf.ix[perf.buy].index,
            perf.portfolio_value[perf.buy],
            '^', markersize=10, color='m')
    ax2.plot(perf.ix[perf.sell].index,
            perf.portfolio_value[perf.sell],
            'v', markersize=10, color='k')

    plt.legend(loc=0)
    plt.gcf().set_size_inches(14, 10)
```

Using this function, we can plot the decisions made and the resulting portfolio value as trades are executed:

```
In [29]:
   analyze(sub_data, results)
```

The crossover points are noted on the graphs using triangles. Upward-pointing red triangles identify buys and downward-pointing black triangles identify sells. Portfolio value stays level after a sell as we are completely divested from the market until we make another purchase.

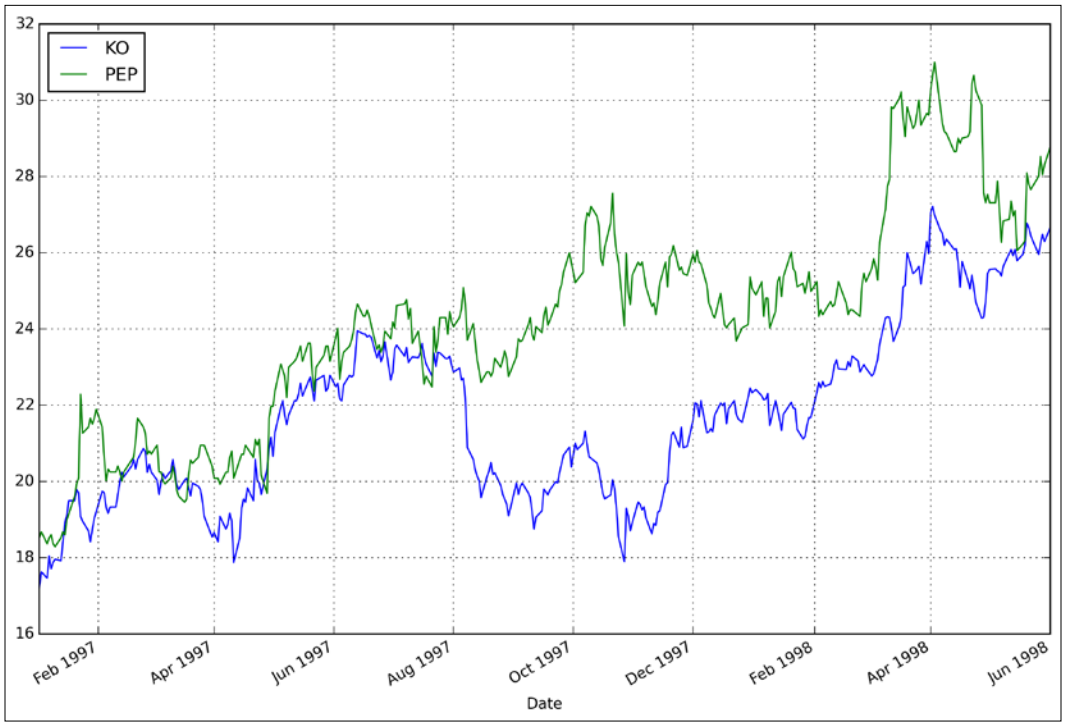
Algorithm – pairs trade

To demonstrate a pairs trade algorithm, we will create one such algorithm and run data for Pepsi and Coca-Cola through the simulation. Since these two stocks are in the same market segment, their prices tend to follow each other based on common influences in the market.

If there is an increase in the delta between the two stocks, a trader can potentially make money by buying the stock that stayed the same and selling the increasing stock. The assumption is that the two stocks will revert to a common spread on the mean. Hence, if the stock that stayed normal increases to close the gap, then the buy will result in increased value. If the rising stock reverts, then the sell will create profit. If both happen, then even better.

To start with, we will need to gather data for Coke and Pepsi:

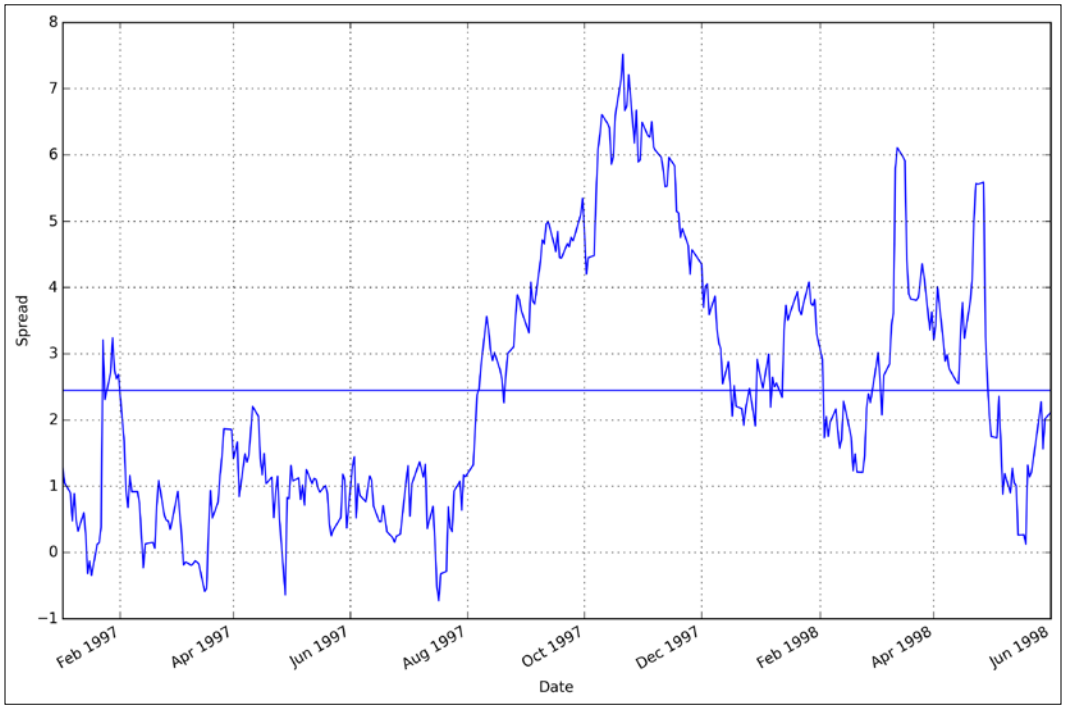
```
In [30]:
   data = zpf.load_from_yahoo(stocks=['PEP', 'KO'],
                              indexes={},
                              start=datetime(1997, 1, 1),
                              end=datetime(1998, 6, 1),
                              adjusted=True)
   data.plot(figsize=(12,8));
   PEP
   KO
```



Analyzing the chart, we can see that the two stocks tend to follow along the same trend line, but that there is a point where Coke takes a drop relative to Pepsi (August 1997 through December 1997). It then tends to follow the same path although with a wider spread during 1998 than in early 1997.

We can dive deeper into this information to see what we can do with pairs trading. In this algorithm, we will examine how the spread between the two stocks change. Therefore, we need to calculate the spread:

```
In [31]:
   data['PriceDelta'] = data.PEP - data.KO
   data['1997':].PriceDelta.plot(figsize=(12,8))
   plt.ylabel('Spread')
   plt.axhline(data.Spread.mean());
```



Using this information, we can make a decision to buy one stock and sell the other if the spread exceeds a particular size. In the algorithm we implement, we will normalize the spread data on a 100-day window and use that to calculate the z-score on each particular day.

If the z-score is > 2, then we will want to buy PEP and sell KO as the spread increases over our threshold with PEP taking the higher price. If the z-score is < -2, then we want to buy KO and sell PEP, as PEP takes the lower price as the spread increases. Additionally, if the absolute value of the z-score < 0.5, then we will sell off any holdings we have in either stock to limit our exposure as we consider the spread to be fairly stable and we can divest.

One calculation that we will need to perform during the simulation is calculating the regression of the two series prices. This will then be used to calculate the z-score of the spread at each interval. To do this, the following function is created:

```
In [32]:
   @zp.transforms.batch_transform
   def ols_transform(data, ticker1, ticker2):
       p0 = data.price[ticker1]
       p1 = sm.add_constant(data.price[ticker2], prepend=True)
       slope, intercept = sm.OLS(p0, p1).fit().params
       return slope, intercept
```

You may wonder what the `@zp.transforms.batch_transform` code does. At each iteration of the simulation, Zipline will only give us the data representing the current price. Passing the data from `handle_data` to this function would only pass the current day's data. This decorator will tell Zipline to pass all of the historical data instead of the current day's data. This makes this very simple as, otherwise, we would need to manage multiple windows of data manually in our code.

The actual algorithm is then implemented using a 100-day window where we will execute on the spread when the z-score is > 2.0 or < -2.0. If the absolute value of the z-score is < 0.5, then we will empty our position in the market to limit exposure:

```
In [33]:
   class Pairtrade(zp.TradingAlgorithm):
       def initialize(self, window_length=100):
           self.spreads=[]
           self.invested=False
           self.window_length=window_length
           self.ols_transform= \
               ols_transform(refresh_period=self.window_length,
                             window_length=self.window_length)

       def handle_data(self, data):
           # calculate the regression, will be None until 100 samples
           params=self.ols_transform.handle_data(data, 'PEP', 'KO')
           if params:
               intercept, slope=params
               zscore=self.compute_zscore(data, slope, intercept)
               self.record(zscore=zscore)
               self.place_orders(data, zscore)
```

```
    def compute_zscore(self, data, slope, intercept):
        # calculate the spread
        spread=(data['PEP'].price-(slope*data['KO'].price+
                                   intercept))
        self.spreads.append(spread) # record for z-score calc
        self.record(spread = spread)

        spread_wind=self.spreads[-self.window_length:]
        zscore=(spread - np.mean(spread_wind))/np.std(spread_wind)
        return zscore

    def place_orders(self, data, zscore):
        if zscore>=2.0 and not self.invested:
            # buy the spread, buying PEP and selling KO
            self.order('PEP', int(100/data['PEP'].price))
            self.order('KO', -int(100/data['KO'].price))
            self.invested=True
            self.record(action="PK")
        elif zscore<=-2.0 and not self.invested:
            # buy the spread, buying KO and selling PEP
            self.order('PEP', -int(100 / data['PEP'].price))
            self.order('KO', int(100 / data['KO'].price))
            self.invested = True
            self.record(action='KP')
        elif abs(zscore)<.5 and self.invested:
            # minimize exposure
            ko_amount=self.portfolio.positions['KO'].amount
            self.order('KO', -1*ko_amount)
            pep_amount=self.portfolio.positions['PEP'].amount
            self.order('PEP', -1*pep_amount)
            self.invested=False
            self.record(action='DE')
        else:
            # take no action
            self.record(action='noop')
```

Then, we can run the algorithm with the following command:

```
In [34]:
   perf = Pairtrade().run(data['1997':])
    [2015-02-16 01:54] INFO: Performance: Simulated 356 trading days
     out of 356.
    [2015-02-16 01:54] INFO: Performance: first open: 1997-01-02
     14:31:00+00:00
    [2015-02-16 01:54] INFO: Performance: last close: 1998-06-01
     20:00:00+00:00
```

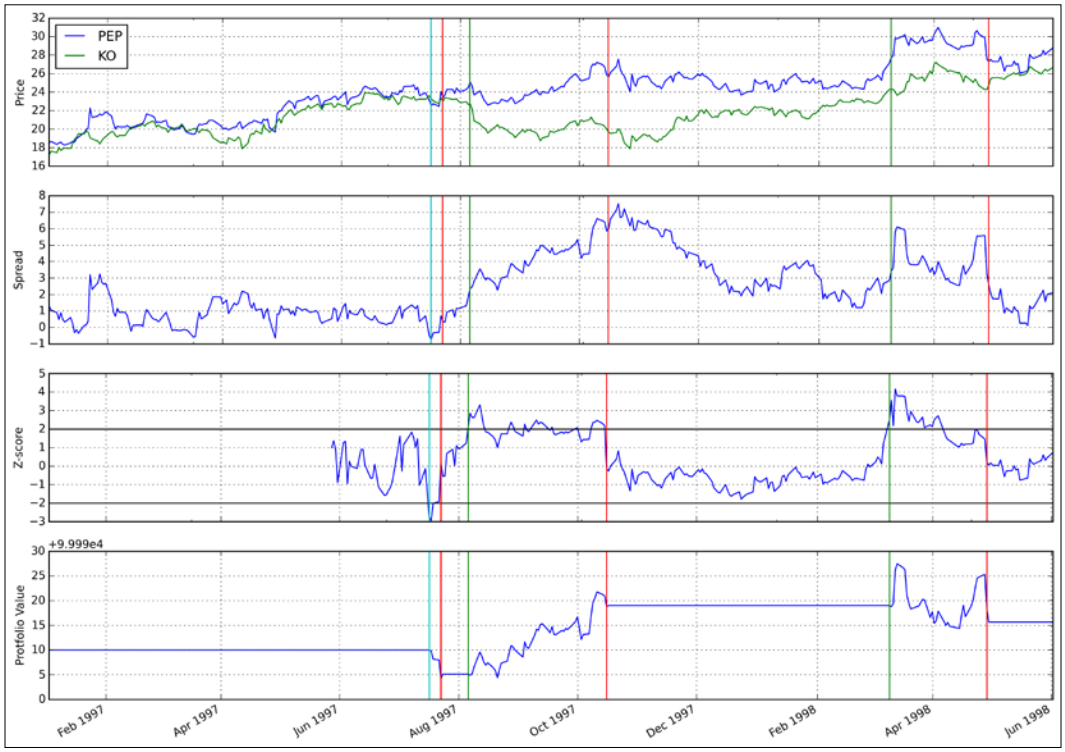
During the simulation of the algorithm, we recorded any transactions made, which can be accessed using the action column of the result `DataFrame`:

```
In [35]:
   selection = ((perf.action=='PK') | (perf.action=='KP') |
                (perf.action=='DE'))
   actions = perf[selection][['action']]
   actions

Out[35]:
   1997-07-16 20:00:00     KP
   1997-07-22 20:00:00     DE
   1997-08-05 20:00:00     PK
   1997-10-15 20:00:00     DE
   1998-03-09 21:00:00     PK
   1998-04-28 20:00:00     DE
```

Our algorithm made six transactions. We can examine these transactions by visualizing the prices, spreads, z-scores, and portfolio values relative to when we made transactions (represented by vertical lines):

```
In [36]:
   ax1 = plt.subplot(411)
   data[['PEP', 'KO']].plot(ax=ax1)
   plt.ylabel('Price')

   data.Spread.plot(ax=ax2)
   plt.ylabel('Spread')

   ax3 = plt.subplot(413)
   perf['1997':].zscore.plot()
```

```
ax3.axhline(2, color='k')
ax3.axhline(-2, color='k')
plt.ylabel('Z-score')

ax4 = plt.subplot(414)
perf['1997':].portfolio_value.plot()
plt.ylabel('Protfolio Value')

for ax in [ax1, ax2, ax3, ax4]:
    for d in actions.index[actions.action=='PK']:
        ax.axvline(d, color='g')
    for d in actions.index[actions.action=='KP']:
        ax.axvline(d, color='c')
    for d in actions.index[actions.action=='DE']:
        ax.axvline(d, color='r')

plt.gcf().set_size_inches(16, 12)
```



The first event is on 1997-7-16 when the algorithm saw the spread become less than -2, and, therefore, triggered a sale of KO and a buy of PEP. This quickly turned around and moved to a z-score of 0.19 on 1997-7-22, triggering a divesting of our position. During this time, even though we played the spread, we still lost because a reversion happened very quickly.

On 1997-08-05, the z-score moved above 2.0 to 2.12985 and triggered a purchase of KO and a sale of PEP. The z-score stayed around 2.0 until 1997-10-15 when it dropped to -0.1482 and, therefore, we divested. Between those two dates, since the spread stayed fairly consistent around 2.0, our playing of the spread made us consistent returns as we can see with the portfolio value increasing steadily over that period.

On 1998-03-09, a similar trend was identified, and again, we bought KO and sold PEP. Unfortunately the spread started to minimize and we lost a little during this period.

Summary

In this chapter, we took an adventure into learning the fundamentals of algorithmic trading using pandas and Zipline. We started with a little theory to set a framework for understanding how the algorithms would be implemented. From there, we implemented three different trading algorithms using Zipline and dived into the decisions made and their impact on the portfolios as the transactions were executed. Finally, we established a fundamental knowledge of how to simulate markets and make automated trading decisions.

8
Working with Options

In this chapter, we will examine working with options data provided by Yahoo! Finance using pandas. Options are a type of financial derivative and can be very complicated to price and use in investment portfolios. Because of their level of complexity, there have been many books written that are focus heavily on the mathematics of options. Our goal will not be to cover the mathematics in detail but to focus on understanding several core concepts in options, retrieving options data from the Internet, manipulating it using pandas, including determining their value, and being able to check the validity of the prices offered in the market.

In this chapter, we will specifically cover:

- A brief introduction to options
- Retrieving options data from Yahoo! Finance
- Examining the attributes of an option
- Implied volatility, including smiles and smirks
- Calculating the payoff of options
- Determining the profit and loss of options
- The pricing of options using Black-Scholes
- Using Mibian to price and determine the implied volatility of options with Black-Scholes
- An introduction to the Greeks
- Examining the behavior of the Greeks

Introducing options

An option is a contract that gives the buyer the right, but not the obligation, to buy or sell an underlying security at a specific price on or before a certain date. Options are considered derivatives as their price is derived from one or more underlying securities. Options involve two parties: the buyer and the seller. The parties buy and sell the option, not the underlying security.

There are two general types of options: the call and the put. Let's look at them in detail:

- **Call**: This gives the holder of the option the right to buy an underlying security at a certain price within a specific period of time. They are similar to having a long position on a stock. The buyer of a call is hoping that the value of the underlying security will increase substantially before the expiration of the option and, therefore, they can buy the security at a discount from the future value.
- **Put**: This gives the option holder the right to sell an underlying security at a certain price within a specific period of time. A put is similar to having a short position on a stock. The buyer of a put is betting that the price of the underlying security will fall before the expiration of the option and they will, thereby, be able to gain a profit by benefitting from receiving the payment in excess of the future market value.

The basic idea is that one side of the party believes that the underlying security will increase in value and the other believes it will decrease. They will agree upon a price known as the strike price, where they place their bet on whether the price of the underlying security finishes above or below this strike price on the expiration date of the option.

Through the contract of the option, the option seller agrees to give the buyer the underlying security on the expiry of the option if the price is above the strike price (for a call).

The price of the option is referred to as the premium. This is the amount the buyer will pay the seller to receive the option. The price of an option depends upon many factors, of which the following are the primary factors:

- The current price of the underlying security
- How long the option needs to be held before it expires (the expiry date)
- The strike price on the expiry date of the option
- The interest rate of capital in the market
- The volatility of the underlying security
- There being an adequate interest between buyer and seller around the given option

The premium is often established so that the buyer can speculate on the future value of the underlying security and be able to gain rights to the underlying security in the future at a discount in the present.

The holder of the option, known as the buyer, is not obliged to exercise the option on its expiration date, but the writer, also referred to as the seller, however, is obliged to buy or sell the instrument if the option is exercised.

Options can provide a variety of benefits such as the ability to limit risk and the advantage of providing leverage. They are often used to diversify an investment portfolio to lower risk during times of rising or falling markets.

There are four types of participants in an options market:

- Buyers of calls
- Sellers of calls
- Buyers of puts
- Sellers of puts

Buyers of calls believe that the underlying security will exceed a certain level and are not only willing to pay a certain amount to see whether that happens, but also lose their entire premium if it does not. Their goal is that the resulting payout of the option exceeds their initial premium and they, therefore, make a profit. However, they are willing to forgo their premium in its entirety if it does not clear the strike price. This then becomes a game of managing the risk of the profit versus the fixed potential loss.

Sellers of calls are on the other side of buyers. They believe the price will drop and that the amount they receive in payment for the premium will exceed any loss in the price. Normally, the seller of a call would already own the stock. They do not believe the price will exceed the strike price and that they will be able to keep the underlying security and profit if the underlying security stays below the strike price by an amount that does not exceed the received premium. Loss is potentially unbounded as the stock increases in price above the strike price, but that is the risk for an upfront receipt of cash and potential gains on the loss of price in the underlying instrument.

A buyer of a put is betting that the price of the stock will drop beyond a certain level. By buying a put they gain the option to force someone to buy the underlying instrument at a fixed price. By doing this, they are betting that they can force the sale of the underlying instrument at a strike price that is higher than the market price and in excess of the premium that they pay to the seller of the put option.

On the other hand, the seller of the put is betting that they can make an offer on an instrument that is perceived to lose value in the future. They will offer the option for a price that gives them cash upfront, and they plan that at maturity of the option, they will not be forced to purchase the underlying instrument. Therefore, it keeps the premium as pure profit. Or, the price of the underlying instruments drops only a small amount so that the price of buying the underlying instrument relative to its market price does not exceed the premium that they received.

Notebook setup

The examples in this chapter will be based on the following configuration in IPython:

```
In [1]:
   import pandas as pd
   import numpy as np
   import pandas.io.data as web
   from datetime import datetime

   import matplotlib.pyplot as plt
   %matplotlib inline

   pd.set_option('display.notebook_repr_html', False)
   pd.set_option('display.max_columns', 7)
   pd.set_option('display.max_rows', 15)
   pd.set_option('display.width', 82)
   pd.set_option('precision', 3)
```

Options data from Yahoo! Finance

Options data can be obtained from several sources. Publicly listed options are exchanged on the **Chicago Board Options Exchange** (**CBOE**) and can be obtained from their website. Through the `DataReader` class, pandas also provides built-in (although in the documentation, this is referred to as experimental) access to options data.

The following command reads all currently available options data for AAPL:

```
In [2]:
   aapl_options = web.Options('AAPL', 'yahoo')
   aapl_options = aapl_options.get_all_data().reset_index()
```

This operation can take a while as it downloads quite a bit of data. Fortunately, it is cached so that subsequent calls will be quicker, and there are other calls to limit the types of data downloaded (such as just getting puts).

For convenience, the following command will save this data to a file for quick reload at a later time. Also, it helps with the repeatability of the examples. The data retrieved changes very frequently, so the actual examples in the book will use the data in the file provided with the book. It saves the data for later use (it's commented out for now so it does not overwrite the existing file). Here's the command we are talking about:

```
In [3]:
   #aapl_options.to_csv('aapl_options.csv')
```

This data file can be reloaded with the following command:

```
In [4]:
   aapl_options = pd.read_csv('aapl_options.csv',
                              parse_dates=['Expiry'])
```

I highly recommend that you use the data file for the purposes of going along with the chapter as options data changes very frequently and loading directly from the Web will make the results you get completely different from those in the chapter.

Whether from the Web or the file, the following command restructures and tidies the data into a format best used in the examples that follow:

```
In [5]:
   aos = aapl_options.sort(['Expiry', 'Strike'])[
      ['Expiry', 'Strike', 'Type', 'IV', 'Bid',
          'Ask', 'Underlying_Price']]
   aos['IV'] = aos['IV'].apply(lambda x: float(x.strip('%')))
```

Now, we can take a look at the data retrieved:

```
In [6]:
   aos


Out[6]:
            Expiry  Strike Type      IV   Bid   Ask  Underlying_Price
   158  2015-02-27      75 call  271.88 53.60 53.85            128.79
   159  2015-02-27      75  put  193.75  0.00  0.01            128.79
```

```
   190  2015-02-27      80 call  225.78 48.65 48.80            128.79
   191  2015-02-27      80  put  171.88  0.00  0.01            128.79
   226  2015-02-27      85 call  199.22 43.65 43.80            128.79
```

There are 1,103 rows of options data available. The data is sorted by `Expiry` and then the `Strike` price to help demonstrate examples.

`Expiry` is the data at which the particular option will expire and potentially be exercised. We have the following expiry dates that were retrieved. Options typically are offered by an exchange on a monthly basis and within a short overall duration from several days to perhaps two years. In this dataset, we have the following expiry dates:

```
In [7]:
   aos['Expiry'].unique()


Out[7]:
   array(['2015-02-26T17:00:00.000000000-0700',
          '2015-03-05T17:00:00.000000000-0700',
          '2015-03-12T18:00:00.000000000-0600',
          '2015-03-19T18:00:00.000000000-0600',
          '2015-03-26T18:00:00.000000000-0600',
          '2015-04-01T18:00:00.000000000-0600',
          '2015-04-16T18:00:00.000000000-0600',
          '2015-05-14T18:00:00.000000000-0600',
          '2015-07-16T18:00:00.000000000-0600',
          '2015-10-15T18:00:00.000000000-0600',
          '2016-01-14T17:00:00.000000000-0700',
          '2017-01-19T17:00:00.000000000-0700'],
   dtype='datetime64[ns]')
```

For each option's expiration date, there are multiple options available, split between puts and calls, and with different strike values, prices, and associated risk values.

As an example, the option with the index `158` that expires on `2015-02-27` is for buying a call on AAPL with a strike price of $75. The price we would pay for each share of AAPL would be the bid price of $53.60. Options typically sell 100 units of the underlying security, and, therefore, this would mean that this option would cost 100 x $53.60 or $5,360 upfront:

```
In [8]:
   aos.loc[158]
```

```
Out[8]:
   Expiry               2015-02-27 00:00:00
   Strike                                75
   Type                                call
   IV                                   272
   Bid                                 53.6
   Ask                                 53.9
   Underlying_Price                     129
   Name: 158, dtype: object
```

This $5,360 does not buy us 100 shares of AAPL. It gives us the right to buy 100 shares of AAPL on `2015-02-27` at $75 per share. We should only buy if the price of AAPL is above $75 on `2015-02-27`. If not, we will have lost our premium of $5360 and purchasing below will only increase our loss.

Also, note that these quotes were retrieved on `2015-02-25`. This specific option has only two days until it expires. That has a huge effect on its pricing. We will examine the payout on options in detail in the next section, but in short, we can derive the following points from this purchase:

- We have paid $5,360 for the option to buy 100 shares of AAPL on `2015-02-27` if the price of AAPL is above $75 on that date.
- The price of AAPL when the option was priced was $128.79 per share. If we were to buy 100 shares of AAPL now, we would have paid $12,879.
- If AAPL is above $75 on `2015-02-27`, we can buy 100 shares for $7500.

There is not a lot of time between the quote and `Expiry` of this option. With AAPL being at $128.79, it is very likely that the price will be above $75 in two days' time.

Therefore, in two days' time:

- We can walk away if the price is $75 or above. Since we paid $5360, we probably wouldn't want to do that.
- At $75 or above, we can force the execution of the option, where we give the seller $7,500 and receive 100 shares of AAPL. If the price of AAPL is still $128.79 per share, then we will have bought $12,879 of AAPL for $7,500+$5,360, or $12,860 in total. Technically, we will have saved $19 over two days! But only if the price didn't drop.
- If, for some reason, AAPL dropped below $75 in two days, we kept our loss to our premium of $5,360. This is not great, but if we had bought $12,879 of AAPL on 2015-02-5 and it dropped to $74.99 on 2015-02-27, we would have lost $12,879 – $7,499, or $5,380. So, we actually would have saved $20 in loss by buying the call option.

It is interesting how this math works out. Excluding transaction fees, options are a zero-loss game. It just comes down to how much risk is involved in the option versus your upfront premium and how the market moves. If you feel you know something, it can be quite profitable. Of course, it can also be devastatingly unprofitable.

We will not examine the put side of this example. It would suffice to say it works out similarly from the side of the seller.

Implied volatility

There is one more field in our dataset that we didn't look at—**implied volatility** (**IV**). We won't get into the details of the mathematics of how this is calculated, but this reflects the amount of volatility that the market has factored into the option.

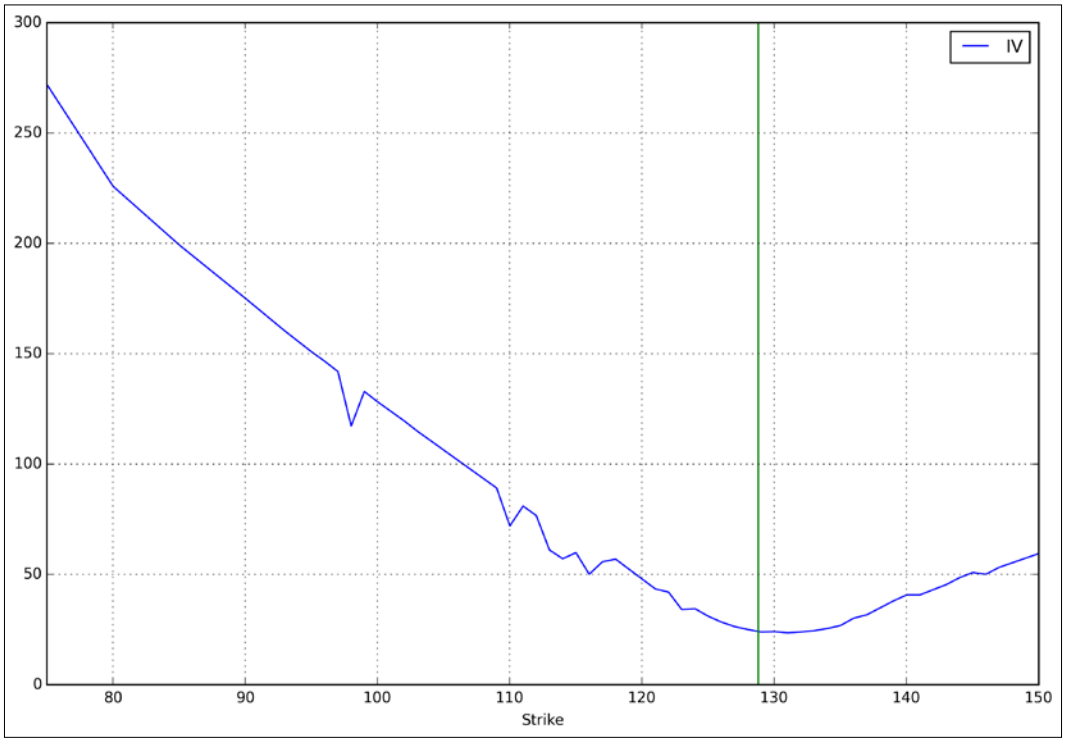
This is different to historical volatility (which is typically the standard deviation of the previous year of returns). We will look at pricing the option in a later section, but this comes out of pricing models as the amount of volatility needed for the strike price/premium value over the duration of the option contract to make those numbers work out nicely, as we have previously shown.

In general, it is informative to examine the IV relative to the strike price on a particular `Expiry` date. The following command shows this in tabular form for calls on `2015-02-27`:

```
In [9]:
   calls1 = aos[(aos.Expiry=='2015-02-27') & (aos.Type=='call')]
   calls1[:5]


Out[9]:
           Expiry  Strike  Type     IV   Bid    Ask  Underlying_Price
   158 2015-02-27      75  call 271.88 53.60  53.85            128.79
   159 2015-02-27      75   put 193.75  0.00   0.01            128.79
   190 2015-02-27      80  call 225.78 48.65  48.80            128.79
   191 2015-02-27      80   put 171.88  0.00   0.01            128.79
   226 2015-02-27      85  call 199.22 43.65  43.80            128.79
```

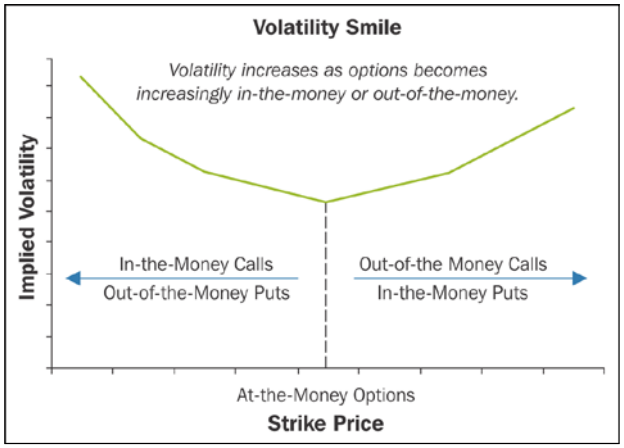
It appears that as the strike price approaches the underlying price, the implied volatility decreases. Plotting this shows it even more clearly:

```
In [10]:
   ax = aos[(aos.Expiry=='2015-02-27') & (aos.Type=='call')] \
           .set_index('Strike')[['IV']].plot(figsize=(12,8))
   ax.axvline(calls1.Underlying_Price.iloc[0], color='g');
```



The shape of this curve is important as it defines points where options are considered to be either in or out of the money. A call option is referred to as in the money when the options strike price is below the market price of the underlying instrument. A put option is in the money when the strike price is above the market price of the underlying instrument. Being in the money does not mean that you will profit; it simply means that the option is worth exercising.

Where and when an option is in our out of the money can be visualized by examining the shape of its implied volatility curve. Because of this curved shape, it is generally referred to as a volatility smile as both ends tend to turn upwards at both ends, particularly, if the curve has a uniform shape around its lowest point. This is demonstrated in the following graph, which shows the nature of being in or out of the money for both puts and calls:



A skew on the smile demonstrates a relative demand that is greater toward the option being either in or out of the money. When this occurs, the skew is often referred to as a smirk.

Volatility smirks

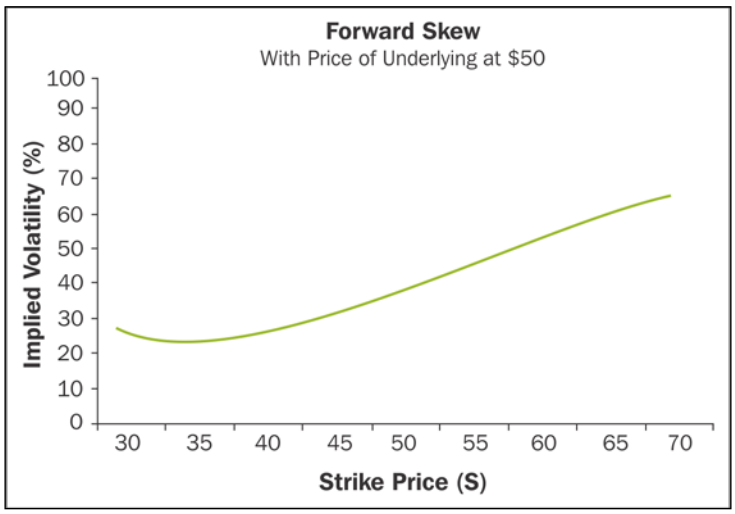
Smirks can either be reverse or forward. The following graph demonstrates a reverse skew, similar to what we have seen with our AAPL `2015-02-27` call:

In a reverse-skew smirk, the volatility for options at lower strikes is higher than at higher strikes. This is the case with our AAPL options expiring on `2015-02-27`. This means that the in-the-money calls and out-of-the-money puts are more expensive than the out-of-the-money calls and in-the-money puts.

A popular explanation for the manifestation of the reverse volatility skew is that investors are generally worried about market crashes and buy puts for protection. One piece of evidence supporting this argument is the fact that the reverse skew did not show up for equity options until after the crash of 1987.

Another possible explanation is that in-the-money calls have become popular alternatives to outright stock purchases as they offer leverage and, hence, increased ROI. This leads to greater demand for in-the-money calls and, therefore, increased IV at the lower strikes.

The other variant of the volatility smirk is the forward skew. In the forward-skew pattern, the IV for options at the lower strikes is lower than the IV at higher strikes. This suggests that the out-of-the-money calls and in-the-money puts are in greater demand compared to the in-the-money calls and out-of-the-money puts:



The forward-skew pattern is common for options in the commodities market. When supply is tight, businesses would rather pay more to secure supply than to risk supply disruption, for example, if weather reports indicate a heightened possibility of an impending frost, fear of supply disruption will cause businesses to drive up demand for out-of-the-money calls for the affected crops.

Calculating payoff on options

The payoff of an option is a relatively straightforward calculation based upon the type of the option and is derived from the price of the underlying security on expiry relative to the strike price. The formula for the call option payoff is as follows:

$$Payoff(call) = Max(S_T - X, 0)$$

The formula for the put option payoff is as follows:

$$Payoff(put) = Max(X - S_T, 0)$$

We will model both of these functions and visualize their payouts.

The call option payoff calculation

An option gives the buyer of the option the right to buy (a call option) or sell (a put option) an underlying security at a point in the future and at a predetermined price. A call option is basically a bet on whether or not the price of the underlying instrument will exceed the strike price. Your bet is the price of the option (the premium). On the expiry date of a call, the value of the option is `0` if the strike price has not been exceeded. If it has been exceeded, its value is the market value of the underlying security.

The general value of a call option can be calculated with the following function:

```
In [11]:
   def call_payoff(price_at_maturity, strike_price):
       return max(0, price_at_maturity - strike_price)
```

When the price of the underlying instrument is below the strike price, the value is `0` (out of the money). This can be seen here:

```
In [12]:
   call_payoff(25, 30)


Out[12]:
   0
```

When it is above the strike price (in the money), it will be the difference between the price and the strike price:

```
In [13]:
```

```
   call_payoff(35, 30)

Out[13]:
   5
```

The following function returns a `DataFrame` object that calculates the return for an option over a range of maturity prices. It uses `np.vectorize()` to efficiently apply the `call_payoff()` function to each item in the specific column of the `DataFrame`:

```
In [14]:
   def call_payoffs(min_maturity_price, max_maturity_price,
                    strike_price, step=1):
       maturities = np.arange(min_maturity_price,
                              max_maturity_price + step, step)
       payoffs = np.vectorize(call_payoff)(maturities, strike_price)
       df = pd.DataFrame({'Strike': strike_price, 'Payoff': payoffs},
                         index=maturities)
       df.index.name = 'Maturity Price'
    return df
```

The following command demonstrates the use of this function to calculate the payoff of an underlying security at finishing prices ranging from `10` to `25` and with a strike price of `15`:

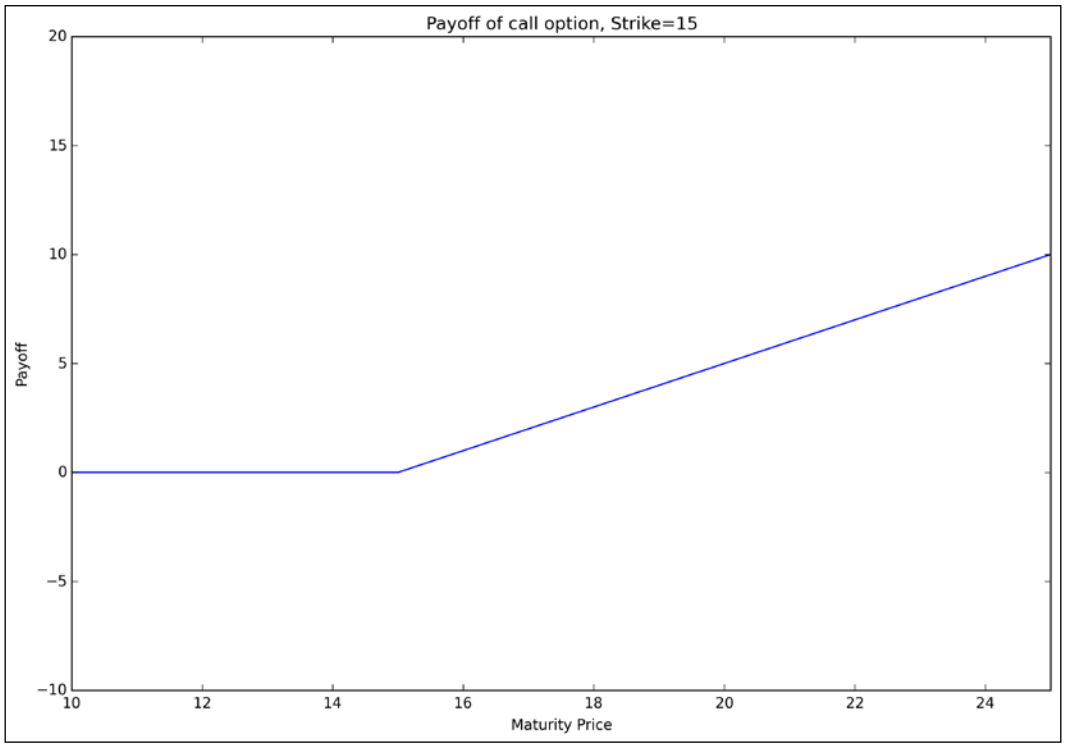
```
In [15]:
   call_payoffs(10, 25, 15)

Out[15]:
                    Payoff  Strike
   Maturity Price
   10                    0      15
   11                    0      15
   12                    0      15
   13                    0      15
   14                    0      15
   ...                 ...     ...
   21                    6      15
   22                    7      15
   23                    8      15
   24                    9      15
   25                   10      15
   [16 rows x 2 columns]
```

Using this result, we can visualize the payoffs using the following function:

```
In [16]:
   def plot_call_payoffs(min_maturity_price, max_maturity_price,
                         strike_price, step=1):
       payoffs = call_payoffs(min_maturity_price, max_maturity_price,
                              strike_price, step)
       plt.ylim(payoffs.Payoff.min() - 10, payoffs.Payoff.max() + 10)
       plt.ylabel("Payoff")
       plt.xlabel("Maturity Price")
       plt.title('Payoff of call option, Strike={0}'
                 .format(strike_price))
       plt.xlim(min_maturity_price, max_maturity_price)
       plt.plot(payoffs.index, payoffs.Payoff.values);
```

The payoffs are visualized as follows:

```
In [17]:
   plot_call_payoffs(10, 25, 15)
```



The put option payoff calculation

The value of a put option can be calculated with the following function:

```
In [18]:
   def put_payoff(price_at_maturity, strike_price):
       return max(0, strike_price - price_at_maturity)
```

While the price of the underlying is below the strike price, the value is `0`:

```
In [19]:
   put_payoff(25, 20)


Out[19]:
   0
```

When the price is below the strike price, the value of the option is the difference between the strike price and the price:

```
In [20]:
   put_payoff(15, 20)


Out[20]:
   5
```

This payoff for a series of prices can be calculated with the following function:

```
In [21]:
   def put_payoffs(min_maturity_price, max_maturity_price,
                   strike_price, step=1):
       maturities = np.arange(min_maturity_price,
                              max_maturity_price + step, step)
       payoffs = np.vectorize(put_payoff)(maturities, strike_price)
       df = pd.DataFrame({'Payoff': payoffs, 'Strike': strike_price},
                         index=maturities)
       df.index.name = 'Maturity Price'
       return df
```

The following command demonstrates the values of the put payoffs for prices of `10` through `25` with a strike price of `25`:

```
In [22]:
   put_payoffs(10, 25, 15)
```

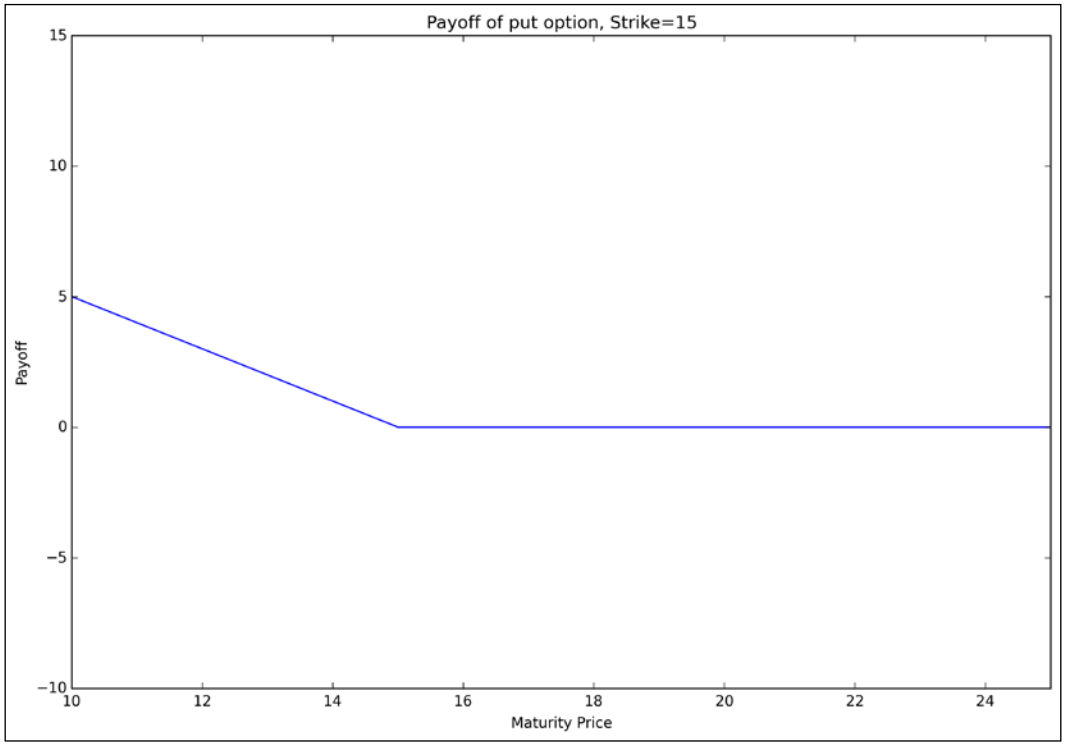
```
Out[22]:
                    Payoff  Strike
   Maturity Price
   10                    5      15
   11                    4      15
   12                    3      15
   13                    2      15
   14                    1      15
   ...                 ...     ...
   21                    0      15
   22                    0      15
   23                    0      15
   24                    0      15
   25                    0      15
   [16 rows x 2 columns]
```

The following function will generate a graph of payoffs:

```
In [23]:
   def plot_put_payoffs(min_maturity_price,
                        max_maturity_price,
                        strike_price,
                        step=1):
       payoffs = put_payoffs(min_maturity_price,
                             max_maturity_price,
                             strike_price, step)
       plt.ylim(payoffs.Payoff.min() - 10, payoffs.Payoff.max() + 10)
       plt.ylabel("Payoff")
       plt.xlabel("Maturity Price")
       plt.title('Payoff of put option, Strike={0}'
                 .format(strike_price))
       plt.xlim(min_maturity_price, max_maturity_price)
       plt.plot(payoffs.index, payoffs.Payoff.values);
```

The following command demonstrates the payoffs for prices between `10` and `25` with a strike price of `15`:

```
In [24]:
   plot_put_payoffs(10, 25, 15)
```



Profit and loss calculation

The general idea with an option is that you want to make a profit on speculation on the movement of the price of a security in the market, over a predetermined time frame.

The amount of profit or loss from the option can be calculated using a combination of the upfront premium and the payoff value of the option upon expiration. It is a zero-sum game as when a buyer profits by a certain amount, the seller loses the same amount, and vice versa.

The following table summarizes all of the profit and loss situations for both the buyer and seller when entering into options contracts:

Type	Scenario	Buyer or seller	Net profit or loss	Cash flow	At the end of the window period	Net amount
Call	The maturity price is above the strike price and the premium is less than the payoff	Buyer	Profit	-Premium	The buyer buys the underlying instrument at a discounted price from the seller	-Premium + payoff
		Seller	Loss	+Premium	The seller sells the underlying instrument to the buyer at a discount	-Payoff + premium
	The maturity price is above the strike price and the payoff is less than the premium	Buyer	Loss	-Premium	The buyer buys the underlying instrument at a discounted price from the seller	Payoff - premium
		Seller	Profit	+Premium	The seller sells the underlying instrument to the buyer at a discount	Premium - payoff
	The maturity price is below the strike price	Buyer	Loss	-Premium	Nil	-Premium
		Seller	Profit	+Premium	Nil	Premium

Type	Scenario	Buyer or seller	Net profit or loss	Cash flow	At the end of the window period	Net amount
Put	The maturity price is equal to or above the strike price	Buyer	Loss	-Premium	Nil	-Premium
		Seller	Profit	+Premium	Nil	-Premium
	The maturity price is less than the strike price and the payoff is greater than the premium	Buyer	Profit	-Premium	The buyer receives the underlying instrument from the seller	Payoff - premium
		Seller	Loss	+Premium		Premium - payoff
	The maturity price is less than the strike price and the payoff is less than the premium	Buyer	Loss	-Premium		-Premium + payoff
		Seller	Profit	+Premium		Premium - payoff

The call option profit and loss for a buyer

A buyer of a call will pay to the seller the premium to obtain the option being in a loss situation until the payoff exceeds the premium.

This can be demonstrated using the following function, which given the premium and strike price and returns a `DataFrame` of return values for a range of maturity prices for the buyer of a call:

```
In [25]:
      def call_pnl_buyer(premium, strike_price, min_maturity_price,
                         max_maturity_price, step = 1):
```

```
    payoffs = call_payoffs(min_maturity_price,
    max_maturity_price,
                           strike_price)
    payoffs['Premium'] = premium
    payoffs['PnL'] = payoffs.Payoff - premium
    return payoffs
```

The following command calculates the values of a call option starting at a price of `12` and with a strike price of `15` through the maturity values of `10` to `30`:

```
In [26]:
   pnl_buyer = call_pnl_buyer(12, 15, 10, 35)
   pnl_buyer
```

```
Out[26]:
                   Payoff  Price  Strike  PnL
   Maturity Price
   10                   0     12      15  -12
   11                   0     12      15  -12
   12                   0     12      15  -12
   13                   0     12      15  -12
   14                   0     12      15  -12
   ...                ...    ...     ...  ...
   31                  16     12      15    4
   32                  17     12      15    5
   33                  18     12      15    6
   34                  19     12      15    7
   35                  20     12      15    8

   [26 rows x 4 columns]
```

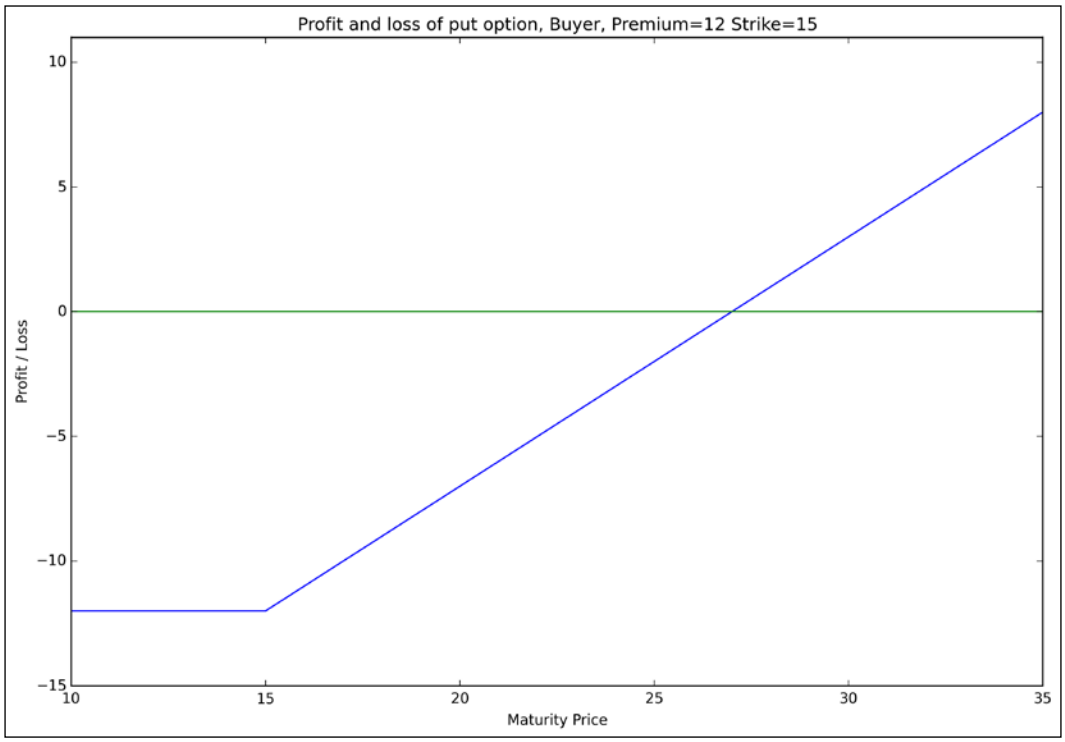
The following function will visualize information in this `DataFrame`:

```
In [27]:
   def plot_pnl(pnl_df, okind, who):
       plt.ylim(pnl_df.Payoff.min() - 10, pnl_df.Payoff.max() + 10)
       plt.ylabel("Profit / Loss")
       plt.xlabel("Maturity Price")
```

```
        plt.title('Profit and loss of {0} option, {1}, Premium={2}
Strike={3}'
                  .format(okind, who, pnl_df.Premium.iloc[0],
                          pnl_df.Strike.iloc[0]))
        plt.ylim(pnl_df.PnL.min()-3, pnl_df.PnL.max() + 3)
        plt.xlim(pnl_df.index[0], pnl_df.index[len(pnl_df.index)-1])
        plt.plot(pnl_df.index, pnl_df.PnL)
        plt.axhline(0, color='g');
```

This visualizes the particular `DataFrame` with the following chart:

```
In [28]:
   plot_pnl(pnl_buyer, "put", "Buyer")
```



The profit and loss stays at a loss of the initial premium until the payoff begins to increase from `0` as the maturity price exceeds the strike price. There is a loss until the payoff exceeds the premium, which, in this case, is at $27 (the premium and the strike price).

The call option profit and loss for the seller

A seller of a call will initially profit from the receipt of the premium from the buyer. The profit for a seller will be the premium as long as the price at maturity is below the strike price. As the payoff increases for the buyer, the profit for the seller decreases and will eventually become a loss once the buyer moves into profit.

This can be demonstrated using the following function, which, given the premium and strike price, returns a `DataFrame` of returns values for a range of maturity prices for the seller of a call:

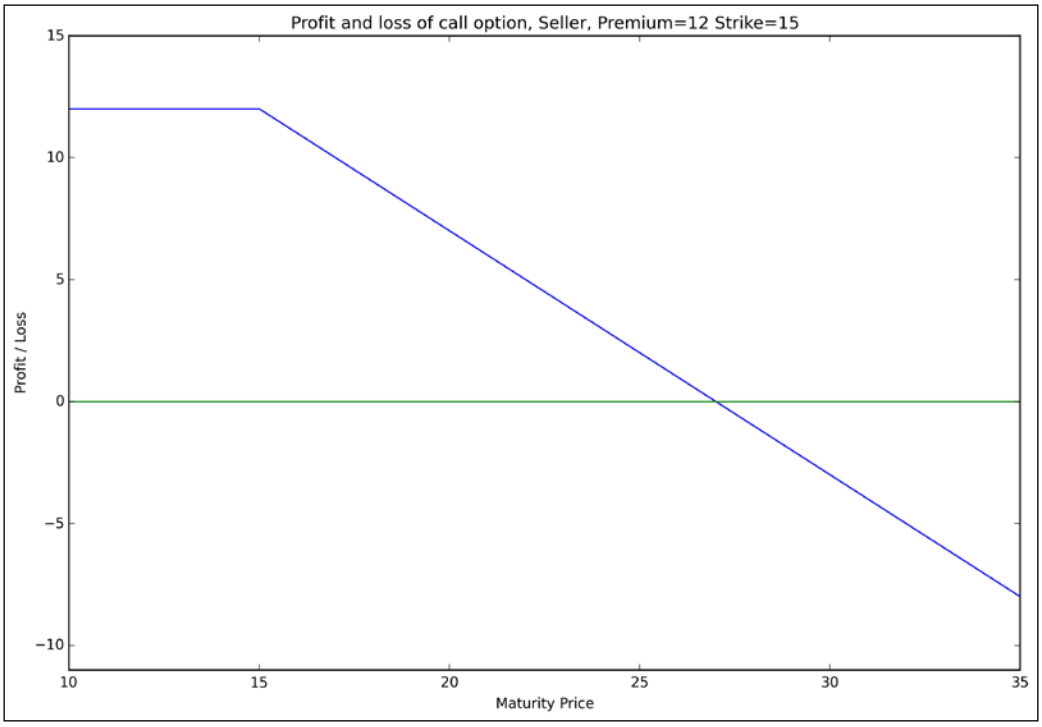
```
In [29]:
   def call_pnl_seller(premium, strike_price, min_maturity_price,
                       max_maturity_price, step = 1):
       payoffs = call_payoffs(min_maturity_price, max_maturity_price,
                              strike_price)
       payoffs['Premium'] = premium
       payoffs['PnL'] = premium - payoffs.Payoff
       return payoffs
```

The following command calculates the values of a call option starting at a price of `12` and with a strike price of `15` through the maturity values of `10` to `30`:

```
In [30]:
   pnl_seller = call_pnl_seller(12, 15, 10, 35)
   pnl_seller

Out[30]:                 Payoff  Strike  Premium  PnL
   Maturity Price
   10                         0      15       12   12
   11                         0      15       12   12
   12                         0      15       12   12
   13                         0      15       12   12
   14                         0      15       12   12
   ...                      ...     ...      ...  ...
   31                        16      15       12   -4
   32                        17      15       12   -5
   33                        18      15       12   -6
   34                        19      15       12   -7
   35                        20      15       12   -8
   [26 rows x 4 columns]
```

This visualizes a particular `DataFrame` with the following chart:

```
In [31]:
   plot_pnl(pnl_seller, "call", "Seller")
```



The profit and loss stays at a profit matching the premium until the payoff begins to increase from `0` as the maturity price exceeds the strike price. There is a profit obtained until the payoff amount exceeds the premium, which in this case is at $27 (the premium + the strike price), at which point the seller of the call will increasingly be at a loss as the maturity value increases.

Combined payoff charts

There will be many instances where you will see the payoffs/profit and loss for both the buy and seller represented on a single chart. The following function will do this for us:

```
Out[32]:
   def plot_combined_pnl(pnl_df):
       plt.ylim(pnl_df.Payoff.min() - 10, pnl_df.Payoff.max() + 10)
       plt.ylabel("Profit / Loss")
```

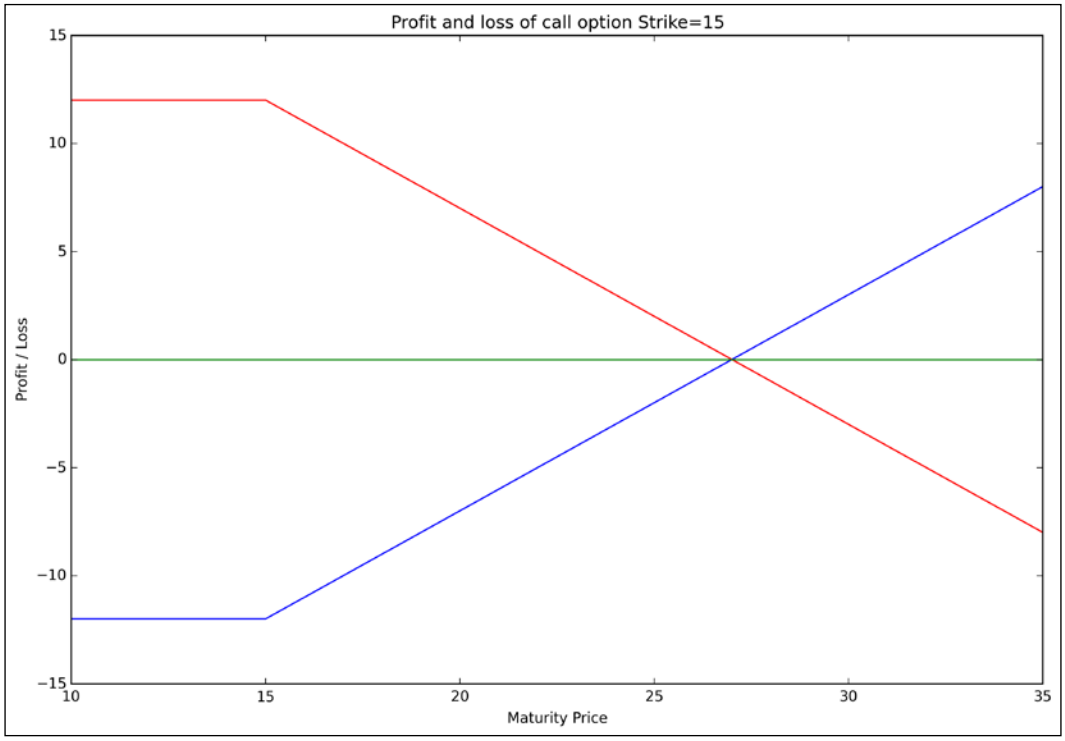
```
    plt.xlabel("Maturity Price")
    plt.title('Profit and loss of call option Strike={0}'
              .format(pnl_df.Strike.iloc[0]))
    plt.ylim(min(pnl_df.PnLBuyer.min(), pnl_df.PnLSeller.min())-3,
             max(pnl_df.PnLBuyer.max(), pnl_df.PnLSeller.max())+3)
    plt.xlim(pnl_df.index[0], pnl_df.index[len(pnl_df.index)-1])
    plt.plot(pnl_df.index, pnl_df.PnLBuyer, color='b')
    plt.plot(pnl_df.index, pnl_df.PnLSeller, color='r')
    plt.axhline(0, color='g');
```

This function expects to be given a `DataFrame`, which combines data from both the profit and loss functions' calls and puts. This `DataFrame` can be constructed as follows:

```
In [33]:
   pnl_combined = pd.DataFrame({'PnLBuyer': pnl_buyer.PnL,
                                'PnLSeller': pnl_seller.PnL,
                                'Premium': pnl_buyer.Premium,
                                'Strike': pnl_buyer.Strike,
                                'Payoff': pnl_buyer.Payoff})
   pnl_combined

Out[33]:
                    Payoff  PnLBuyer  PnLSeller  Premium  Strike
   Maturity Price
   10                    0       -12         12       12      15
   11                    0       -12         12       12      15
   12                    0       -12         12       12      15
   13                    0       -12         12       12      15
   14                    0       -12         12       12      15
   ...                 ...       ...        ...      ...     ...
   31                   16         4         -4       12      15
   32                   17         5         -5       12      15
   33                   18         6         -6       12      15
   34                   19         7         -7       12      15
   35                   20         8         -8       12      15
   [26 rows x 5 columns]
```

Now, passing this in to the function, we are presented with the following graph with both series of profit and loss plotted:

```
In [34]:
   plot_combined_pnl(pnl_combined)
```



This shows how the overall effect of buying and selling an option is a zero-sum game. There are fixed losses or gains for the buyer and seller as long as the maturity price is below the strike price. A maturity price above the strike price begins to flow value back to the buyer from the seller. Conceptually, there is unlimited upside for the buyer and unlimited downside for the seller.

The put option profit and loss for a buyer

A buyer of a put pays a premium to the put seller. They are at a loss of the premium if the maturity price exceeds the strike price. As the maturity price falls below the strike price at maturity, the loss will decrease. There will be an overall loss until the payoff exceeds the premium.

This can be demonstrated using the following function which, given the premium and strike price, returns a `DataFrame` of returns values for a range of maturity prices for the buyer of a put option:

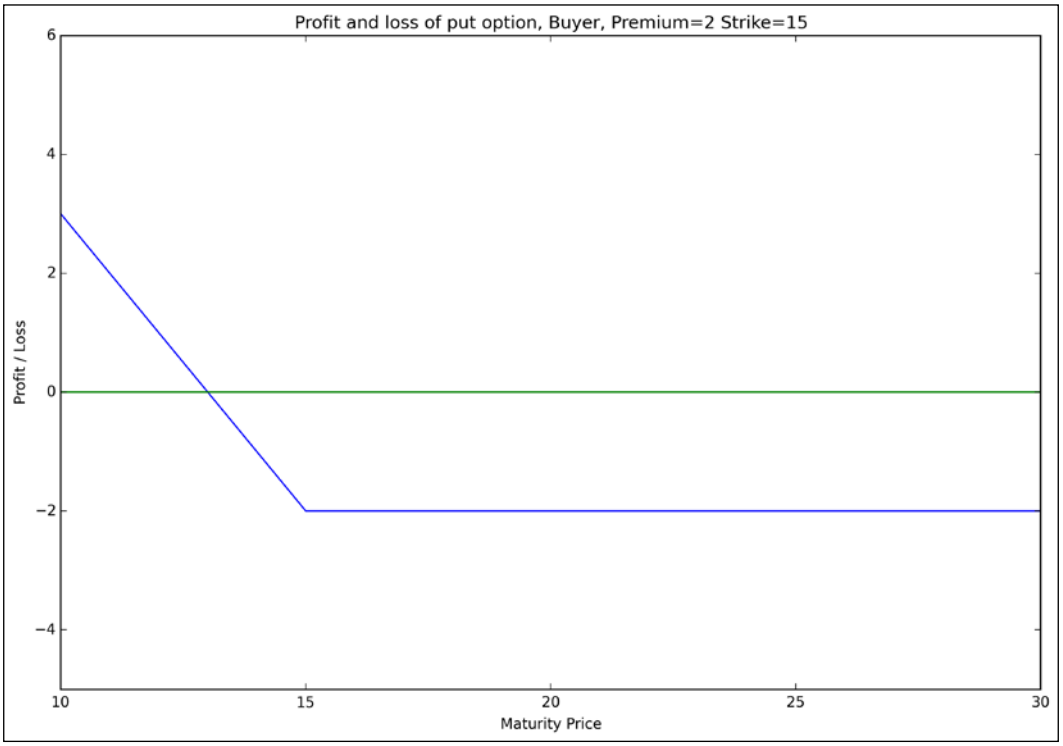
```
In [35]:
   def put_pnl_buyer(premium, strike_price, min_maturity_price,
                     max_maturity_price, step = 1):
       payoffs = put_payoffs(min_maturity_price, max_maturity_price,
                             strike_price)
       payoffs['Premium'] = premium
       payoffs['Strike'] = strike_price
       payoffs['PnL'] = payoffs.Payoff - payoffs.Premium
       return payoffs
```

The following command calculates the profit and loss of a put option for the buyer starting at a price of `2` and with a strike price of `15` through the maturity values of `10` to `30`:

```
In [36]:
   pnl_put_buyer = put_pnl_buyer(2, 15, 10, 30)
   pnl_put_buyer


Out[36]:
                    Payoff  Strike  Premium  PnL
   Maturity Price
   10                    5      15        2    3
   11                    4      15        2    2
   12                    3      15        2    1
   13                    2      15        2    0
   14                    1      15        2   -1
   ...                 ...     ...      ...  ...
   26                    0      15        2   -2
   27                    0      15        2   -2
   28                    0      15        2   -2
   29                    0      15        2   -2
   30                    0      15        2   -2
   [21 rows x 4 columns]
```

The following function will visualize information in this `DataFrame`:

```
In [37]:
   plot_pnl(pnl_put_buyer, "put", "Buyer")
```



There is a tendency to read this chart as the put buyer profiting at the purchase of the put option. Remember that the horizontal axis is not time that increases from left to right. Although it looks as though the buyer profits by $3 at the onset of purchasing the option, this chart really shows how profit and loss varies at maturity for different maturity prices. As long as the maturity price is greater than the strike price, there is only a loss of the amount of the premium. The more the maturity price finishes below the strike price, the better the chance to earn profit.

The put option profit and loss for the seller

A seller of a put receives the premium from the buyer of the put. They have a profit of the premium if the maturity price exceeds the strike price. As the maturity price falls below the strike price at maturity, the profit will decrease by the amount of the payoff.

This can be demonstrated using the following function, which, given the premium and strike price, returns a `DataFrame` of returns values for a range of maturity prices for the seller of a put option:

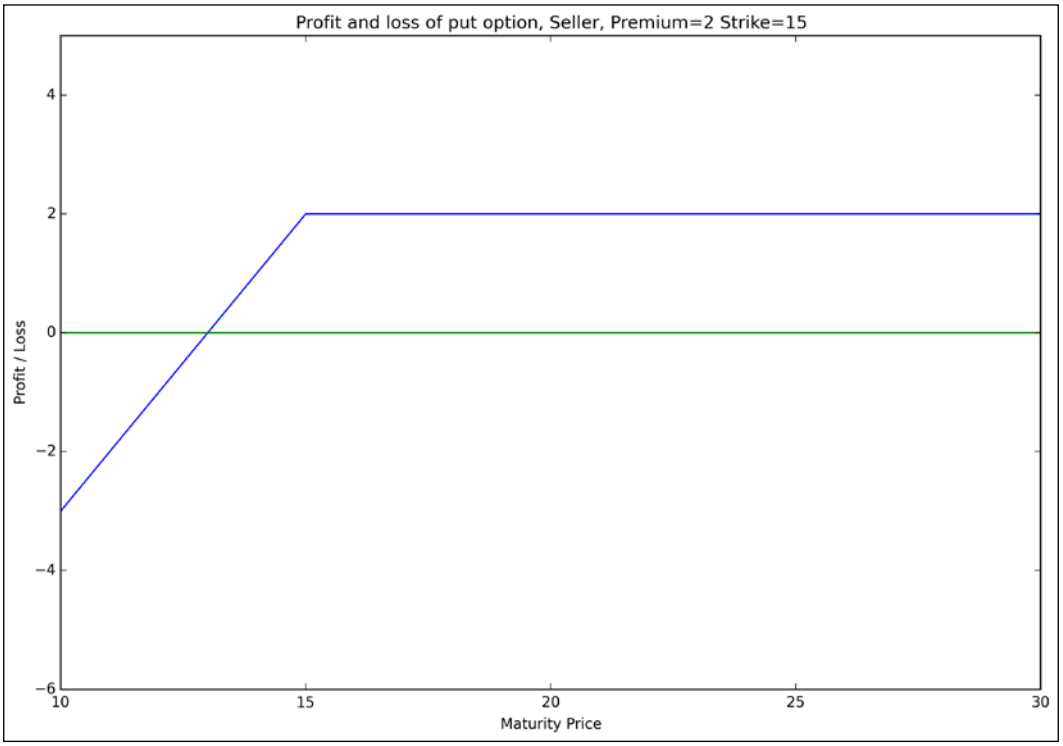
```
In [38]:
   def put_pnl_seller(premium, strike_price, min_maturity_price,
                      max_maturity_price, step = 1):
       payoffs = put_payoffs(min_maturity_price, max_maturity_price,
                             strike_price)
       payoffs['Premium'] = premium
       payoffs['Strike'] = strike_price
       payoffs['PnL'] = payoffs.Premium - payoffs.Payoff
       return payoffs
```

The following command calculates the profit and loss of a put option for the seller starting at a price of `2` and with a strike price of `15` through the maturity values of `10` to `30`:

```
In [39]:
   pnl_put_seller = put_pnl_seller(30, 45, 20, 50)
   pnl_put_seller


Out[39]:
                    Payoff  Strike  Premium  PnL
   Maturity Price
   10                    5      15        2   -3
   11                    4      15        2   -2
   12                    3      15        2   -1
   13                    2      15        2    0
   14                    1      15        2    1
   ...                 ...     ...      ...  ...
   26                    0      15        2    2
   27                    0      15        2    2
   28                    0      15        2    2
   29                    0      15        2    2
   30                    0      15        2    2
   [21 rows x 4 columns]
```

The following function will visualize information in this `DataFrame`:

```
In [40]:
   plot_pnl(pnl_put_seller, "put", "Seller")
```



The pricing of options

There are two general styles of options: European and American. A European option is an option that cannot be exercised before its expiration date. An American option can be exercised at any point before its expiration date. American options are the most common form of options traded in the market.

The pricing model of the two styles of options is significantly different. Since a European option can only be exercised at its expiration, there exists a closed form calculation for its market price. The common form of modeling for a European option is the Black-Scholes pricing model.

The pricing of American options is complicated by their ability to be exercised at any time, which prevents them having a closed-form pricing model. However, there are several ways to price an American option, one of which we will examine later in the chapter and is known as the binomial tree method.

A general characteristic of an American option compared to a European option is that its price generally will be higher due to the flexibility and increased risk on the counterparty side.

We will examine the pricing of European options using the Black-Scholes formula. Our purpose is not to derive a complete understanding of how the prices are derived but to use a pricing library to verify the price and implied volatility of options retrieved from Yahoo! Finance.

Additionally, we will examine several underlying characteristics of the options referred to as The Greeks, which are various partial derivatives of the Black-Scholes formula relative to the various parameters of the function. These values are often used in decision making with respect to the purchase of options.

The pricing of options with Black-Scholes

The Black-Scholes formula was developed by Fischer Black and Myron Scholes and is a stochastic partial-differential equation that estimates the price of an option, specifically a European option, which is an option that can only be exercised at the end of its life. This is in contrast to an American option, which can be exercised at any point after its purchase.

The basic idea behind Black-Sholes is to determine the value today of an options contract for an underlying security in a year. The contract will have different values depending upon whether the stock goes up or down, so the payoff curve is not symmetrical. The model helps us to derive an underlying measure of the probabilities of the underlying security ending up at various values at the end of the year. If we can determine this, then we can also estimate a value for the contract.

The Black-Scholes model also makes several assumptions to keep the modeling simple:

- There is no arbitrage
- There is the ability to borrow money at a constant risk-free interest rate throughout the life of the option
- There are no transaction costs
- The pricing of the underlying security follows a Brownian motion with constant drift and volatility
- No dividends are paid from the underlying security

This seems to be a list of very important assumptions but it is needed to get a baseline model in place. More complicated scenarios can then be handled with other derivations, but even with these assumptions, the resulting model is quite representative of actual prices (as we will see).

Deriving the model

There are three primary factors that are taken into account for determining the value of an option:

- The value of the cash to buy the option
- The value of the underlying security that is received (if any)
- The volatility of the underlying price during the life of the option

We have seen these three factors taken into account in our payoff models. We now need to quantify these a bit more to be able to work out their expected values and derive a value for the contract.

The value of the cash to buy

If the option is exercised, then the cash is paid only if the underlying stock price is above the strike at maturity. Therefore, we need to determine the expected value based upon the probability that the stock finishes above the strike price. The strike price will be referred to as K, and the probability of the stock finishing above K will be referred to as $N(d_2)$. The expected value is then $N(d_2)K$ with $N()$ representing the cumulative normal function. The d_2 variable represents a formulation of the probability of the option exceeding the strike price (a little more on this later).

Given that the expected value is $N(d_2)K$, this amount can be discounted using $e^{-r(T-t)}$ to give us the value of the cash to buy the option today as $N(d_2)Ke^{-r(T-t)}$.

The value of the stock received

If the option is exercised, then we take possession of the underlying security at its value in the market at the maturity of the option. It happens that the expected value of this is proportional to the current value of the stock, which is referred to as S. In the Black-Scholes model, this expected value is referred to as $N(d_1)S$.

$N(d_1)$ represents the proportion of the value of the current value of the stock, S at maturity of the option only if the option is exercised and 0 otherwise. Like d_2, d_1 will be stated a little later.

The formulas

Options are either calls or puts, so there are two derivations of the model. The simpler of the two is the model for call options:

$$C(S,t) = N(d_1)S - N(d_2)Ke^{-r(T-t)}$$

This states that the value of the call is the difference between the stock price and the strike price using the probability scaling of each and discounting the strike price.

The formula for a put is slightly more complicated but similar:

$$P(S,t) = Ke^{-r(T-t)} - S + C(S,t) = N(-d_2)Ke^{-r(T-t)} - N(-d_1)S$$

d1 and d2

Finally, we get to d_1 and d_2. These formulas are at the heart of the Black-Scholes model. The mathematics of d_1 and d_2 are fairly complex and represent the probability scale factors for the stock price (d_1) and strike price (d_2) using the cumulative normal function $N()$. These will be presented as follows without further explanation in this text. The formula for d_1 is as follows:

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left[r + \frac{\sigma^2}{2}\right](T-t)\right]$$

The formula for d_2 is as follows:

$$d_2 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left[r - \frac{\sigma^2}{2}\right](T-t)\right] = d_1 - \sigma\sqrt{T}$$

These appear complex (and their derivation is) but are easily implemented in a programming language with the values simply plugged in. Also, the volatility of the underlying price is represented in these equations by the sigma variable.

The parameters that can be plugged in are the following:

- **N**: The cumulative normal function
- **T**: Time to maturity expressed in years
- **S**: The stock price or other underlying assets
- **K**: The strike price
- **r**: The risk-free interest rate

You may have noticed that we have not parameterized the volatility. This is one of the things you need to remember using Black-Scholes. The volatility will be implied via the other parameters.

Now, with this all in hand, we can now implement the Black-Sholes algorithm in Python.

Black-Scholes using Mibian

For the sake of brevity, we will not get into the actual implementation of Black-Scholes in Python. Instead, we will use a small but convenient library: MibianLib. MibianLib is available at `http://code.mibian.net/` and is open source. It provides several methods for options price calculation, one of which is Black-Scholes. You can examine the implementation to verify the previous formulations.

Now, let's examine the basic use of Mibian to calculate values using Black-Scholes. To do this, we will examine two options that we retrieved from Yahoo! Finance earlier in the chapter—the put and call expiring on `2015-01-15` with IV of 57.23 (the put) and 52.73 (the call):

```
In [41]:
   aos[aos.Expiry=='2016-01-15'][:2]


Out[41]:
         Expiry  Strike  Type     IV    Bid    Ask  Underlying_Price
   0 2016-01-15   34.29  call  57.23  94.10  94.95            128.79
   1 2016-01-15   34.29   put  52.73   0.01   0.07            128.79
```

At the time of retrieving these, these options are `324` days from expiring:

```
In [42]:
   date(2016, 1, 15) - date(2015, 2, 25)


Out[42]:
   datetime.timedelta(324)
```

We have now collected all of the parameters to use the Black-Scholes pricing (using an assumed 1 percent interest rate):

```
In [43]:
   import mibian
   c = mibian.BS([128.79, 34.29, 1, 324], 57.23)
```

The call price can be retrieved via the `.callPrice` property:

```
In [43]:
   c.callPrice


Out[44]:
   94.878970089456217
```

Our result is a few cents off the actual quoted bid but between the bid and ask prices. Given that we assumed a 1 percent interest rate, the result is right in the range we would expect.

The put price is retrieved via the `.putPrice` property:

```
In [45]:
   c.putPrice


Out[45]:
   0.075934592996542705
```

This is very close to the ask value of the put option.

We can also use Mibian to calculate the implied volatility:

```
In [46]:
   c = mibian.BS([128.79, 34.29, 1, 324],
                 callPrice=94.878970089456217 )
Out[46]:
   57.22999572753906
```

Charting option price change over time

It can be useful to plot the price of an option until its expiration. We can do this by varying the time to expiration and plotting the results. This can be done very easily using pandas.

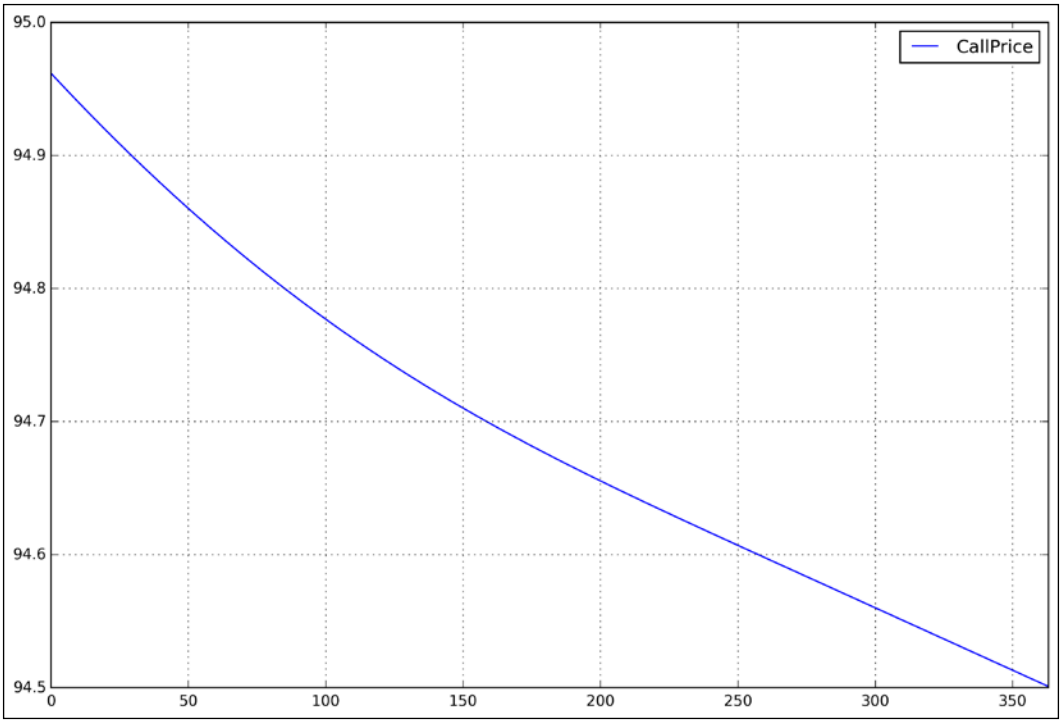
The following command calculates the call price for the AAPL option, varying from 1 to 364 days to expiry, and plots the change in price showing that the price of the call decreases as the number of days to expiry increases:

```
In [47]:
   df = pd.DataFrame({'DaysToExpiry': np.arange(364, 0, -1)})
   df
```

```
Out[47]:
        DaysToExpiry
   0             364
   1             363
   2             362
   3             361
   4             360
   ..            ...
   359             5
   360             4
   361             3
   362             2
   363             1
   [364 rows x 1 columns]

In [48]:
   bs_v1 = mibian.BS([128.79, 34.29, 1, 324], volatility=57.23)
   calc_call = lambda r: mibian.BS([128.79, 34.29, 1,
                                    r.DaysToExpiry],
                                   volatility=57.23).callPrice
   df['CallPrice'] = df.apply(calc_call, axis=1)
   df

Out[48]:
        DaysToExpiry  CallPrice
   0             364      94.96
   1             363      94.96
   2             362      94.96
   3             361      94.96
   4             360      94.95
   ..            ...        ...
   359             5      94.50
   360             4      94.50
   361             3      94.50
   362             2      94.50
   363             1      94.50
   [364 rows x 2 columns]
```

The following graph shows the call price decreasing as the days to expiry also decreases:

```
In [49]:
   df[['CallPrice']].plot();
```



The Greeks

The Greeks are quantities representing the sensitivity of the price of options to the change in the underlying parameters of the valuation of the derivative. The first-order Greeks of options represent the change value relative to the change in price, volatility, and time to expiry. Second-order and third-order Greeks do exist, but we will only focus on the first-order Greeks and a single second-order Greek known as Gamma.

The first-order Greeks are named and represented in the following table:

Name	Description
Delta	This is the rate of change of the option value with respect to a change in the price of the underlying security
Vega	This is the rate of change of the option value with respect to a change in the volatility of the underlying security
Theta	This is the rate of change of the option value with respect to the time to expiry
Rho	This is the rate of change of the option value with respect to the interest rate
Gamma	This is the rate of change of the Delta Greek with respect to a change in the price of the underlying security

The Greeks are important tools in risk management to manage the exposure of individual investments or combinations, such as in an investment portfolio. We will not get into the detailed use for risk management as that is beyond the scope of this book (and pandas), but they are worth mentioning in a chapter on options pricing.

Calculation and visualization

The Greeks in Black-Scholes are straightforward to calculate and are given with the following formulas:

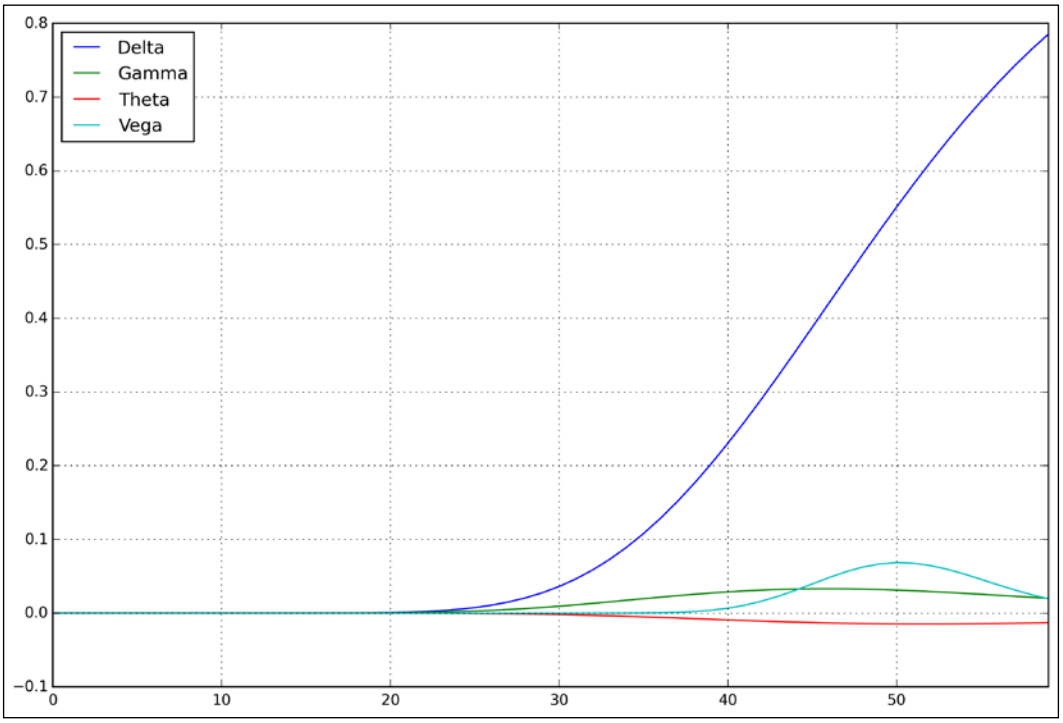
Greek	Derivation	Calls	Puts
Delta	$\frac{\partial C}{\partial S}$	$N(d_1)$	$-N(d_1) = N(d_1) - 1$
Gamma	$\frac{\partial^2 C}{\partial S^2}$	$\frac{N'(d_1)}{S\sigma\sqrt{T-t}}$	
Vega	$\frac{\partial C}{\partial \sigma}$	$SN'(d_1)\sqrt{T-t}$	
Theta	$\frac{\partial C}{\partial t}$	$-\frac{SN'(d_1)\sigma}{2\sqrt{T-t}} - rKe^{-r(T-t)}N(d_2)$	$-\frac{SN'(d_1)\sigma}{2\sqrt{T-t}} + rKe^{-r(T-t)}N(-d_2)$
Rho	$\frac{\partial C}{\partial r}$	$K(T-t)e^{-r(T-t)}N(d_2)$	$-K(T-t)e^{-r(T-t)}N(-d_2)$

We will not examine their implementation in this book, especially since they are implemented in Mibian. However, we will demonstrate how the Greeks vary in value by creating a `DataFrame` to alternate the values of the input in the Black-Scholes pricing algorithm:

```
In [50]:
   greeks = pd.DataFrame()
   delta = lambda r: mibian.BS([r.Price, 60, 1, 180],
                               volatility=30).callDelta
   gamma = lambda r: mibian.BS([r.Price, 60, 1, 180],
                               volatility=30).gamma
   theta = lambda r: mibian.BS([r.Price, 60, 1, 180],
                               volatility=30).callTheta
   vega = lambda r: mibian.BS([r.Price, 60, 1, 365/12],
                              volatility=30).vega

   greeks['Price'] = np.arange(10, 70)
   greeks['Delta'] = greeks.apply(delta, axis=1)
   greeks['Gamma'] = greeks.apply(gamma, axis=1)
   greeks['Theta'] = greeks.apply(theta, axis=1)
   greeks['Vega'] = greeks.apply(vega, axis=1)
   greeks[:5]

Out[50]:
       Price     Delta     Gamma     Theta      Vega
   0      10  2.73e-17  1.10e-16 -1.37e-18  1.96e-96
   1      11  1.15e-15  4.00e-15 -6.00e-17  1.17e-86
   2      12  2.94e-14  8.88e-14 -1.59e-15  3.36e-78
   3      13  4.99e-13  1.32e-12 -2.78e-14  8.21e-71
   4      14  6.05e-12  1.42e-11 -3.45e-13  2.63e-64
```

The following plot demonstrates how the different values for `Delta`, `Gamma`, `Theta`, and `Vega` change for this particular option relative to change in their respective parameters:

```
In [51]:
   greeks[['Delta', 'Gamma', 'Theta', 'Vega']].plot();
```



Summary

In this chapter, we examined several techniques for using pandas to calculate the prices of options, their payoffs, and the profit and loss for the various combinations of calls and puts for both buyers and sellers. We started with a brief introduction to options, covered how to load current market data for options from Yahoo! Finance, and then examined the properties of the data retrieved from the web services.

We then examined the pricing of options using Black-Scholes with a brief explanation of how the algorithm models option prices. We also used the Mibian library to calculate prices using Black-Scholes. We finished with a brief explanation of the Greeks and how to calculate their values for various configurations of options.

In the next chapter, we will look at the modeling of investment portfolios using Python and pandas and how we can calculate optimal portfolios that balance risk and return for different investor types.

9
Portfolios and Risk

A portfolio is a grouping of financial assets, which may include stocks, bonds, and mutual funds. It is generally accepted that a portfolio is designed based upon an investor's risk tolerance, time frames, and investment goals. The allocation of the assets in a portfolio, referred to as asset allocation, influences the risk/reward ratio of the portfolio. The specific assets in a portfolio and the relative weighting of the assets within the portfolio are designed to maximize the expected return, while also minimizing the risk.

The process of determining the proper assets and their proportion relative to each other within a portfolio involves a concept known as **modern portfolio theory** (**MPT**). This is a theory in finance that has evolved since the 1950s and describes the mathematics of constructing an optimal portfolio based upon risk and return parameters. This involves selecting assets that are correlated based upon historical returns, in such a manner that they function to diversify the portfolio.

In this chapter, we will examine the concepts of modern portfolio theory. We will first start with an overview of MPT and how it utilizes a concept known as the 'efficient frontier' to determine an optimal portfolio. We will then examine a means of modeling a portfolio with pandas, and then implement the mathematics of MPT to calculate optimum portfolios and determine and visualize the efficient frontier for a particular mix of assets. The chapter then closes of with a brief discussion of Value at Risk, which helps us to understand the level potential loss that can be expected in a portfolio for a specific period of time.

In this chapter, we will cover the following:

- An overview of modern portfolio theory
- Mathematical models of portfolios
- Risk and expected return
- The concepts of diversification and the efficient frontier

- Modeling a portfolio with pandas
- Gathering historical stock data within a portfolio
- Modeling different weights of assets in a portfolio
- Optimization and minimization using SciPy
- Calculating the Sharpe ratio of a portfolio
- Constructing an efficient portfolio
- Visualizing the efficient frontier for a set of assets
- Computing **Value at Risk** (**VaR**)

Notebook setup

The examples in this chapter will be based upon the following configuration in IPython. One main difference in this setup is that in this chapter, we will be using SciPy, specifically its optimization and statistical features, so this has imports that are required for several of the examples:

```
In [1]:
   import pandas as pd
   import numpy as np
   import pandas.io.data as web
   from datetime import datetime

   import scipy as sp
   import scipy.optimize as scopt
   import scipy.stats as spstats

   import matplotlib.pyplot as plt
   import matplotlib.mlab as mlab
   %matplotlib inline

   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', 82)
   pd.set_option('precision', 3)
```

An overview of modern portfolio theory

Modern portfolio theory (**MPT**) is a theory of finance that attempts to maximize the expected return on a set of investments (known as the portfolio), relative to the overall risk of the combined items in the portfolio. The concept is that given a particular level of risk, the return will be maximized for that risk. This is common in retirement plans. The younger the investor and the smaller the amount in the portfolio, the more there is a willingness to take risks on higher returns. As the investor comes close to retirement and the total value of the portfolio is higher, the more likely they are to take lower risks, to ensure that the base of the portfolio is not lost but that at the tradeoff of potential gains being lower.

MPT provides a mathematical model of diversified investment with the goal of selecting a collection of investments that has a combined risk that is less than any individual asset in the portfolio. This is achievable by selecting individual investments that have opposite correlations such that when one particular investment goes down in value, another gains similarly in value and the overall net of the portfolio remains consistent or at least minimizes the loss during downturns. However, at the same time, this may also lower the overall gains in upturns. And additionally, diversification has a tendency to also lower risk even if various assets in the portfolio are not negatively correlated as the diversity itself tends to give an overall less risky portfolio.

MPT assumes an individual investment's returns as normally distributed and then defines risk as the standard deviation of the returns. It then models a portfolio as a weighted combination of the assets such that the return of the overall portfolio is a weighted sum of the combination of the returns of the assets. Then, by selecting a set of investments that are not perfectly correlated, MPT attempts to reduce the total variance of the overall portfolio return.

MPT was developed in the 1950s and through to the 1970s and represented a significant advance in financial modeling. As a theory, it is interesting and does have practical applications. But like other models of finance (for example, Black-Scholes), it is heavily dependent on those assumptions and can lead to suboptimal results when those conditions are not met. Nonetheless, it is an important financial concept—one that can be implemented effectively using pandas and Python—and is important to understand before branching out into more detailed models.

Concept

The basic idea behind MPT is that assets in a portfolio should not be selected individually based upon their individual performance. It is instead important to consider how each asset changes in value relative to other assets in the portfolio. This represents a tradeoff between risk and expected return. The stocks in an efficient portfolio are chosen based on the investor's risk tolerance, with an efficient portfolio having at least two stocks above the minimum variance portfolio. For a given amount of risk, MPT describes how to select a portfolio out of a set of investments that has the highest expected return while being at or below the specified risk level. On the flip side, for a given return, MPT specifies how to select a portfolio with the least possible risk.

Mathematical modeling of a portfolio

In this chapter, we will examine the classical model of MPT. There have been many extensions, but we will focus on the core.

Risk and expected return

A fundamental assumption of MPT is that investors are risk averse. This means that given two portfolios that offer the same expected return, the investor will prefer the less risky portfolio. Therefore, an investor will only take on a riskier portfolio if higher expected returns make it worthwhile. And conversely, an investor wanting higher expected returns must accept greater risk.

MPT makes the assumption that the standard deviation of returns can be used as an accurate representation of risk. This is valid if asset returns are normally jointly distributed, which are otherwise elliptically distributed.

Then, under the model:

- Portfolio return is the proportion-weighted combination of the constituents of the returns of the assets
- Portfolio volatility is a function of the correlations of the constituent assets, for all pairs of assets *(i, j)*.

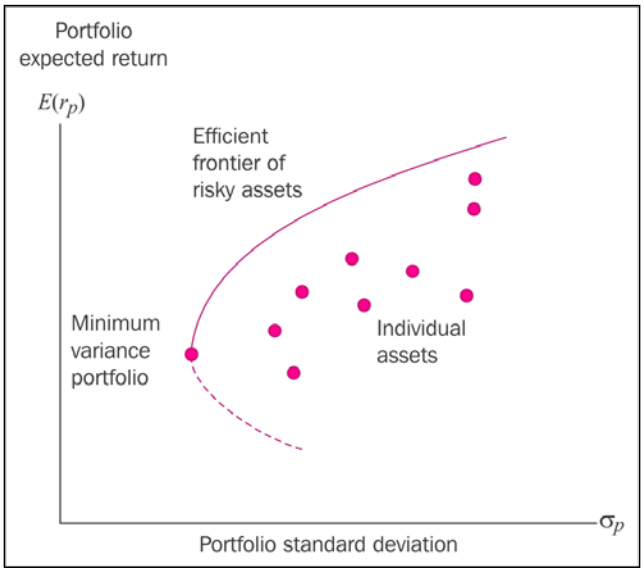
This goes on up to *n* assets in a portfolio. We will return to these formulas later when we implement them in Python and then with pandas when we optimize portfolios.

Diversification

An investor can then reduce risk by holding combinations of instruments that are not positively correlated. If the asset pairs are perfectly uncorrelated (correlation of 0), then the portfolio's return variance is the sum over all the instruments of the square of the fraction held in the instrument multiplied by the instrument's return variance.

The efficient frontier

Using this model, the risk and expected returns of all possible combinations of risky assets is computed. This can then be plotted in the risk-return space, a two-dimensional space with the risk along the x axis and the expected return along the y axis. The collection of all such portfolios will define a region of the graph, with the left edge of what forms a hyperbola. This following hyperbola is often referred to as the Markowitz Bullet:



The upper portion of the hyperbola, represented with a solid line, represents the efficient frontier. All portfolios along the solid portion on the line can only increase in return with increased risk. However, also note that any portfolio on the efficient frontier also has a matching portfolio on the lower half of the bullet, which represents a portfolio with the same amount of risk but with less expected return. All things considered, an investor will want to take the portfolio with higher return over one with lower return and with the same risk. Hence, only portfolios on the portion of the hyperbola at higher returns than the minimum variance portfolio are considered on the efficient frontier.

Modeling a portfolio with pandas

A basic portfolio model consists of a specification of one or more investments and their quantities. A portfolio can be modeled in pandas using a `DataFrame` with one column representing the particular instrument (such as a stock symbol) and the other representing the quantity of the item held.

The following command will create a `DataFrame` representing a portfolio:

```
In [2]:
   def create_portfolio(tickers, weights=None):
       if (weights is None):
           shares = np.ones(len(tickers))/len(tickers)
       portfolio = pd.DataFrame({'Tickers': tickers,
                                 'Weights': weights},
                                index=tickers)
       return portfolio
```

Using this, we can create a portfolio of two instruments, `Stock A` and `Stock B`. The amount of shares for each is initialized to `1`. This would represent an equally weighted portfolio as the number of shares of each stock is the same:

```
In [3]:
   portfolio = create_portfolio(['Stock A', 'Stock B'],
                                [1, 1])
   portfolio


Out[3]:
            Shares  Tickers
   Stock A       1  Stock A
   Stock B       1  Stock B
```

We can then model mock returns for the last 5 years. The values used for returns are picked to demonstrate a point about creating an equally-weighted portfolio and to use negatively correlated instruments to create a representation of the diversification effect:

```
In [4]:
   returns = pd.DataFrame(
           {'Stock A': [0.1, 0.24, 0.05, -0.02, 0.2],
            'Stock B': [-0.15, -0.2, -0.01, 0.04, -0.15]})
```

```
   returns
```

```
Out[4]:
      Stock A  Stock B
   0     0.10    -0.15
   1     0.24    -0.20
   2     0.05    -0.01
   3    -0.02     0.04
   4     0.20    -0.15
```

Using the portfolio share values and the returns, the following function will compute the equally-weighted return for the underlying instruments:

```
In [5]:
   def calculate_weighted_portfolio_value(portfolio,
                                          returns,
                                          name='Value'):
       total_weights = portfolio.Weights.sum()
       weighted_returns = returns * (portfolio.Weights /
                                     total_weights)
       return pd.DataFrame({name: weighted_returns.sum(axis=1)})
```

We can now calculate the equally-weighted portfolio and concatenate it with our original `DataFrame` of returns:

```
In [6]:
   wr = calculate_weighted_portfolio_value(portfolio,
                                           returns,
                                           "Value")
   with_value = pd.concat([returns, wr], axis=1)
   with_value
```

```
Out[6]:
      Stock A  Stock B  Value
   0     0.10    -0.15 -0.025
   1     0.24    -0.20  0.020
   2     0.05    -0.01  0.020
   3    -0.02     0.04  0.010
   4     0.20    -0.15  0.025
```

We can examine the volatility of each of the individual instruments combined with the results of the weighted portfolio, as shown here:

```
In [7]:
   with_value.std()


Out[7]:
   Stock A    0.106677
   Stock B    0.103102
   Value      0.020310
   dtype: float64
```

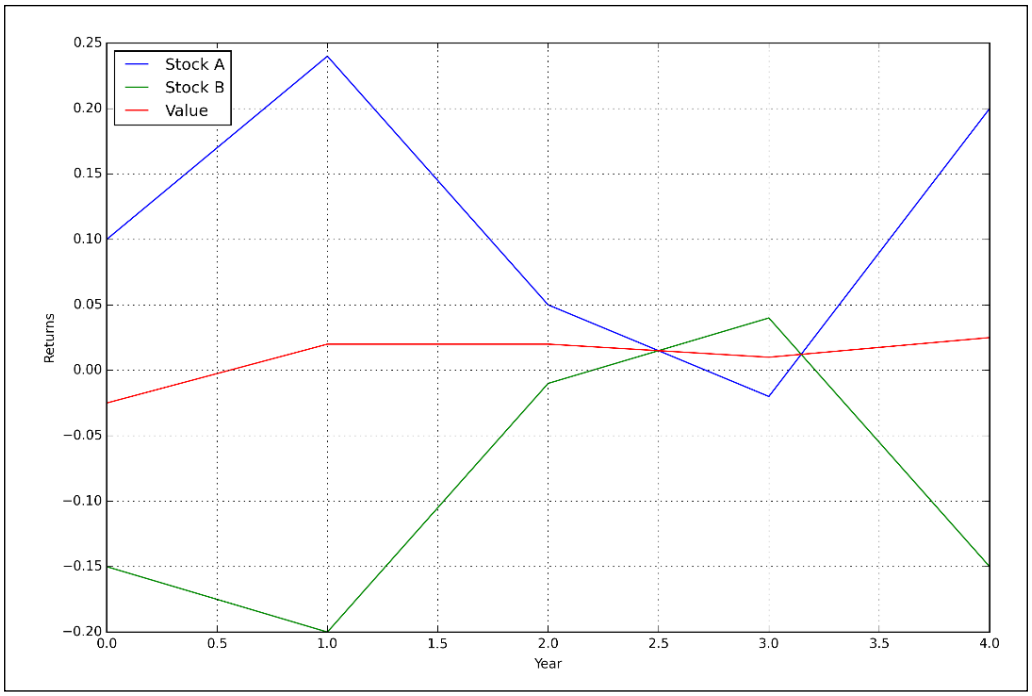
`Stock A` had a volatility of 11 percent and `Stock B` of 10 percent. The combined portfolio represented significantly lower volatility of 2 percent. This is because we picked two negatively correlated stocks with similar volatility and combining them has therefore reduced the overall risk.

We can visualize this using the following function:

```
In [8]:
   def plot_portfolio_returns(returns, title=None):
       returns.plot(figsize=(12,8))
       plt.xlabel('Year')
       plt.ylabel('Returns')
       if (title is not None): plt.title(title)
       plt.show()
```

Also examine the following graph:

```
In [9]:
   plot_portfolio_returns(with_value)
```



It becomes apparent from this graph that the overall portfolio had much less variability, and hence risk, than those of the individual instruments in the portfolio.

Just to check, we can also calculate the correlation of the original returns:

```
In [10]:
   returns.corr()


Out[10]:
              Stock A   Stock B
   Stock A  1.000000 -0.925572
   Stock B -0.925572  1.000000
```

The returns of our two stocks have a negative correlation of `-0.93`, which tells us that they can be used to offset each other's volatility.

This scenario used an equally-weighted portfolio of stocks that have a strong negative correlation and returns of similar magnitude. The real trick that we will examine in the upcoming sections will be to select an optimal portfolio from a set of stocks and to also determine the proper weighting for each stock to reach the optimized portfolio- that is, the efficient frontier.

Constructing an efficient portfolio

At the beginning of the chapter, we briefly covered the formulas to calculate the estimated return and variance of a portfolio. We will now dive into implementations of those calculations along with selecting portfolios that are on the efficient frontier.

To do this, we will need to cover the following concepts:

- Gathering of historical returns on the assets in the portfolio
- Formulation of portfolio risk based on historical returns
- Determining the Sharpe ratio for a portfolio
- Selecting optimal portfolios based upon Sharpe ratios

Gathering historical returns for a portfolio

In our examples, we will use data retrieved from Yahoo! Finance to create historical returns for the stocks in the portfolio. The calculations we will perform will utilize annualized returns. Yahoo! Finance data represents daily prices for the stocks, so we will need to convert those prices into annualized returns.

We can start this process using the following function, which will retrieve the adjusted closing prices for a list of stocks between the two dates and organize it in a convenient way for the processes we will undertake:

```
In [11]:
   def get_historical_closes(ticker, start_date, end_date):
       p = web.DataReader(ticker, "yahoo", start_date, end_date)
       d = p.to_frame()['Adj Close'].reset_index()
       d.rename(columns={'minor': 'Ticker',
                         'Adj Close': 'Close'}, inplace=True)
       pivoted = d.pivot(index='Date', columns='Ticker')
       pivoted.columns = pivoted.columns.droplevel(0)
       return pivoted
```

Our examples will utilize `AAPL`, `MSFT`, and `KO` stocks, from `2010-01-01` through `2014-12-31`. We can retrieve those daily prices as follows:

```
In [12]:
   closes = ef_get_historical_closes(['MSFT', 'AAPL', 'KO'],
                                     '2010-01-01', '2014-12-31')

In [13]:
   closes[:5]

Out[13]:
   Ticker           AAPL        KO      MSFT
   Date
   2010-01-04  28.83805  24.46602  26.94331
   2010-01-05  28.88790  24.17006  26.95201
   2010-01-06  28.42840  24.16148  26.78661
   2010-01-07  28.37585  24.10143  26.50804
   2010-01-08  28.56450  23.65535  26.69085
```

Using this data, the following function will calculate annualized returns for each of the stocks. We start with the following function, which converts daily prices into daily returns:

```
In [14]:
   def calc_daily_returns(closes):
       return np.log(closes/closes.shift(1))
```

Our daily returns are shown here:

```
In [15]:
   daily_returns = calc_daily_returns(closes)
   daily_returns[:5]

Out[15]:
   Ticker           AAPL        KO      MSFT
   Date
   2010-01-04        NaN       NaN       NaN
   2010-01-05   0.001727 -0.012171  0.000323
   2010-01-06  -0.016034 -0.000355 -0.006156
   2010-01-07  -0.001850 -0.002488 -0.010454
   2010-01-08   0.006626 -0.018682  0.006873
```

From the daily returns, we can calculate annualized returns using the following function:

```
In [16]:
   def calc_annual_returns(daily_returns):
       grouped = np.exp(daily_returns.groupby(
           lambda date: date.year).sum())-1
       return grouped
```

This gives us the following as the annual returns:

```
In [17]:
   annual_returns = calc_annual_returns(daily_returns)
   annual_returns
```

```
Out[17]:
   Ticker      AAPL        KO      MSFT
   2010    0.507219  0.189366 -0.079442
   2011    0.255580  0.094586 -0.045156
   2012    0.325669  0.065276  0.057989
   2013    0.080695  0.172330  0.442979
   2014    0.406225  0.052661  0.275646
```

Formulation of portfolio risks

Since we now have a return matrix, we can estimate its variance-covariance matrix, and by combining it with a vector of weights for each of the assets, we can calculate the overall portfolio variance (this flows into the Sharpe ratio calculation we will do next).

The formulation of the portfolio variance starts with the calculation of the mean of the returns for an individual stock:

$$\overline{R} = \frac{\sum_{i=1}^{n} R_i}{n}$$

Using this, we can then calculate the variance in the returns of a single stock:

$$\sigma^2 = \frac{\sum_{i=1}^{n}(R_i - \overline{R})^2}{n-1}$$

Here, R_i is the stock's return for period *i*, $\overline{R}$ is the mean of the returns, and n is the number of the observations.

The return volatility is simply the square root of the variance:

$$\sigma = \sqrt{\sigma^2}$$

A portfolio will consist of one or more stocks. The return matrix for those stocks consists of *n* stocks and *m* returns:

$$R = \begin{pmatrix} R_{1,1} & \cdots & R_{1,m} \\ \vdots & \ddots & \vdots \\ R_{n,1} & \cdots & R_{n.m} \end{pmatrix}$$

Using this return matrix, we can derive the formula for the expected return of stock *i*:

$$E(R_i) = \sum_{i=1}^{n} w_i R_{i,n}$$

Each stock will make up a certain percentage of the portfolio. We represent this mix of the stock in the portfolio using a vector of weights, *w*, which necessarily sums up to 1:

$$w = (w_1, w_1, w_1, \cdots, w_m)$$

We can apply this vector of weights to the assets in an n-stock portfolio, resulting in the following formula that gives us the weighted expected return of the portfolio:

$$E(R_{port}) = \sum_{i=1}^{n} w_i E(R_i)$$

The variance of an n-stock portfolio is formulated using the following formula:

$$\sigma_{port}^2 = \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j \, \sigma_i \sigma_j \rho_{ij}$$

Here ρ_{ij} is the correlation coefficient between returns on assets *i* and *j*, and $\rho_{ij} = 1$ for *i=j*.

Examining this formula more closely, the following equation can be seen:

$$\Sigma = \sigma_i \sigma_j \rho_{ij}$$

Sigma happens to be the covariance matrix calculated from the returns matrix.

Pulling this all together with the summations, we come to the following formula, which describes the variance of a weighted portfolio of n-stocks:

$$\sigma_{port}^2 = w * \Sigma * w'$$

Therefore, the variance of a portfolio is determined by multiplying the weights vector by the covariance matrix of the returns, and then multiplying that result by the transpose of the weights vector.

This can be very succinctly implemented in Python using NumPy arrays and matrices and the `np.cov()` function, which will calculate the covariance of the returns:

```
In [18]:
   def calc_portfolio_var(returns, weights=None):
       if (weights is None):
           weights = np.ones(returns.columns.size) / \
           returns.columns.size
       sigma = np.cov(returns.T,ddof=0)
       var = (weights * sigma * weights.T).sum()
       return var
```

Using this function, the variance of our portfolio (using equal weighting for each stock) is determined by the following command:

```
In [19]:
   calc_portfolio_var(annual_returns)


Out[19]:
   0.0028795357274894692
```

The Sharpe ratio

The Sharpe ratio is a measurement of the risk-adjusted performance of portfolios. It is calculated by subtracting the risk-free rate from the expected return of a portfolio and then by dividing that result by the standard deviation of the portfolio returns. It is described by the following equation:

$$Sharpe = \frac{E(R) - R_f}{\sigma_p}$$

The Sharpe ratio tells us whether a portfolio's returns are due to smart investment decisions or a result of excess risk. Although one portfolio or fund can reap higher returns than its peers, it is only a good investment if those higher returns do not come with too much additional risk. The greater a portfolio's Sharpe ratio, the better its risk-adjusted performance has been. A negative Sharpe ratio indicates that a less risky asset would perform better than the security being analyzed.

The following function calculates Sharp Ratio for a portfolio with specified returns, weights, and a risk-free rate:

```
In [20]:
   def sharpe_ratio(returns, weights = None, risk_free_rate = 0.015):
       n = returns.columns.size
       if weights is None: weights = np.ones(n)/n
       var = calc_portfolio_var(returns, weights)
       means = returns.mean()
       return (means.dot(weights) - risk_free_rate)/np.sqrt(var)
```

We can use this to evaluate the Sharpe ratio of our current portfolio with equal weights using the following statement:

```
In [21]:
   Sharpe_ratio(returns)


Out[21]:
   3.2010949029381952
```

Now that we can calculate the Sharpe ratio for a portfolio with a given set of weights, we need to be able to simulate the generation of different combinations of weights and select the weights where the Sharpe ratio is maximized. This will give us the efficient portfolio. This simulation of weights will be performed using SciPy's optimization capabilities.

Optimization and minimization

We now need to perform optimizations to find the efficient portfolio. Optimizations in Python can be performed using `scipy.optimize`. We will first demonstrate optimization using a basic example and then later, we will optimize portfolios based on Sharpe ratios.

Our basic example will be to minimize the following objective function:

$$y = 2 + x^2$$

Intuitively, we know that when `x` is `0`, `y` is minimized. We can use this to check the results of the minimization. The first step is to define the function we wish to minimize:

```
In [22]:
   def y_f(x): return 2+x**2
```

We can perform the optimization using SciPy's `fmin()` function. The value `1000` is passed as a seed value for `x`, and the function will iterate values of `x` to find the value of `x` where `y_f` is minimized:

```
In [23]:
   scopt.fmin(y_f, 1000)
   Optimization terminated successfully.
            Current function value: 2.000000
            Iterations: 27
```

```
            Function evaluations: 54

Out[23]:
   array([ 0.])
```

The `fmin()` function ran 27 iterations, called `y_f(x)` with 54 different values of `x`, and determined that the minimum result is `2.0`. The array that is returned contains the values for `x` at which `y_f(x) = 2`, which is a single value `x=0`.

Constructing an optimal portfolio

We are now able to create a function to use `fmin()` to determine the set of weights that maximize the Sharpe ratio for a given set of returns representing the stocks in our portfolio.

Since `fmin()` finds a minimum of the applied function, and the efficient portfolio exists at the maximized Sharpe ratio, we need to provide a function that, in essence, returns the negative of the Sharpe ratio, hence allowing `fmin()` to find a minimum:

```
In [24]:
   def negative_sharpe_ratio_n_minus_1_stock(weights,
                                             returns,
                                             risk_free_rate):
       """
       Given n-1 weights, return a negative sharpe ratio
       """
       weights2 = sp.append(weights, 1-np.sum(weights))
       return -sharpe_ratio(returns, weights2, risk_free_rate)
```

Our final function is given a `DataFrame` of returns, and a risk-free rate will run a minimization process on our negative `sharpe` function. The process is seeded with an array of equal weights, and `fmin()` will start from those values and try different combinations of weights until we find the minimized negative Sharpe ratio. The function then returns a tuple of the weights satisfying the minimization, along with the optimal Sharpe ratio:

```
In [25]:
   def optimize_portfolio(returns, risk_free_rate):
       w0 = np.ones(returns.columns.size-1,
                    dtype=float) * 1.0 / returns.columns.size
       w1 = scopt.fmin(negative_sharpe_ratio_n_minus_1_stock,
```

```
                        w0, args=(returns, risk_free_rate))
        final_w = sp.append(w1, 1 - np.sum(w1))
        final_sharpe = sharpe_ratio(returns, final_w, risk_free_rate)
        return (final_w, final_sharpe)
```

Using this function, we can now determine the most efficient portfolio:

```
In [26]:
   optimize_portfolio(annual_returns, 0.0003)

   Optimization terminated successfully.
            Current function value: -7.829864
            Iterations: 46
            Function evaluations: 89
Out[26]:
   (array([ 0.76353353,  0.2103234 ,  0.02614307]),
    7.8298640872716048)
```

We are told that our best portfolio would have 76.4 percent AAPL, 21.0 percent KO, and 2.6 percent MSFT, and that portfolio would have a Sharpe ratio of `7.8298640872716048`.

Visualizing the efficient frontier

Our optimization code generated the portfolio that is optimal for the specific risk-free rate of return. This is one type of? portfolio. To be able to plot all of the portfolios along the Markowitz bullet, we can change the optimization around a little bit.

The following function takes a weights vector, the returns, and a target return and calculates the variance of that portfolio with an extra penalty the further the mean is from the target return, so as to help push portfolios with weights further from the mean considering they are on the frontier:

```
In [27]:
   def objfun(W, R, target_ret):
        stock_mean = np.mean(R,axis=0)
        port_mean = np.dot(W,stock_mean)
        cov=np.cov(R.T)
        port_var = np.dot(np.dot(W,cov),W.T)
        penalty = 2000*abs(port_mean-target_ret)
        return np.sqrt(port_var) + penalty
```

We now create a function that will run through a set of desired return values, ranging from the lowest returning stock to the highest returning stock. These create the bounds for the possible rates of returns.

Each of these desired returns is passed to an optimizer, which will create a weights vector that satisfies the minimization of the Sharpe ratio of a portfolio that matches that specific level of risk.

For each optimal set of weights, the program will return the mean and standard deviation (and weights) that represent the curve of the efficient frontier:

```
In [28]:
   def calc_efficient_frontier(returns):
       result_means = []
       result_stds = []
       result_weights = []

       means = returns.mean()
       min_mean, max_mean = means.min(), means.max()

       nstocks = returns.columns.size

       for r in np.linspace(min_mean, max_mean, 100):
           weights = np.ones(nstocks)/nstocks
           bounds = [(0,1) for i in np.arange(nstocks)]
           constraints = ({'type': 'eq',
                           'fun': lambda W: np.sum(W) - 1})
           results = scopt.minimize(objfun, weights, (returns, r),
                                    method='SLSQP',
                                    constraints = constraints,
                                    bounds = bounds)
           if not results.success: # handle error
               raise Exception(result.message)
           result_means.append(np.round(r,4)) # 4 decimal places
           std_=np.round(np.std(np.sum(returns*results.x,axis=1)),6)
           result_stds.append(std_)
```

```
            result_weights.append(np.round(results.x, 5))
        return {'Means': result_means,
                'Stds': result_stds,
                'Weights': result_weights}
```

Given our previous set of stocks (AAPL, MSFT, and KO), the following command will calculate all of the pairs of standard deviation and mean returns that fall on the efficient frontier:

```
In [29]:
   frontier_data = calc_efficient_frontier(annual_returns)
```

The `frontier_data` function is a dictionary that contains an array for each of the calculated standard deviations, mean returns, and weights that resulted from the optimization.

We can examine the results by inspecting the values of several of the items in the dictionary. The following command examines the first five standard deviations, means, and entries in an array of optimal weights:

```
In [30]:
   frontier_data['Stds'][:5]


Out[30]:
   [0.055842999999999997, 0.053446, 0.052564, 0.051706000000000002,
0.050871]


In [31]:
   frontier_data['Stds'][:5]


Out[31]:
   [0.1148, 0.1169, 0.11890000000000001, 0.12089999999999999, 0.1229]


In [32]:
   frontier_data['Weights'][:5]


Out[32]:
   [array([-0.,  1.,  0.]),
    array([ 0.00512,  0.9308 ,  0.06408]),
    array([ 0.01497,  0.9177 ,  0.06733]),
```

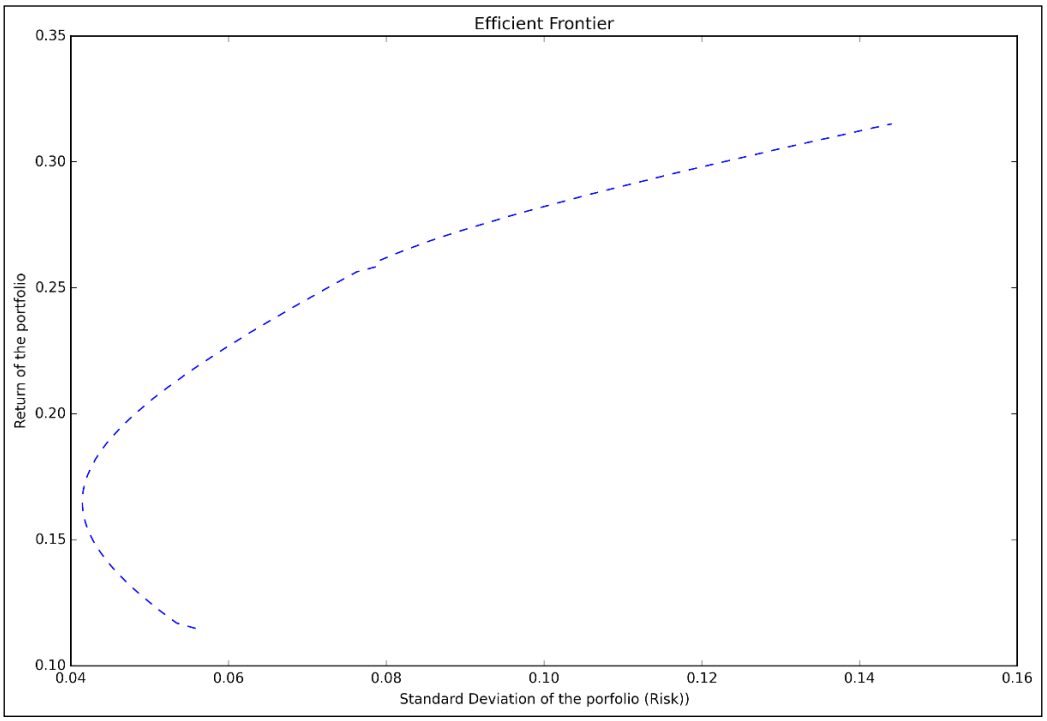
```
    array([ 0.02469,  0.90303,  0.07228]),
    array([ 0.03458,  0.89049,  0.07493])]
```

We can use the following function to visualize this efficient frontier:

```
In [33]:
   def plot_efficient_frontier(ef_data):
       plt.figure(figsize=(12,8))
       plt.title('Efficient Frontier')
       plt.xlabel('Standard Deviation of the porfolio (Risk))')
       plt.ylabel('Return of the portfolio')
       plt.plot(ef_data['Stds'], ef_data['Means'], '--');
```

The following shows how our efficient frontier look:

```
In [34]:
plot_efficient_frontier(frontier_data)
```



Value at Risk

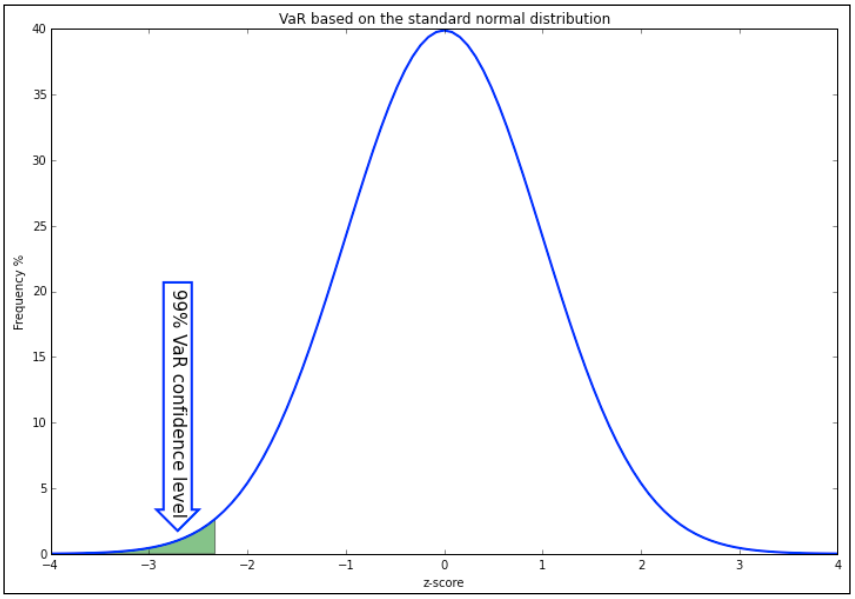
Value at Risk (**VaR**) is a statistical technique used to measure the level of financial risk within an investment portfolio, over a specific timeframe. It measures in three variables—the amount of potential loss, the probability of the loss, and the timeframe.

As an example, a portfolio may have a 1-month 5 percent VaR of $1 million. This means that there is a 5 percent probability that the portfolio will fall in value by more than $1 million over a 1-month period. Likewise, it also means that a $1 million loss should be expected once every 20 months.

The most common means of measuring VaR is by calculating the volatility. There are three common means of calculating the volatility: using historical data, variance-covariance, and the Monte Carlo simulation. We will examine the variance-covariance method here, as there is a straightforward formulation for the VaR once you have historical returns.

VaR assumes that returns are normally distributed. The returns for a stock or portfolio over the desired period of time can then be created, and then we can examine the amount of distribution of returns that fits within a z-score for the desired confidence interval.

This concept can be visualized using a normal distribution curve. Common percentages for VaR calculations typically are 1 percent and 5 percent. The following example demonstrates calculating a 99 percent confidence interval, which is where we would find the area in the normal distribution where the z-score less than -2.33:



To apply this to the returns of a stock, the formula for the VaR for a given period is shown here:

$$VaR_{period} = position * (\mu_{period} - z * \sigma_{period})$$

The position is the current market value of the stock, μ_{period} is the mean of the returns for the specific period, and σ_{period} is the volatility (standard deviation of the returns); *z* is the z-score representing the specific confidence interval—z=2.33 for a 99 percent confidence interval, and z=1.64 for a 95 percent confidence interval.

To demonstrate this, we will examine the 1-year VaR for AAPL using returns from the entirety of 2014. To calculate this, we can reuse the functions that we created for calculating an efficient frontier.

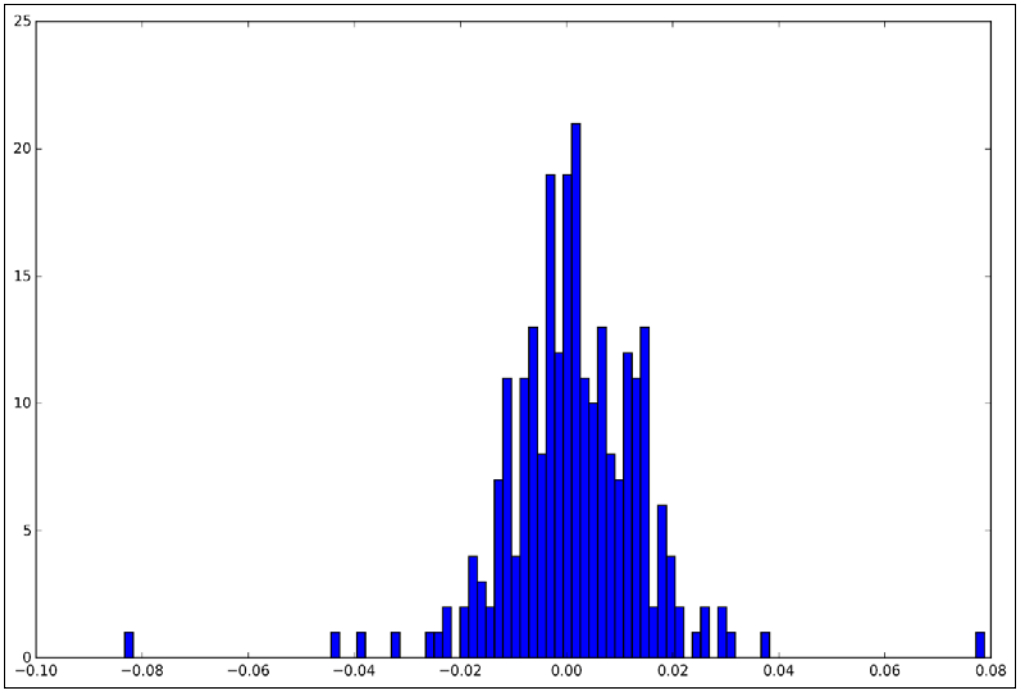
We start the analysis by loading the daily prices for 2014 for AAPL and calculating the daily returns:

```
In [35]:
   aapl_closes = get_historical_closes(['AAPL'],
                                       datetime(2014, 1, 1),
                                       datetime(2014, 12, 31))
   aapl_closes[:5]

Out[35]:
   Ticker          AAPL
   Date
   2014-01-02  77.08570
   2014-01-03  75.39245
   2014-01-06  75.80357
   2014-01-07  75.26144
   2014-01-08  75.73806

In [36]:
   returns = calc_daily_returns(aapl_closes)
   returns[:5]

Out[36]:
   Ticker          AAPL
   Date
```

```
2014-01-02       NaN
2014-01-03 -0.022211
2014-01-06  0.005438
2014-01-07 -0.007177
2014-01-08  0.006313
```

We can plot these returns in a histogram to check that they appear to be normally distributed:

```
In [37]:
   plt.figure(figsize=(12,8))
   plt.hist(returns.values[1:], bins=100);
```



We can explicitly code `z` for the confidence interval, but we can also get the value of `z` for any percentage using `norm.ppf()` from `scipy.stats`:

```
In [38]:
   z = spstats.norm.ppf(0.95)
   z

Out[38]:
   1.6448536269514722
```

We will model our position as though we have 1,000 shares of AAPL on `2014-12-31`:

```
In [39]:
   position = 1000 * aapl_closes.ix['2014-12-31'].AAPL
   position

Out[39]:
   109950.0
```

The VaR is calculated as follows:

```
In [40]:
   VaR = position * (z * returns.AAPL.std())
   VaR

Out[40]:
   2467.5489391697483
```

This states that our holdings in AAPL at $109,950 have a VaR of $2,647. Therefore, our maximum loss in the next year is $2,647 with a confidence of 95 percent.

Summary

In this chapter, we examined how to combine combinations of assets into a portfolio and how to model those portfolios using pandas objects. Using a portfolio, we examined how to calculate the overall risk involved in the portfolio, and learned how we can use negatively correlated assets to be able to minimize risk.

We then expanded upon this concept of risk minimization, using concepts from modern portfolio theory to be able to determine whether our portfolio represents the best mix of assets to yield the highest return at a specific level of risk. This included calculating the efficiency of a portfolio using the Sharpe ratio, and then using optimization tools from SciPy to determine the optimum allocation of instruments in the portfolio.

In closing, we went on a significant tour of using pandas to perform various tasks related to finance. We touched on a number of the features built directly into pandas to be able to model and manipulate financial data, particularly using time-series data and the capabilities pandas provides to help solve complicated date- and time-related problems. We also dived into other domain-specific analyses, such as historical stock analysis, analyzing social data to make trading decisions, algorithmic trading, options pricing, and portfolio management, thus offering a practical set of examples for you to learn these concepts.

Index

E

F

G

H

I

J

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

Thank you for buying
Mastering pandas for Finance

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

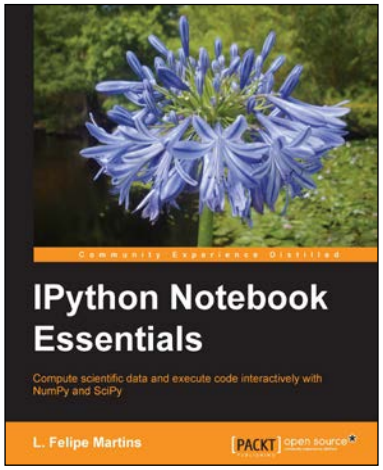
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

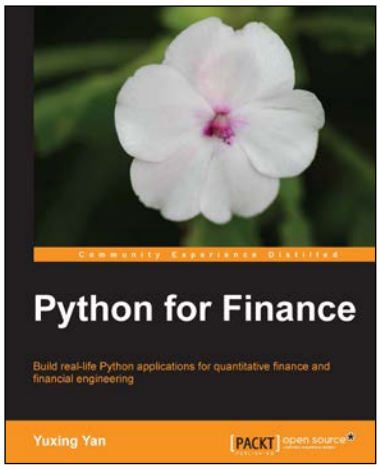


IPython Notebook Essentials

ISBN: 978-1-78398-834-1 Paperback: 190 pages

Compute scientific data and execute code interactively with NumPy and SciPy

1. Perform Computational Analysis interactively.
2. Create quality displays using matplotlib and Python Data Analysis.
3. Step-by-step guide with a rich set of examples and a thorough presentation of the IPython Notebook.



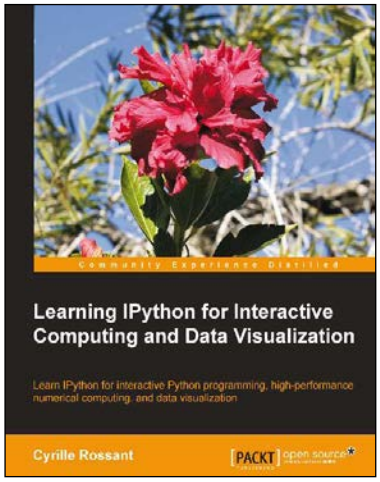
Python for Finance

ISBN: 978-1-78328-437-5 Paperback: 408 pages

Build real-life Python applications for quantitative finance and financial engineering

1. Estimate market risk, form various portfolios, and estimate their variance-covariance matrixes using real-world data.
2. Explains many financial concepts and trading strategies with the help of graphs.
3. A step-by-step tutorial with many Python programs that will help you learn how to apply Python to finance.

[PACKT] open source
community experience distilled
PUBLISHING

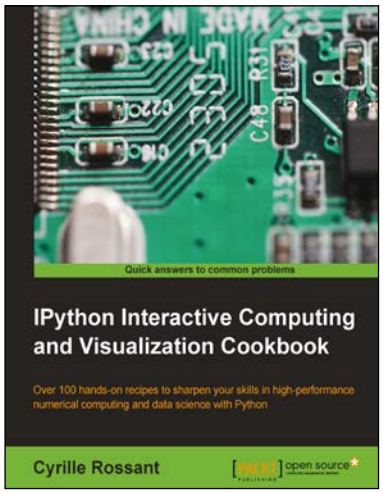


Learning IPython for Interactive Computing and Data Visualization

ISBN: 978-1-78216-993-2 Paperback: 138 pages

Learn IPython for interactive Python programming, high-performance numerical computing, and data visualization

1. A practical step-by-step tutorial, which will help you to replace the Python console with the powerful IPython command-line interface.
2. Use the IPython Notebook to modernize the way you interact with Python.
3. Perform highly efficient computations with NumPy and pandas.



IPython Interactive Computing and Visualization Cookbook

ISBN: 978-1-78328-481-8 Paperback: 512 pages

Over 100 hands-on recipes to sharpen your skills in high-performance numerical computing and data science with Python

1. Leverage the new features of the IPython Notebook for interactive web-based big data analysis and visualization.
2. Become an expert in high-performance computing and visualization for data analysis and scientific modeling.
3. A comprehensive coverage of scientific computing through many hands-on, example-driven recipes with detailed, step-by-step explanations.