



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Python for Finance

Build real-life Python applications for quantitative finance and financial engineering

Yuxing Yan

[PACKT] open source*
PUBLISHING community experience distilled

Python for Finance

Build real-life Python applications for quantitative finance and financial engineering

Yuxing Yan



Python for Finance

Copyright © 2014 Packt Publishing

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

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

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

First published: April 2014

Production Reference: 1180414

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

ISBN 978-1-78328-437-5

www.packtpub.com

Cover Image by Aniket Sawant (aniket_sawant_photography@hotmail.com)

Credits

Author

Yuxing Yan

Project Coordinator

Swati Kumari

Reviewers

Mourad MOURAFIQ

Loucas Papayiannis

Jiri Pik

Proofreaders

Simran Bhogal

Maria Gould

Ameesha Green

Paul Hindle

Joanna McMahon

Commissioning Editor

Usha Iyer

Acquisition Editors

Pramila Balan

Llewellyn Rozario

Indexers

Mehreen Deshkmukh

Monica Ajmera Mehta

Rekha Nair

Tejal Soni

Priya Subramani

Content Development Editor

Ruchita Bhansali

Technical Editors

Shubhangi Dhamgaye

Krishnaveni Haridas

Arwa Manasawala

Ankita Thakur

Graphics

Abhinash Sahu

Production Coordinator

Aditi Gajjar Patel

Copy Editors

Roshni Banerjee

Sarang Chari

Adithi Shetty

Cover Work

Aditi Gajjar Patel

About the Author

Yuxing Yan graduated from McGill university with a PhD in finance. He has taught various finance courses, such as Financial Modeling, Options and Futures, Portfolio Theory, Quantitative Financial Analysis, Corporate Finance, and Introduction to Financial Databases to undergraduate and graduate students at seven universities: two in Canada, one in Singapore, and four in the USA.

Dr. Yan has actively done research with several publications in Journal of Accounting and Finance, Journal of Banking and Finance, Journal of Empirical Finance, Real Estate Review, Pacific Basin Finance Journal, Applied Financial Economics, and Annals of Operations Research. For example, his latest publication, co-authored with Shaojun Zhang, will appear in the Journal of Banking and Finance in 2014. His research areas include investment, market microstructure, and open source finance.

He is proficient at several computer languages such as SAS, R, MATLAB, C, and Python. From 2003 to 2010, he worked as a technical director at Wharton Research Data Services (WRDS), where he debugged several hundred computer programs related to research for WRDS users. After that, he returned to teaching in 2010 and introduced R into several quantitative courses at two universities. Based on lecture notes, he has the first draft of an unpublished manuscript titled *Financial Modeling using R*.

In addition, he is an expert on financial data. While teaching at NTU in Singapore, he offered a course called *Introduction to Financial Databases* to doctoral students. While working at WRDS, he answered numerous questions related to financial databases and helped update CRSP, Compustat, IBES, and TAQ (NYSE high-frequency database). In 2007, Dr. Yan and S.W. Zhu (his co-author) published a book titled *Financial Databases, Shiwu Zhu and Yuxing Yan, Tsinghua University Press*. Currently, he spends considerable time and effort on public financial data. If you have any queries, you can always contact him at yany@canisius.edu.

Acknowledgments

I would like to thank Ben Amoako-Adu, Brian Smith (who taught me the first two finance courses and offered unstinting support for many years after my graduation), George Athanassakos (one of his assignments "forced" me to learn C), Jin-Chun Duan, Wei-Hung Mao, Jerome Detemple, Bill Sealey, Chris Jacobs, Mo Chaudhury (my former professors at McGill), and Laurence Kryzanowski. (His wonderful teaching inspired me to concentrate on empirical finance and he edited my doctoral thesis word by word even though he was not my supervisor!)

There is no doubt that my experience at Wharton has shaped my thinking and enhanced my skill sets. I thank Chris Schull and Michael Boldin for offering me the job; Mark Keintz, Dong Xu, Steven Crispi, and Dave Robinson, my former colleagues, who helped me greatly during my first two years at Wharton; and Eric Zhu, Paul Ratnaraj, Premal Vora, Shuguang Zhang, Michelle Duan, Nicholle Mcniece, Russ Ney, Robin Nussbaum-Gold, and Mireia Gine for all their help.

In addition, I'd like to thank Shaobo Ji, Tong Yu, Shaoming Huang, Xing Zhang, Changwen Miao, Karyl Leggio, Lisa Fairchild, K. G. Viswanathan, Na Wang, Mark Lennon, and Qiyu (Jason) Zhang for helping me in many ways. I also want to thank Shaojun Zhang and Qian Sun, my former colleagues and co-authors on several papers, for their valuable input and discussions.

Creating a good book involves many talented publishing professionals and external reviewers in addition to the author(s). I would like to acknowledge the excellent efforts and input from the staff of my publisher, Packt Publishing, especially Llewellyn F. Rozario, Swati Kumari, Arwa Manasawala, Ruchita Bhansali, Apeksha Chitnis, and Pramila Balan as well as the external reviewers, Martin Olveyra, Mourad MOURAFIQ, and Loucas Parayiannis, for their valuable advice, suggestions, and criticism.

Finally, and most importantly, I thank my wife, Xiaoning Jin, for her strong support, my daughter, Jing Yan, and son, James Yan, for their understanding and love they have showered on me over the years.

About the Reviewers

Jiri Pik is a finance and business intelligence consultant working with major investment banks, hedge funds, and other financial players. He has architected and delivered breakthrough trading, portfolio and risk management systems, and decision-support systems across industries.

His consulting firm, WIXESYS, provides their clients with certified expertise, judgment, and execution at the speed of light. WIXESYS' power tools include revolutionary Excel and Outlook add-ons available at <http://spearian.com>.

Loucas Papayiannis was born and raised in Cyprus, where he graduated from the English School in Nicosia. After completing his mandatory military service at the Cyprus National Guard, Loucas left for the University of California, Berkeley, where he obtained a BSc in Electrical Engineering and Computer Science. While at Berkeley, he had the opportunity to work for the Bosch Research Center in Palo Alto, where he developed a strong interest in computer-human interface.

In an unexpected turn of events, an opportunity to work for Bloomberg LP in London came up, after he completed his studies. Despite the fact that financial software was a sharp change of direction from where Loucas was heading at the time, he moved to London and seized the opportunity. He quickly grew to enjoy this new field and consequently enrolled in an MSc program in Financial Mathematics at King's College, London while still working full time, completing this degree in 2011.

In 2010, he started at Goldman Sachs, and in August 2012, he joined Barclays Capital, where he is currently employed. His work is focused on developing an FX Options application, and he mainly works with C++. However, he has worked with a variety of languages and technologies through the years. He is a Linux and Python enthusiast and spends his free time experimenting and developing applications with them.

Mourad MOURAFIQ is a software engineer and data scientist. After successfully completing his studies in Applied Mathematics, he worked at an investment bank as a quantitative modeler in the structured products market, specializing in ABS, CDO, and CDS. Then, he worked as a quantitative analyst for the largest French bank.

After a couple of years in the financial world, he discovered a passion for machine learning and computational mathematics and decided to join a start-up that specializes in software mining and artificial intelligence.

I would like to thank my mentors who took me under their wings during my initial days on the trading floor.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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

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



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across 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 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 nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Introduction and Installation of Python	9
Introduction to Python	10
Installing Python	12
Different versions of Python	12
Ways to launch Python	13
Launching Python with GUI	13
Launching Python from the Python command line	14
Launching Python from our own DOS window	15
Quitting Python	16
Error messages	16
Python language is case sensitive	17
Initializing the variable	17
Finding the help window	18
Finding manuals and tutorials	19
Finding the version of Python	21
Summary	21
Exercises	22
Chapter 2: Using Python as an Ordinary Calculator	23
Assigning values to variables	24
Displaying the value of a variable	24
Error messages	24
Can't call a variable without assignment	25
Choosing meaningful names	25
Using dir() to find variables and functions	26
Deleting or unassigning a variable	27

Basic math operations – addition, subtraction, multiplication, and division	28
The power function, floor, and remainder	28
A true power function	30
Choosing appropriate precision	31
Finding out more information about a specific built-in function	32
Listing all built-in functions	32
Importing the math module	33
The pi, e, log, and exponential functions	34
"import math" versus "from math import *"	34
A few frequently used functions	36
The print() function	36
The type() function	36
Last expression _ (underscore)	36
Combining two strings	37
The upper() function	37
The tuple data type	39
Summary	40
Exercises	40
Chapter 3: Using Python as a Financial Calculator	43
Writing a Python function without saving it	44
Default input values for a function	45
Indentation is critical in Python	45
Checking the existence of our functions	46
Defining functions from our Python editor	47
Activating our function using the import function	48
Debugging a program from a Python editor	48
Two ways to call our pv_f() function	49
Generating our own module	50
Types of comments	51
The first type of comment	51
The second type of comment	52
Finding information about our pv_f() function	52
The if() function	53
Annuity estimation	54
Converting the interest rates	55
Continuously compounded interest rate	57
A data type – list	58
Net present value and the NPV rule	58
Defining the payback period and the payback period rule	60
Defining IRR and the IRR rule	61

Showing certain files in a specific subdirectory	62
Using Python as a financial calculator	63
Adding our project directory to the path	64
Summary	66
Exercises	67
Chapter 4: 13 Lines of Python to Price a Call Option	71
Writing a program – the empty shell method	73
Writing a program – the comment-all-out method	75
Using and debugging other programs	76
Summary	76
Exercises	77
Chapter 5: Introduction to Modules	79
What is a module?	80
Importing a module	80
Adopting a short name for an imported module	81
Showing all functions in an imported module	82
Comparing "import math" and "from math import *"	82
Deleting an imported module	83
Importing only a few needed functions	84
Finding out all built-in modules	85
Finding out all the available modules	86
Finding the location of an imported module	87
More information about modules	88
Finding a specific uninstalled module	90
Module dependency	90
Summary	92
Exercises	93
Chapter 6: Introduction to NumPy and SciPy	95
Installation of NumPy and SciPy	96
Launching Python from Anaconda	96
Examples of using NumPy	97
Examples of using SciPy	98
Showing all functions in NumPy and SciPy	102
More information about a specific function	103
Understanding the list data type	103
Working with arrays of ones, zeros, and the identity matrix	104
Performing array manipulations	105
Performing array operations with +, -, *, /	105
Performing plus and minus operations	105

Performing a matrix multiplication operation	105
Performing an item-by-item multiplication operation	107
The x.sum() dot function	107
Looping through an array	108
Using the help function related to modules	108
A list of subpackages for SciPy	109
Cumulative standard normal distribution	109
Logic relationships related to an array	110
Statistic submodule (stats) from SciPy	111
Interpolation in SciPy	112
Solving linear equations using SciPy	113
Generating random numbers with a seed	114
Finding a function from an imported module	116
Understanding optimization	116
Linear regression and Capital Assets Pricing Model (CAPM)	117
Retrieving data from an external text file	118
The loadtxt() and getfromtxt() functions	118
Installing NumPy independently	119
Understanding the data types	119
Summary	120
Exercises	120
Chapter 7: Visual Finance via Matplotlib	123
Installing matplotlib via ActivePython	124
Alternative installation via Anaconda	125
Understanding how to use matplotlib	125
Understanding simple and compounded interest rates	129
Adding texts to our graph	131
Working with DuPont identity	133
Understanding the Net Present Value (NPV) profile	135
Using colors effectively	137
Using different shapes	139
Graphical representation of the portfolio diversification effect	140
Number of stocks and portfolio risk	142
Retrieving historical price data from Yahoo! Finance	144
Histogram showing return distribution	145
Comparing stock and market returns	148
Understanding the time value of money	150
Candlesticks representation of IBM's daily price	151
Graphical representation of two-year price movement	153
IBM's intra-day graphical representations	154

Presenting both closing price and trading volume	156
Adding mathematical formulae to our graph	157
Adding simple images to our graphs	158
Saving our figure to a file	159
Performance comparisons among stocks	160
Comparing return versus volatility for several stocks	161
Finding manuals, examples, and videos	163
Installing the matplotlib module independently	163
Summary	163
Exercises	164
Chapter 8: Statistical Analysis of Time Series	167
Installing Pandas and statsmodels	168
Launching Python using the Anaconda command prompt	169
Launching Python using the DOS window	169
Launching Python using Spyder	170
Using Pandas and statsmodels	171
Using Pandas	171
Examples from statsmodels	173
Open data sources	174
Retrieving data to our programs	176
Inputting data from the clipboard	176
Retrieving historical price data from Yahoo! Finance	177
Inputting data from a text file	178
Inputting data from an Excel file	179
Inputting data from a CSV file	180
Retrieving data from a web page	180
Inputting data from a MATLAB dataset	181
Several important functionalities	182
Using <code>pd.Series()</code> to generate one-dimensional time series	182
Using date variables	183
Using the DataFrame	183
Return estimation	185
Converting daily returns to monthly returns	187
Converting daily returns to annual returns	190
Merging datasets by date	191
Forming an n-stock portfolio	192
T-test and F-test	193
Tests of equal means and equal variances	194
Testing the January effect	195

Many useful applications	196
52-week high and low trading strategy	196
Roll's model to estimate spread (1984)	197
Amihud's model for illiquidity (2002)	198
Pastor and Stambaugh (2003) liquidity measure	199
Fama-French three-factor model	204
Fama-MacBeth regression	206
Estimating rolling beta	207
Understanding VaR	210
Constructing an efficient frontier	211
Estimating a variance-covariance matrix	212
Optimization – minimization	214
Constructing an optimal portfolio	215
Constructing an efficient frontier with n stocks	217
Understanding the interpolation technique	220
Outputting data to external files	221
Outputting data to a text file	221
Saving our data to a binary file	222
Reading data from a binary file	222
Python for high-frequency data	222
Spread estimated based on high-frequency data	227
More on using Spyder	228
A useful dataset	230
Summary	232
Exercise	232
Chapter 9: The Black-Scholes-Merton Option Model	237
Payoff and profit/loss functions for the call and put options	238
European versus American options	242
Cash flows, types of options, a right, and an obligation	243
Normal distribution, standard normal distribution, and cumulative standard normal distribution	243
The Black-Scholes-Merton option model on non-dividend paying stocks	247
The p4f module for options	248
European options with known dividends	250
Various trading strategies	251
Covered call – long a stock and short a call	252
Straddle – buy a call and a put with the same exercise prices	253
A calendar spread	254

Butterfly with calls	256
Relationship between input values and option values	257
Greek letters for options	258
The put-call parity and its graphical representation	259
Binomial tree (the CRR method) and its graphical representation	261
The binomial tree method for European options	268
The binomial tree method for American options	268
Hedging strategies	269
Summary	270
Exercises	271
Chapter 10: Python Loops and Implied Volatility	275
Definition of an implied volatility	276
Understanding a for loop	277
Estimating the implied volatility by using a for loop	278
Implied volatility function based on a European call	279
Implied volatility based on a put option model	280
The enumerate() function	281
Estimation of IRR via a for loop	282
Estimation of multiple IRRs	283
Understanding a while loop	284
Using keyboard commands to stop an infinitive loop	285
Estimating implied volatility by using a while loop	286
Nested (multiple) for loops	288
Estimating implied volatility by using an American call	288
Measuring efficiency by time spent in finishing a program	289
The mechanism of a binary search	290
Sequential versus random access	292
Looping through an array/DataFrame	293
Assignment through a for loop	294
Looping through a dictionary	294
Retrieving option data from CBOE	295
Retrieving option data from Yahoo! Finance	297
Different expiring dates from Yahoo! Finance	299
Retrieving the current price from Yahoo! Finance	300
The put-call ratio	300
The put-call ratio for a short period with a trend	302
Summary	303
Exercises	304

Chapter 11: Monte Carlo Simulation and Options	307
Generating random numbers from a standard normal distribution	308
Drawing random samples from a normal (Gaussian) distribution	309
Generating random numbers with a seed	309
Generating n random numbers from a normal distribution	310
Histogram for a normal distribution	310
Graphical presentation of a lognormal distribution	311
Generating random numbers from a uniform distribution	312
Using simulation to estimate the pi value	313
Generating random numbers from a Poisson distribution	315
Selecting m stocks randomly from n given stocks	315
Bootstrapping with/without replacements	317
Distribution of annual returns	319
Simulation of stock price movements	320
Graphical presentation of stock prices at options' maturity dates	322
Finding an efficient portfolio and frontier	324
Finding an efficient frontier based on two stocks	324
Impact of different correlations	326
Constructing an efficient frontier with n stocks	329
Geometric versus arithmetic mean	332
Long-term return forecasting	333
Pricing a call using simulation	334
Exotic options	335
Using the Monte Carlo simulation to price average options	335
Pricing barrier options using the Monte Carlo simulation	337
Barrier in-and-out parity	339
Graphical presentation of an up-and-out and up-and-in parity	340
Pricing lookback options with floating strikes	342
Using the Sobol sequence to improve the efficiency	344
Summary	344
Exercises	345
Chapter 12: Volatility Measures and GARCH	347
Conventional volatility measure – standard deviation	348
Tests of normality	349
Estimating fat tails	350
Lower partial standard deviation	352
Test of equivalency of volatility over two periods	354
Test of heteroskedasticity, Breusch, and Pagan (1979)	355
Retrieving option data from Yahoo! Finance	358
Volatility smile and skewness	360
Graphical presentation of volatility clustering	362

The ARCH model	363
Simulating an ARCH (1) process	364
The GARCH (Generalized ARCH) model	365
Simulating a GARCH process	366
Simulating a GARCH (p,q) process using modified garchSim()	367
GJR_GARCH by Glosten, Jagannathan, and Runkle (1993)	369
Summary	373
Exercises	373
Index	375

Preface

It is our firm belief that an ambitious student major in finance should learn at least one computer language. The basic reason is that we have entered the Big Data era. In finance, we have a huge amount of data, and most of it is publically available free of charge. To use such rich sources of data efficiently, we need a tool. Among many potential candidates, Python is one of the best choices.

Why Python?

There are various reasons that Python should be used. Firstly, Python is free in terms of license. Python is available for all major operating systems, such as Windows, Linux/Unix, OS/2, Mac, and Amiga, among others. Being free has many benefits. When students graduate, they could apply what they have learned wherever they go. This is true for the financial community as well. In contrast, this is not true for SAS and MATLAB. Secondly, Python is powerful, flexible, and easy to learn. It is capable of solving almost all our financial and economic estimations. Thirdly, we could apply Python to Big Data. Dasgupta (2013) argues that R and Python are two of the most popular open source programming languages for data analysis. Fourthly, there are many useful modules in Python. Each model is developed for a special purpose. In this book, we focus on NumPy, SciPy, Matplotlib, Statsmodels, and Pandas modules.

A programming book written by a finance professor

There is no doubt that the majority of programming books are written by professors from computer science. It seems odd that a finance professor writes a programming book. It is understandable that the focus would be quite different. If an instructor from computer science were writing this book, naturally the focus would be Python, whereas the true focus should be finance. This should be obvious from the title of the book *Python for Finance*. This book intends to change the fact that many programming books serving the finance community have too much for the language itself and too little for finance.

Small programs oriented

Based on the author's teaching experience at seven schools, McGill and Wilfrid Laurier University (in Canada), NTU (in Singapore), and Loyola University, Maryland, UMUC, Hofstra University, and Canisius College (in the United States), and his eight-year consulting experience at Wharton School, he knows that many finance students like small programs that solve one specific task. Most programming books offer just a few complete and complex programs. The number of programs is far too less than enough. There are two side effects for such an approach. First, finance students are drowned in programming details, get intimidated, and eventually lose interest in learning a computer language. Second, they don't learn how to apply what they just learned, such as running a capital asset pricing model (CAPM) to estimate IBM's beta from 1990 to 2013. This book offers about 300 complete Python programs around many finance topics.

Using real-world data

Another shortcoming of the majority of books for programming is that they use hypothetical data. In this book, we use real-world data for various financial topics. For example, instead of showing how to run CAPM to estimate the beta (market risk), I show you how to estimate IBM, Apple, or Walmart's betas. Rather than just presenting formulae that shows you how to estimate a portfolio's return and risk, the Python programs are given to download real-world data, form various portfolios, and then estimate their returns and risk including Value at Risk (VaR). When I was a doctoral student, I learned the basic concept of volatility smiles. However, until writing this book, I had a chance to download real-world data to draw IBM's volatility smile.

What this book covers

Chapter 1, Introduction and Installation of Python, offers a short introduction, and explains how to install Python and covers other related issues such as how to launch and quit Python.

Chapter 2, Using Python as an Ordinary Calculator, presents some basic concepts and several frequently used Python built-in functions, such as basic assignment, precision, addition, subtraction, division, power function, and square root function.

Chapter 3, Using Python as a Financial Calculator, teaches us how to write simple functions, such as functions to estimate the present value of one future cash flow, the future value of one present value, the present value of annuity, the future value of annuity, the present value of perpetuity, the price of a bond, and internal rate of return (IRR).

Chapter 4, 13 Lines of Python to Price a Call Option, shows how to write a call option without detailed knowledge about options and Python.

Chapter 5, Introduction to Modules, discusses modules, such as finding all available or installed modules, and how to install a new module.

Chapter 6, Introduction to NumPy and SciPy, introduces the two most important modules, called NumPy and SciPy, which are used intensively for scientific and financial computation.

Chapter 7, Visual Finance via Matplotlib, shows you how to use the matplotlib module to vividly explain many financial concepts by using graphs, pictures, color, and size.

Chapter 8, Statistical Analysis of Time Series, discusses many concepts and issues associated with statistics in detail. Topics include how to download historical prices from Yahoo! Finance; estimate returns, total risk, market risk, correlation among stocks, correlation among different countries' markets; form various types of portfolios; and construct an efficient portfolio.

Chapter 9, The Black-Scholes-Merton Option Model, discusses the Black-Scholes-Merton option model in detail. In particular, it will cover the payoff and profit/loss functions and their graphic presentations of call and put options, various trading strategies and their visual presentations, normal distribution, Greeks, and put-call parity.

Chapter 10, Python Loops and Implied Volatility, introduces different types of loops. Then it demonstrates how to estimate the implied volatility based on both European and American options.

Chapter 11, Monte Carlo Simulation and Options, discusses how to use Monte Carlo simulation to price European, American, average, lookback, and barrier options.

Chapter 12, Volatility Measures and GARCH, focuses on two issues: volatility measures and ARCH/GARCH.

What could you achieve after reading this book?

Here, we use several concrete examples to show what a reader could achieve after going through this book carefully.

First, after reading the first two chapters, a reader/student should be able to use Python to calculate the present value, future value, present value of annuity, IRR (internal rate of return), and many other financial formulae. In other words, we could use Python as a free ordinary calculator to solve many finance problems. Second, after the first three chapters, a reader/student or a finance instructor could build a free financial calculator, that is, combine about a few dozen small Python programs into a big Python program. This big program behaves just like any other module written by others. Third, readers learn how to write Python programs to download and process financial data from various public data sources, such as Yahoo! Finance, Google Finance, Federal Reserve Data Library, and Prof. French Data Library.

Fourth, readers would understand basic concepts associated with modules, which are packages written by experts, other users, or us, for specific purposes. Fifth, after understanding the module of Matplotlib, a reader could do various graphs. For instance, readers could use graphs to demonstrate payoff/profit outcomes based on various trading strategies by combining the underlying stocks and options. Sixth, readers would be able to download IBM's daily price, and S&P 500 index price, data from Yahoo! Finance and estimate its market risk (beta) by applying CAPM. They could also form a portfolio with different securities, such as risk-free assets, bonds, and stocks. Then, they can optimize their portfolios by applying Markowitz's mean-variance model. In addition, readers will know how to estimate the VaR of their portfolios.

Seventh, a reader should be able to price European and American options by applying both the Black-Scholes-Merton option model for European options only, and the Monte Carlo Simulation, for both European and American options. Last but not least, a reader learns several ways to measure volatility. In particular, they will learn how to use Autoregressive Conditional Heteroskedasticity (ARCH) and Generalized Autoregressive Conditional Heteroskedasticity (GARCH) models.

Who this book is for

If you are a graduate student major in finance, especially studying computational finance, financial modeling, financial engineering, and business analytics, this book will benefit you. If you are a professional, you could learn Python and use it in many financial projects. If you are an individual investor, you could benefit from reading this book as well.

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, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Depending on your computer, choose the appropriate package, for example, Python 3.3.2 Windows x86 MSI Installer (Windows binary -- does not include source)."

If we have a program, we will see the following codes:

```
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd
import statsmodels.api as sm
ticker='IBM'
begdate=(2008,10,1)
enddate=(2013,11,30)
p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
```

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

```
>>>from matplotlib.pyplot import *
>>>plot([1,2,3,10])
>>>xlabel("x- axis")
>>>ylabel("my numbers")
>>>title("my figure")
>>>show()
```

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: "Click on **Start** and then on **All Programs**."

Two ways to use the book

Generally speaking, there are two ways to learn this book: read the book and learn Python by yourself, or learn Python in a classroom setting. For a beginner, going slow is a better strategy, such as spending two weeks per chapter except *Chapter 8, Statistical Analysis of Time Series*, which needs at least three weeks. Professionals with basic programming experience of another computer language could go through the first few chapters relatively quickly and move to more advanced topics (chapters). They should focus on option theory, implied volatility and measures of volatility, and GARCH models. One feature of this book is that most chapters after *Chapter 3, Using Python as a Financial Calculator*, are loosely connected. Because of this, after learning the first three chapters in addition to *Chapter 5, Introduction to Modules*, readers could jump to the chapters they are interested in.

On the other hand, the book is ideal to be used as a textbook for *Financial Modeling using Python* or simply *Python for finance* courses to master degree students in the areas of quantitative finance, computational finance, or financial engineering. The amount of content of the book and expected effort needed is suitable for one semester. The students could be senior undergraduate students with a reduced depth. To teach undergraduate students, the last chapter should be dropped.

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 via the subject of your message.

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.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/43750S_Images.pdf.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

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

Introduction and Installation of Python

In this chapter, first we offer a short introduction on why we adopt Python as our computational tool and what the advantages are of using Python. Then, we discuss how to install Python and other related issues, such as how to start and quit Python, whether Python is case sensitive, and a few simple examples.

In particular, we will cover the following topics:

- Introduction to Python
- Installing Python
- Which version of Python should we use and what is the version of our installed Python?
- Ways to launch and quit Python
- Error messages
- Python is case sensitive
- Initializing the variables
- Finding help, manuals, and tutorials
- Finding the Python versions

Introduction to Python

Our society entered the information era many years ago. Actually, we are drowning in a sea of information, such as too many e-mails to read or too many web pages we could possibly explore. With the Internet, we could find a huge amount of information about almost everything such as important events or how to learn Python. We could find information for a specific firm by searching online. For instance, if we want to collect financial information associated with **International Business Machines (IBM)**, we could use Yahoo! Finance, Google Finance, **Securities and Exchange Commission (SEC)** filings, and the company's web pages. Since we are confronted with a lot of publicly available information, investors, professionals, and researchers need a tool to process such a huge amount of information. In addition, our society would move towards a more open and transparent society. In finance, a new concept of open source finance has merged recently. Dane and Masters (2009) suggest three components for open source finance: open software, open data, and open codes. For the first component of open software, Python is one of the best choices. An equally popular open source software is R. In the next section, we summarize the advantages of learning and applying Python to finance.

Firstly, Python is free in terms of license. Being free has many benefits. Let's perform a simple experiment here. Let's assume readers know nothing about Python and they have no knowledge about option theory. How long do you think it would take them to run a Python program to price a Black-Scholes call option? Less than 2 hours? Here is what they could do; they could download and install Python after reading the *Installing Python* section of this chapter, and it would take less than 10 minutes. Spend another 10 minutes to launch and quit Python and also try a few examples. Then, read the first page of *Chapter 4, 13 Lines of Python to Price a Call Option*, which contains the code for the famous Black-Scholes call option model. In total, the program has 13 lines. The reader could spend the next 40 minutes typing, correcting typos, and retyping those 13 lines. With less than 2 hours, they should be able to run the program to price a call option. The cost of adopting a new computer language includes many aspects such as annual license cost, maintenance costs, available packages, and support.

Another example is related to an SEC proposal. In 2010, the SEC proposed that all financial institutions are to accompany their new **Asset-Backed Security (ABS)** with a computer program showing the contractual cash flows of the securities (www.sec.gov/rules/proposed/2010/33-9117.pdf). The proposed computer language is Python. Obviously, any investor can access Python because it is free.

For bond analytics or credit risks, Roger Ehrenberg (2007) argues for an open source approach. Whether or not ratings should be required for institutional investors to buy certain securities is not the issue; the essential point is getting better transparency and analysis of instruments constituting the investable universe. Just imagine what the impact would be if many financial institutions adopt the open source initiative by storing their own debt ratings models into the public domain and allowing others to contribute to its development! To contribute to such an open source approach, Python (or R, free software as well) would be ideal in terms of computational tools.

Secondly, Python is powerful, flexible, and easy to learn. It is capable of solving almost all our financial and economic estimations. Python is available for all major operating systems such as Windows, Linux/Unix, OS/2, Mac, and Amiga, among others.

Thirdly, Python is suitable for Big Data. Dasgupta (2013) argues that R and Python are two of the most popular open source programming languages for data analysis. Compared with R, Python is generally a better overall language, especially when you consider its blend of functional programming with object orientation. Combined with `Scipy/Numpy`, `Matplotlib`, and `Statsmodel`, it provides a powerful tool. In this book, we will discuss a module called `Pandas` when we deal with financial data.

Fourthly, there are many useful modules for Python, such as toolboxes in MATLAB and packages in R. Each model is developed for a special purpose. Later in the book, we will touch base with about a dozen modules. However, we will pay special attention to five of the most useful modules in finance: `NumPy`, `SciPy`, `Matplotlib`, `Statsmodels`, and `Pandas`. The first two modules are associated with mathematical estimations, formulae, matrices and their manipulation, data formats, and data manipulations. `Matplotlib` is for visual presentations such as graphs. In *Chapter 8, Introduction to the Black-Scholes Option Model*, we use this module intensively to explain visually different payoff functions and profit/loss functions for various trading strategies. The `Statsmodels` module deals with econometrics such as T-test, F-test, and GARCH models. Again, the `Pandas` module is used for financial data analysis.

We should mention some disadvantages of Python as well. The most important shortcoming is the lack of support because it is free. Some experts argue that the Python community needs to grow and should include more statisticians and mathematicians.

Installing Python

To install Python, perform the following two steps:

1. Go to <http://www.python.org/download>.
2. Depending on your computer, choose the appropriate package, for example, Python 3.3.2 Windows x86 MSI Installer (Windows binary -- does not include source).

At this stage, a new user could install the latest Python version. In other words, they could simply ignore the next section related to the version and go directly to the *How to launch Python* section.

Generally speaking, the following are the three ways to launch Python:

- From Python IDLE (GUI)
- From the Python command line
- From your command-line window

The three ways will be introduced in the *How to launch Python?*, *Launch Python from Python command line*, and *The third way to launch Python* sections.

Different versions of Python

One of the most frequently asked questions related to Python's installation is which version we should download. At this stage, any latest version would be fine, that is, the version does not matter. There are three reasons behind this statement:

- The contents of the first four chapters are compatible with any version
- Removing and downloading Python is trivial
- Different versions could coexist

Later in the book, we will explain the module dependency which is associated with a Python version. A module is a collection of many Python programs, written by one or a group of experts, to serve a special purpose. For example, we will discuss a module called `statsmodels`, which is related to statistical and econometric models, linear regression and the like. Generally speaking, we have built-in modules, standard modules, third-party modules, and modules built by ourselves. We will spend several chapters on this important topic.

In this book, we will mention about two dozen modules. In particular, we will discuss in detail the NumPy, SciPy, Matplotlib, Pandas, and Statsmodels modules. The NumPy, Matplotlib, and Statsmodels modules depend on Python 2.7 or above. All these packages have different versions for Python 2.x (2.5-2.6 and above, depending on the case).

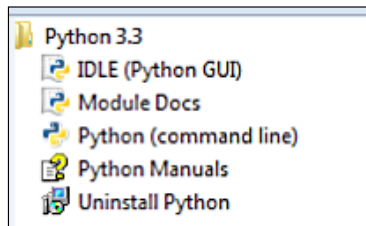
Ways to launch Python

There are three ways to launch Python and they are explained in the following sections.

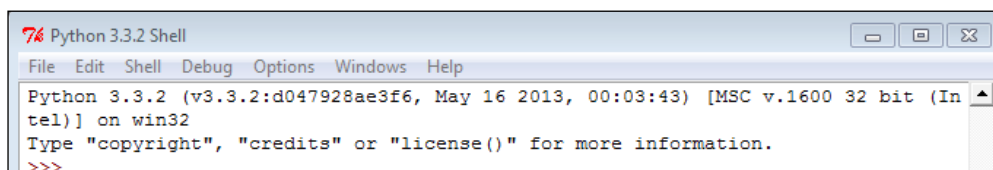
Launching Python with GUI

To launch Python, perform the following steps:

1. Click on **Start** and then on **All Programs**.
2. Find **Python 3.3**.
3. Click on **IDLE (Python GUI)** as shown in the following screenshot:



4. After Python starts, the following window appears:



Assume that an estimate of \$100 is expected to be received in one year with an annual discount rate of 10 percent. The present value of one future cash flow is as follows:

$$PV = \frac{FV}{(1 + R)^n} \quad (1)$$

In this equation, PV is the present value, FV is the future value, R is the discount rate, and n is the number of periods. According to the preceding formula, we could manually type those values to get the present value of this one future cash flow. Assume that we would receive \$100 in one year. If the annual discount rate is 10 percent, what is the present value of this \$100? For this, let's take a look at the following lines of code:

```
>>>100/(1+0.1)
90.9090909090909
>>>
```



The triple larger than signs (`>>>`) is the Python prompt.

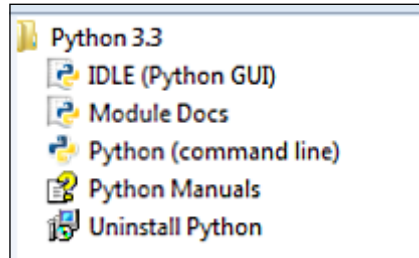
It is a good idea to create a Python icon on your desktops for your convenience. In addition to the preceding method, there are other methods to launch Python; see the next two sections: *Launching Python from the Python command line* and *Launching Python from our own DOS window*.

Launching Python from the Python command line

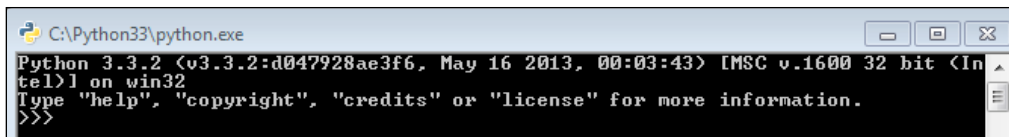
A new user could skip this section and go to the *Quitting Python* section because learning how to launch Python with GUI is more than enough. There are two reasons for this. The first is because we know how to launch Python by using Python IDLE or by clicking on the Python icon on our desktops, and the second reason is that we could save and run our Python programs easily using Python IDLE.

To launch Python from the Python command line, we have to perform the following steps:

1. Click on **Start** and then on **All Programs**.
2. Find **Python 3.3**.
3. Click on **Python (command line)** as shown in the following screenshot:



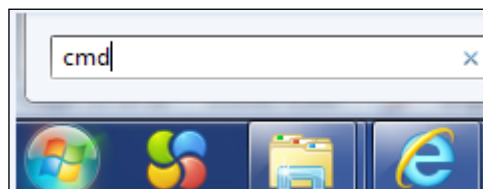
4. After we click on **Python (command line)**, we will see the following window:



Launching Python from our own DOS window

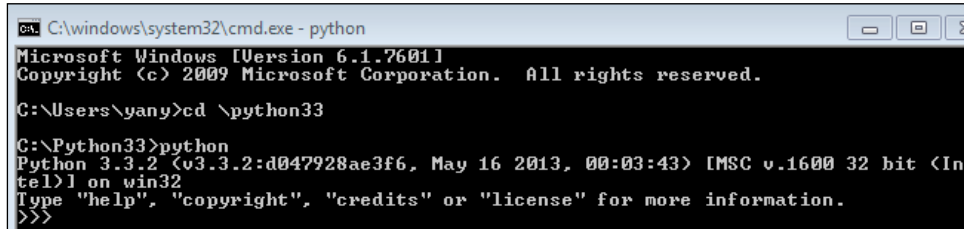
We could generate our own DOS window, and then launch Python from there. In addition, we could navigate to the subdirectory, which contains our Python programs. In order to this, perform the following steps:

1. Open a Windows command line by clicking on **Start** and then enter `cmd` in the run window as shown in the following screenshot:



2. Type `cd c:\python33` to move to the appropriate directory.

3. Type `python` to run the software as shown in the following screenshot:



```
C:\windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\yany>cd \python33
C:\Python33>python
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If we want to launch Python anywhere else, we have to include the path of our Python directory. Assume that after installation we have `python33` in `C:.` Replace step 2 with the following DOS command:

```
set path=%path%;C:\python33
```

Quitting Python

Usually, we have several ways to quit Python, which are as follows:

- The first way to quit Python is to use *Ctrl + D*
- The second way to quit is *Ctrl + Q*
- The third way to quit is to click on **File** and then on **Exit**
- The fourth way is to click on **X** at the top-right corner of the window (that is, close the window)

Later in the book, we will explain how to embed certain codes to quit Python when a currently running program is finished.

Error messages

For the previous example, if we enter `100/(1+0.1)^2` instead of `100/(1+0.1)`, we will see the following error message, which tells us that `^` is not supported:

```
>>>100/(1+0.1)^2
Traceback (most recent call last):
File "<psyhell#1>", line 1, in <module>
100/(1+0.1)^2
TypeError: unsupported operand type(s) for ^: 'float' and 'int'
>>>
```



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.

At this stage, a new user needs to pay attention to the last sentence of the error message. Obviously, the last line tells us that `^` is not supported. Again, for a power function, we should use double multiplications, `**`, instead of a karat, `^`. In *Chapter 2, Using Python as an Ordinary Calculator*, we will show that a true power function, `pow()`, is available.

Python language is case sensitive

Case sensitive means that `x` is different from `X`. The variable of `John` is different from the variable of `john`. If we assume a value for `x` (lowercase `x`) and then call `x` (uppercase `X`), we will see the following error message:

```
>>>x=2
>>>X
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    X
NameError: name 'X' is not defined
>>>
```

In the preceding example, `x` is not assigned any value. Thus, when we call it by typing `x`, we will receive an error message. Note that the last line mentions `NameError` instead of `TypeError`. In Python, we use `name` for variables.

Initializing the variable

From the previous example, we know that after we assign a value to `x`, we can use `x`, which means that `x` is now defined in the sense of other computer languages such as FORTRAN and C/C++. The opposite is also true, that we could not use `x` if it is not assigned a value in Python. In other words, when we assign a value to `x`, we have to define it first. Compared to languages such as C++ or FORTRAN, we don't have to define `x` as an integer before we assign 10 to it.

Another advantage is that we could change the data type of a variable easily. For the FOTRAN language, if we have defined `x` as an integer, we cannot assign a string to it. Since there is only assignment in Python, we could assign any value to a variable. For example, we could assign 10 to `x`. It is legal to assign a string, such as Hello World, to `x` in the next minute. However, we should not be confused with the data type conversion, such as converting an integer to a string or vice versa. Conversion between different data types will be discussed in the later chapters.

Finding the help window

After we launch Python, typing `help()` would initiate the help window (as shown in the following lines of code). The prompt of the help window is `help>`. To quit the help window, we simply press the *Enter* key once or type `quit`. After we quit the help window, the Python prompt of `>>>` would reappear. Now, we launch the help window as shown in the following lines of code:

```
>>>help()
Welcome to Python 3.3! This is the interactive help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.3/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics". Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

After typing keywords, we will have the following information:

```
>>>help> keywords
Here is a list of the Python keywords. Enter any keyword to get more
help.

False      def        if         raise
None       del        import     return
True       elif       in         try
and        else       is         while
as         except     lambda     with
assert     finally   nonlocal   yield
```

```
break      for      not
class      from     or
continue   global   pass
```

```
help>
```

On the other hand, after typing `topics`, we will see what is shown in the following screenshot:

```
help> topics

Here is a list of available topics.  Enter any topic name to get more help.

ASSERTION      DELETION      LITERALS      SEQUENCES
ASSIGNMENT     DICTIONARIES  LOOPING       SHIFTING
ATTRIBUTEMETHODS  DICTIIONARYLITERALS  MAPPINGMETHODS  SLICINGS
ATTRIBUTES     DYNAMICFEATURES  MAPPINGS       SPECIALATTRIBUTES
AUGMENTEDASSIGNMENT  ELLIPSIS        METHODS        SPECIALIDENTIFIERS
BASICMETHODS    EXCEPTIONS      MODULES        SPECIALMETHODS
BINARY         EXECUTION       NAMESPACES     STRINGMETHODS
BITWISE        EXPRESSIONS     NONE           STRINGS
BOOLEAN        FILES           NUMBERMETHODS  SUBSCRIPTS
CALLABLEMETHODS  FLOAT          NUMBERS        TRACEBACKS
CALLS          FORMATTING      OBJECTS        TRUTHVALUE
CLASSES        FRAMEOBJECTS    OPERATORS      TUPLELITERALS
CODEOBJECTS     FRAMES          PACKAGES       TUPLES
COMPARISON      FUNCTIONS       POWER          TYPEOBJECTS
COMPLEX         IDENTIFIERS     PRECEDENCE     TYPES
CONDITIONAL     IMPORTING       PRIVATENAMES   UNARY
CONTEXTMANAGERS  INTEGER        RETURNING      UNICODE
CONVERSIONS     LISTLITERALS   SCOPING
DEBUGGING       LISTS          SEQUENCEMETHODS

help>
```

At the moment, a new user doesn't need to understand those topics. Just remember that we have a command to show us all the topics we could use.

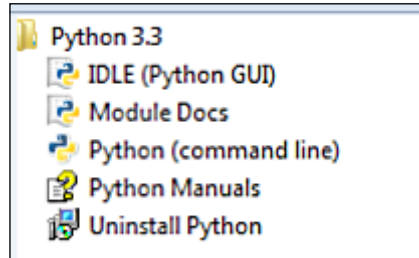
Finding manuals and tutorials

There are many ways to find Python manuals and other related materials online. We just mentioned two ways: from your computer and from the Python home. These two ways are explained in details as follows:

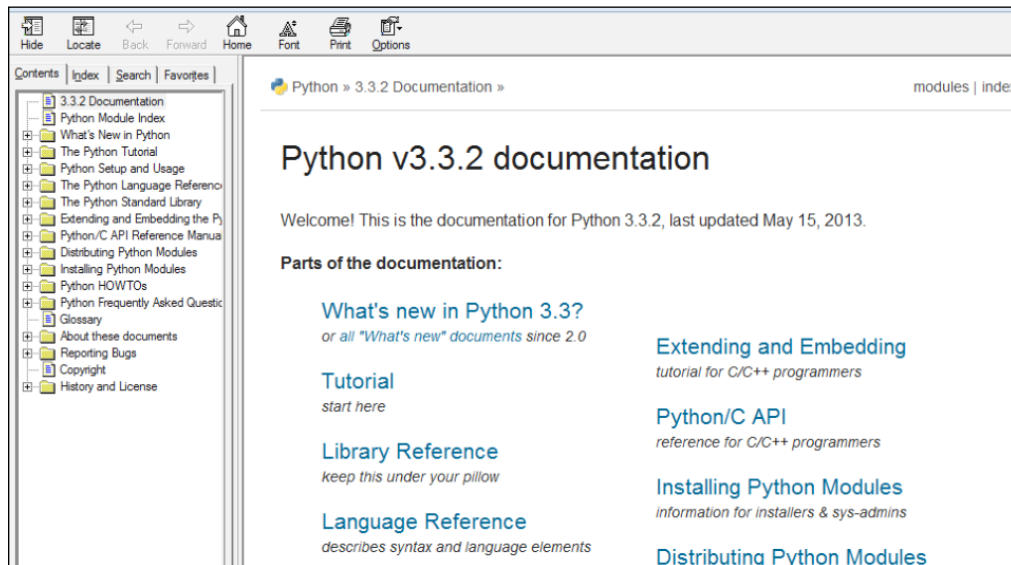
To implement the first method (to have it manually installed on your computer), we need to perform the following steps:

1. Click on **Start** and then on **All Programs**.
2. Find **Python 3.3**.

- Click on **Python Manuals** as shown in the following screenshot:



- After we click on **Python Manuals**, we will see the following window:



From the Python home, the following documents can be downloaded:

- Python 3.2 documents (3.2.5, last updated on May 15, 2013) at <http://docs.python.org/3.2/download.html>
- Python 3.3 documents (3.3.2, last updated on August 04, 2013) at <http://docs.python.org/3.3/download.html>
- Python 2.7 document(2.7.5, last updated on September 20, 2013) at <http://docs.python.org/2.7/download.html>

For new Python learners, the following are the web pages where they could find many tutorial materials related to Python learning:

- Online tutorials:
 - <http://docs.python.org/3/tutorial/>
 - <http://docs.python.org/2/tutorial/>
- PDF version (424 pages):
 - http://www.tutorialspoint.com/python/python_pdf_version.htm
 - <http://anh.cs.luc.edu/python/hands-on/3.1/Hands-onPythonTutorial.pdf>

Finding the version of Python

When Python is launched, the first line will show our current version. Another way is to issue the following two lines of Python code after we launch Python:

```
>>>import sys
>>>sys.version
'3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit
(Intel)]'
>>>
```

The first line of command imports a module called `sys`. A module is a collection of many Python programs serving a special purpose. Understanding a module is critical in learning Python. We will discuss this in more detail in *Chapter 5, Introduction to Modules*; *Chapter 6, Introduction to NumPy and SciPy*; *Chapter 7, Visual Finance via Matplotlib*; and *Chapter 8, Statistical Analysis of Time Series*.

Summary

In this chapter, we learned how to install Python and other related issues, such as how to launch and quit Python, whether Python is case sensitive, and a few simple examples. Since it's a simple and straightforward explanation, any reader who is new to Python could easily download and install Python in a few minutes. After that, they could try a few given examples. We also offered a brief introduction as to why we adopt Python as our computational tool, and what the advantages and disadvantages are of using Python.

In the next chapter, you will learn some basic concepts and several frequently used Python built-in functions. We will demonstrate how to use Python as an ordinary calculator to solve many finance-related problems. For example, we could estimate the present value of one future cash flow, the future value of one cash flow today, the present value of a perpetuity, or the present value of a growing perpetuity. In addition, we will discuss the `dir()`, `type()`, `floor()`, `round()`, and `help()` functions.

Exercises

1. Use a few sentences to describe the Python software.
2. What are the advantages and disadvantages of using Python as our computational tool?
3. Where can we download and install Python?
4. Is Python case sensitive? What is the basic rule to define various variables (names)?
5. Can we use a variable without defining it first?
6. Is it possible that we use a variable before we assign a value to it?
7. Is the version of Python important at this stage? Is the version of Python important later in the book?
8. In how many ways can we launch Python?
9. Where can we find videos on how to install Python?
10. What is the URL for Python's homepage?
11. Estimate the area of a circle if the diameter is 10 using Python.
12. How do you assign a value to a new variable?
13. How can you find some sample examples related to Python?
14. How do you launch Python's `help` function?
15. Where is the location of Python on your PC (Mac)? How do we find the path?
16. What is the difference between defining a variable and assigning a value to it?

2

Using Python as an Ordinary Calculator

In this chapter, we will learn some basic concepts and several frequently used built-in functions of Python, such as basic assignment, precision, addition, subtraction, division, power function, and square root function. In short, we demonstrate how to use Python as an ordinary calculator to solve many finance-related problems.

In this chapter, we will cover the following topics:

- Assigning values to variables
- Displaying the value of a variable
- Exploring error messages
- Understanding why we can't call a variable without assignment
- Choosing meaningful variable names
- Using `dir()` to find variables and functions
- Deleting or unsigning a variable
- Learning basic math operations—addition, subtraction, multiplication, and division
- Learning about the power function, floor, and remainder
- Choosing appropriate precision
- Finding out more information about a specific built-in function
- Importing the `math` module
- The `pi`, `e`, `log`, and exponential functions

- Distinguishing between `import math` and `from math import *`
- Understanding frequently used functions—`print()`, `type()`, last expression `_`, `upper()`, and combining two strings
- Learning about the `tuple` data type

Assigning values to variables

To assign a value to a variable is simple because unlike many other languages such as C++ or FORTRAN, in Python, we don't need to define a variable before we assign a value to it.

```
>>>pv=22
>>>pv+2
24
```

We could assign the same value to different variables simultaneously. In the following example, we assign 100 to the three variables `x`, `y`, and `z` at once:

```
>>>x=y=z=100
```

Displaying the value of a variable

To find out the value of a variable, just type its name as shown in the following code:

```
>>>pv=100
>>>pv
100
>>>R=0.1
>>>R
0.1
```

Error messages

Assuming that we issue the `sqrt(3)` command to estimate the square root of three, we would get the following error message:

```
>>>sqrt(3)
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    sqrt(3)
NameError: name 'sqrt' is not defined
```

The last line of the error message tells us that the `sqrt()` function is not defined. Later in the chapter, we learn that the `sqrt()` function is included in a module called `math` and that we have to load (import) the module before we can call the functions contained in it. A module is a package that contains a set of functions around a specific subject.

Can't call a variable without assignment

Assuming that we never assign a value to the `abcde` variable, after typing `abcde`, we would get the following error message:

```
>>>abcde
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    abcde
NameError: name 'abcde' is not defined
>>>
```

The last line tells us that this variable is not defined or assigned. In a sense, we could view our value assignment as being equivalent to doing two things: assigning a value to a variable and defining it at the same time.

Choosing meaningful names

A perpetuity describes the situations where equivalent periodic cash flows happen in the future and last forever. For example, we receive \$5 at the end of each year forever. A real-world example is the UK government bond, called *consol*, that pays fixed coupons. To estimate the present value of a perpetuity, we use the following formula if the first cash flow occurs at the end of the first period:

$$PV(\text{perpetuity}) = \frac{C}{R} \quad (1)$$

Here, PV is the present value, C is a perpetual periodic cash flow that happens at a fixed interval, and R is the periodic discount rate. Here C and R should be consistent. For example, if C is annual (monthly) cash flow, then R must be an annual (monthly) discount rate. This is true for other frequencies too. Assume that a constant annual cash flow is \$10, with the first cash flow at the end of the first year, and that the annual discount rate is 10 percent. Compare the following two ways to name the C and R variables:

```
>>>x=10          # bad way for variable names
>>>y=0.1
```

```
>>>z=x/y
>>>Z
100
>>>C=10          # good way for assignments
>>>R=0.1
>>>pv=C/R
>>>pv
100
```

Using C for our future periodic cash flow is better than x , and using R for the discount rate is better than y since both C and R are exactly the same as the variables used in equation (1), while x and y bear no specific meanings. A growing perpetuity is when the future cash flow grows at a constant growth rate, g . Its related present value is given in the following formula:

$$PV(\text{perpetuity}) = \frac{C}{R - g} \quad (2)$$

In this formula, C is the first cash flow one period from today, R is the periodic discount rate, and g is the growth rate. Obviously, the growth rate g should be less than the discount rate R . Here is a real-world example: we purchase a perpetuity bond with an annual payment C and an annual discount rate R . When we estimate its true value today, we have to consider future inflation rates. If the future annual inflation is **CPI (consumer price index)**, then the growth rate will be the negative CPI.

Using `dir()` to find variables and functions

After assigning values to a few variables, we could use the `dir()` function to show their existence. In the following example, variables `n`, `pv`, and `r` are shown among other names. At the moment, just ignore the first five objects in the following code, which start and end with two underscores:

```
>>>pv=100
>>>r=0.1
>>>n=5
>>>dir()
['_builtins_', '__doc__', '__loader__', '__name__', '__package__', 'n',
'pv', 'r']
```

Deleting or unsigning a variable

Sometimes, when we write our programs, it might be a good idea to delete those variables that we no longer need. In this case, we could use the `del()` function to remove or unsign a variable. In the following example, we assign a value to `rate`, show its value, delete it, and type the variable name trying to retrieve its value again:

```
>>>rate=0.075
>>>rate
0.075
```

The value `0.075` seen in the previous code is an output, because the variable called `rate` was assigned a value. The following code is used retrieve the value of the deleted variable:

```
>>>del rate
>>>rate
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    Rate
NameError: name 'rate' is not defined [End of codes]
```

This output tells us that the `rate` variable is not defined (refer to the last sentence of the previous output). To remove/delete/unsign several variables at once, we use a comma to separate those variables as shown in the following code:

```
>>>pv=100
>>>r=0.85
>>>dir()
['__builtins__', '__doc__', '__name__', '__package__', 'pv', 'r']
>>>del pv, r
>>>dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

Basic math operations – addition, subtraction, multiplication, and division

For basic math operations in Python, we use the conventional mathematical operators `+`, `-`, `*`, and `/`. These operators represent plus, minus, multiplication, and division operations respectively. All these operators are embedded in the following line of code:

```
>>>3.09+2.1*5.2-3/0.56
8.652857142857144
```

Although we use integer division less frequently in finance, a user might type the division sign twice (`//`) accidentally to get a weird result. The integer division is done with double slash `//`, which would return an integer value that is the largest integer than the final output. The result of 7 divided by 3 is 2.33, and 2 will be the largest integer smaller than 2.33. This example is shown in the following code:

```
>>>7/3
2.3333333333333335
```

For Python 2.x versions, `7/3` could be 2 instead of 2.333. Thus, we have to be careful. In order to avoid an integer division, we could use `7/2` or `7/2.`, that is, at least one of them is a real (float) number:

```
>>>7//3
2
```

Here, n/m is equivalent to an integer function of `int(n/m)` as shown in the following code:

```
>>>x=7/3
>>>x
2.3333333333333335
>>>int(x)
2
```

The power function, floor, and remainder

For our $FV = PV(1+R)^n$, we use a power function. The floor function would give the largest integer smaller than the current value. The remainder is the value that remains after an integer division. Given a positive discount rate, the present value of a future cash flow is always smaller than its corresponding future value.

The following formula specifies the relationship between a present value and its future value:

$$PV = \frac{FV}{(1+R)^n} \quad (3)$$

In this formula, PV is the present value, FV is the future value, R is the cost of capital (discount rate) per period, and n is the number of periods. Assume that we would receive \$100 payment in two years and that the annual discount rate is 10 percent. What is the equivalent value today that we are willing to accept?

```
>>>100/(1+0.1)**2
82.64462809917354
```

Here, `**` is used to perform a power function. The `%` operator is used to calculate the remainder. Refer to the following example for the implementation of these operators:

```
>>>17/4          # normal division
4.25
>>>17//4         # save as floor(17/4)
4
>>>17%4          # find out the remainder
1
```

Assume that we would receive \$10 at the end of each year forever and that the first cash flow would occur at the end of the ninth year. What is the present value if the discount rate is 8 percent per year? To solve this problem, we could combine the first and third formulae we discussed as follows:

$$PV(\text{perpetuity}, 1^{\text{st}} \text{ cash flow at } m^{\text{th}} \text{ period}) = \frac{1}{(1+R)^{m-1}} \frac{C}{R} \quad (4)$$

In this formula, C is the periodic cash flow, R is the discount rate, the first cash flow occurs at the m_{th} period. Notice that when m is 1, equation (4) collapses to equation (1). Applying equation (4), we would get a value of \$67.53 as shown in the following code:

```
>>>10/0.08/(1+0.08)**(9-1)
67.53361056274696
```


A true power function

If we deposit \$100 today and the annual interest rate is 10 percent, what is the value of our deposit one year later? If we use FV for the future value, PV for the present value, R for the annual (periodic) interest rate, and n is the number of years (periods), we get the following formula:

$$FV = PV(1 + R)^n \quad (5)$$

Note that the two variables R and n should be consistent. It means that if R is an effective monthly rate, n must be the number of months. If R is an effective annual rate, n must be the number of years. This is true for other frequencies as well, as shown in the following code:

```
>>>pv=100
>>>r=0.1
>>>n=1
>>>pv*(1+r)**n
110.00000000000001
```

Again, two multiplication signs `**` stand for a power function. Actually, Python has a built-in function for power function, `pow(x, y)` for raising x to the power of y . An example of the power function is shown as follows:

```
>>>pow(2,3)
8
>>>100*pow((1+0.1),1)
110.00000000000001
```

Apparently, in the previous example, we use two inputs for the power function `pow(x, y)`. In this two-input case, it is equivalent to $x^{**}y$. Actually, the function could have three input variables. Using `help(pow)`, we will find more information on this function refer to the following output. In the previous example, `pow((1+0.1), 1)` is the same as `pow(1+0.1, 1)`. The parentheses around `1+0.1` are not necessary, but their usage makes the expression clearer. In Python, we have the so-called *LEGB* rule related to local variables and global variables as shown in the following table:

L	Local refers to names assigned in any way within a function (def) and not declared global in that function.
E	Enclosing refers to enclosing function locals, such as names, in the local scope of any and all enclosing functions (def).

G	Global refers to names such as those assigned at the top level of a module or declared as a global variable within a function defined by <code>def</code> .
B	Built-in refers to names pre-assigned in the built-in modules, such as <code>open</code> , <code>range</code> , and <code>SyntaxError</code> .

To find out more information about a function, we use the `help()` function as follows:

```
>>>help(pow)
Help on built-in function pow in module builtins:
pow(...)
    pow(x, y[, z]) -> number
    With two arguments, equivalent to x**y. With three arguments,
    equivalent to (x**y) % z, but may be more efficient (e.g. for longs).
```

According to the previous definition, we have an example as follows:

```
>>>pow(3,10,4)
1
>>>3**10%4
1
>>>3**10
59049
>>>59049%4
1
```

Choosing appropriate precision

The default precision for Python has 16 decimal places as shown in the following example. This is good enough for most finance-related problems or research:

```
>>>7/3
2.3333333333333335
```

We could use the `round()` function to change the precision as follows:

```
>>>payment1=3/7
>>>payment1
0.42857142857142855
>>>payment2=round(y, 5)
>>>payment2
0.42857
```

Assume that the units for both `payment1` and `payment2` are in millions. The difference could be huge after we apply the `round()` function with just two decimal places! If we use one dollar as our unit, the exact payment is \$428,571. However, if we use millions instead and apply two decimal places, we end up with 430,000, which is shown in the following example. The difference is \$1,429:

```
>>>payment1*10**6
428571.4285714285
>>>payment2=round(payment1,2)
>>>payment2
0.43
>>>payment2*10**6
430000.0
```

Finding out more information about a specific built-in function

To understand each math function, we apply the `help()` function, such as `help(round)`, as shown in the following example:

```
>>>help(round)
Help on built-in function round in module builtins:
round(...)
    round(number[, ndigits]) -> number
Round a number to a given precision in decimal
digits (default 0 digits).This returns an int when
called with one argument, otherwise the same type as
the number. ndigits may be negative.
```

Listing all built-in functions

To find out all built-in functions, we perform the following two-step approach. First, we issue `dir()` to find the default name that contains all default functions. When typing its name, be aware that there are two underscores before and another two underscores after the letters of `builtins`, that is, `__builtins__`:

```
>>>dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', 'x']
```

Then, we type `dir(__builtins__)`. The first and last couple of lines of the output are given as follows:

```
>>>dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException', 'BlockingIOError', 'BrokenPipeError',
'BufferError', 'BytesWarning', 'ChildProcessError',

'range', 'repr', 'reversed', 'round', 'set', 'setattr',
'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'vars', 'zip']
```

Importing the math module

When learning finance with real-world data, we deal with many issues such as downloading data from Yahoo! finance, choosing an optimal portfolio, estimating volatility for individual stocks or for a portfolio, and constructing an efficient frontier. For each subject (topic), experts develop a specific module (package). To use them, we have to import them. For example, we can use `import math` to import all basic math functions. In the following codes, we calculate the square root of a value:

```
>>>import math
>>>math.sqrt(3)
1.732050807568772
```

To find out all functions contained in the `math` module, we call the `dir()` function again as follows:

```
>>>import math
>>>dir(math)
['__doc__', '__loader__', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
```

To make our command simpler, we could use `import math as m` instead as shown in the following example:

```
>>>import math as m
>>>m.sqrt(5)
2.23606797749979
```

The pi, e, log, and exponential functions

Pi (3.14159265) and e (2.71828) are special values in math and finance. To show their values, we have the following code. The first command imports a module called `math`. A new learner just needs to memorize those commands without a deep understanding of their meanings. Later in the book, we will devote four chapters to modules:

```
>>>import math
>>>math.pi
3.141592653589793
>>>math.e
2.718281828459045
>>>math.exp(2.2)
9.025013499434122
>>>math.log(math.e) # log() is a natural log function
1.0
>>>math.log10(10)   # log10()
1.0
```

Again, we simply type `pi` or `e` to see their values. Since they are reserved values, it is a good idea that we don't use them as our variables and don't assign a value to them.

"import math" versus "from math import *"

To make our program simpler, it is a good idea to use `from math import *`. Let's use the `sqrt()` function as an example. If we use `import math`, we have to use `math.sqrt(2)`. On the other hand, if we use `from math import *`, we simply use `sqrt(2)` as shown in the following example:

```
>>>from math import *
>>>dir()
```

```
[ '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
```

Now, we could call those functions or set of values, such as pi and e, directly.

Now, `math.pi` is not defined if we issue it from `math import *` as shown in the following code:

```
>>>pi
3.141592653589793
>>>math.pi
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    math.pi
NameError: name 'math' is not defined
```

One of the advantages of such a treatment is to make our programming a little bit easier since these functions are available directly. However, if we assign a value to e or pi, their values would be changed with our new assignment as shown in the following code. Thus, we should be careful with those specific values:

```
>> pi
3.141592653589793
>>>pi=10
>>>pi
10
```

We could import a few functions from a specific module such as `math` as shown in the following example:

```
>>>dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__']
>>>from math import sqrt,log
>>>dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'log', 'sqrt']
```

A few frequently used functions

There are several functions we are going to use quite frequently. In this section, we will discuss them briefly. The functions are `print()`, `type()`, `upper()`, `strip()`, and last expression `_`. We will also learn how to merge two string variables. The true power function `pow()` discussed earlier belongs to this category as well.

The `print()` function

Occasionally, we need to print something on screen. One way to do so is to apply the `print()` function as shown in the following example:

```
>>>import math
>>>print('pi=',math.pi)
pi= 3.141592653589793
```

At this stage, a new user just applies this format without going into more detail about the `print()` function.

The `type()` function

In Python, the `type()` function can be used to find out the type of a variable as follows:

```
>>>pv=100.23
>>>type(pv)
<class 'float'>
>>>n=10
>>>type(n)
<class 'int'>
>>>
```

From these results we know that `pv` is of the type `float` (real number) and `n` is of the type `integer`. In finance, integer and float are the two most used types. Later in the book, we will discuss other types of data (variables).

Last expression `_` (underscore)

In the interactive mode, the last printed expression is assigned to `_` as shown in the following example:

```
>>>x=1.56
>>>y=5.77
```

We can assign a string in several ways. The following three lines show two ways to assign a string to a variable and concatenation:

In this assignment, one variable uses the single quotation mark, and the second one applies double quotation marks. The result of concatenation is shown as follows:

The `upper()` function will convert the entire string into all capital letters as follows:

Please pay attention to how we call such a function. This is our first time to see such usage of a function. To remove the leading and trailing spaces, we can use the `strip()` function. The following example uses a function called `strip()` that is used to remove the leading and trailing spaces:

[37] —

We could combine the assignment operation and the `strip()` function as follows:

```
>>>z=" Hello  ".strip()
```

If we want to know about all string functions, we can issue the `dir('')` command as follows:

```
>>>dir('') # list all string functions
```

The output of this command is shown as follows:

```
>>>dir('')
['_add_', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattr__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__
le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__
setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
```

To find out specific information about a string function, we can use the following code:

```
>>>help(''.upper)
```

Help on the built-in `upper()` function is displayed as follows:

```
upper(...)
    S.upper() -> string
    Return a copy of the string S converted to uppercase.
```

Here is another example related to a built-in function called `capitalize`:

```
>>>print(''.capitalize)
```

Help on the built-in `capitalize()` function is displayed as follows:

```
capitalize(...)
    S.capitalize() -> string
    Return a copy of the string S with only its first character
    capitalized.
>>>
```

The tuple data type

For Python, a tuple is a data type or object. A tuple could contain multiple data types such as integer, float, string, and even another tuple. All data items are included in a pair of parentheses as shown in the following example:

```
>>>x= ('John',21)
>>>x
('John', 21)
```

We can use the `len()` function to find out how many data items are included in each variable. Like C++, the subscript of a tuple starts from 0. If a tuple contains 10 data items, its subscript will start from 0 and end at 9:

```
>>>x= ('John',21)
>>>len(x)
2
>>>x[0]
'John'
>>>type(x[1])
<class 'int'>
```

The following commands generate an empty tuple and one data item separately:

```
>>>z= ()
>>>type(z)
<class 'tuple'>
>>>y=(1,)          # generate one item tuple
>>>type(y)
<class 'tuple'>
>>>x=(1)           # is x a tuple?
```

For a tuple, one of its most important features as shown in the following example, is that we cannot modify the value of a tuple, that is, the tuple is immutable. In the next chapter, we will discuss another data type called list. For a list, we can modify its values.

```
>>>investment=('NPV',100,'R=',0.08,'year',10)
>>>investment[1]
100
>>>investment[1]=345
```

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    investment[1]=345
TypeError: 'tuple' object does not support item assignment
```

Assume that we are interested in assigning a name and age to a variable x as John and 21 respectively. Then we print `My name is John and 21 year-old`. We could use the tuple type. Note that `%d` is the format for the integer type. We will mention other data types in such a printing environment in later chapters:

```
>>>x=('John',21)
>>>print('My name is %s and %d year-old' % x)
My name is John and 21 year-old
```

Summary

In this chapter, we learned some basic concepts and several frequently used Python built-in functions such as basic assignment, precision, addition, subtraction, division, power function, and square root function. In short, we demonstrated how to use Python as an ordinary calculator to solve many finance-related problems.

For example, how to estimate the present value of one future cash flow, the future value of one cash flow today, the present value of a perpetuity, and the present value of a growing perpetuity. In addition, we discussed the `dir()`, `type()`, `floor()`, `round()`, and `help()` functions. We show how to get the list of all Python built-in functions and how to get help for a specific function.

Based on the understanding of the first two chapters, in the next chapter, *Chapter 3, Using Python as a Financial Calculator*, we plan to use Python as a financial calculator.

Exercises

1. What is the difference between showing the existence of our variables and showing their values?
2. How can you find out more information about a specific function, such as `print()`?
3. What is the definition of built-in functions?
4. What is a tuple?

5. How do we generate a one-item tuple? What is wrong with the following way to generate a one-item tuple?

```
>>>abc= ("John")
```

6. Can we change the values of a tuple?

7. Is `pow()` a built-in function? How do we use it?

8. How do we find all built-in functions? How many built-in functions are present?

9. When we estimate the square root of three, which Python function should we use?

10. Assume that the present value of a perpetuity is \$124 and the annual cash flow is \$50; what is the corresponding discount rate?

11. Based on the solution of the previous question, what is the quarterly rate?

12. The growing perpetuity is defined as: the future cash flow is increased at a constant growth rate forever. We have the following formula:

$$PV(\text{growing perpetuity}) = \frac{C}{R - g}$$

Here PV is the present value, C is the cash flow of the next period, g is a growth rate, and R is the discount rate. If the first cash flow is \$12.50, the constant growth rate is 2.5 percent, and the discount rate is 8.5 percent. What is the present value of this growing perpetuity?

13. For an n -day variance, we have the following formula:

$$\sigma_{n\text{-day}}^2 \sigma_{n\text{-day}}^2 = n \sigma_{\text{daily}}^2$$

Here σ_{daily}^2 is the daily variance and σ_{daily} is the daily standard deviation (volatility). If the volatility (daily standard deviation) of a stock is 0.2, what is its 10-day volatility?

14. We expect to have \$25,000 in 5 years. If the annual deposit rate is 4.5 percent, how much amount do we have to deposit today?

15. How do we convert *This is great!* into all capital letters?

16. How do we limit our output to cents, such as rounding 2.567 to 2.57?

17. What is the difference between one forward slash / and two forward slashes //?

18. We have 41 students in our class. If three students form a group for their term projects, how many groups would result and how many students remain? How about seven per group?

19. Is the `lower()` function a built-in function? How can you find its usage?

20. Explain the following results in terms of the `round()` function:

```
>>>x=5.566
>>>round(x,2)
5.57
```

21. What is the present value of a growing perpetuity when its growth rate is higher than the discount rate ($g > R$)?

$$PV(\text{perpetuity}) = \frac{C}{R - g}$$

3

Using Python as a Financial Calculator

In this chapter, we will learn how to write simple functions such as estimation of the present value for a given future value, the present value of an annuity, and the monthly payment of our mortgage. In addition, we will show how to combine two dozens of small functions as a big Python program and use it for financial estimation. In other words, we plan to create a financial calculator using Python.

In particular, we will cover the following topics:

- Writing a Python function without saving it
- Why indentation is critical in Python
- Three ways to input values and a default value for an input variable
- Using `dir()` to check the existence of our newly generated function
- Saving our `pv_f()` function
- Activating our function from our Python editor using `import()`
- While debugging a program, activate your program from a Python editor
- `import test01` versus `from test01 import *`
- Removing a function using the `del()` method
- Generating our own module
- Two types of comments
- The `if()` function
- Estimation of annuity

- Interest rate conversion and continuously compounded interest rate
- Data type – list
- NPV rule, payback rule, and **internal rate of return (IRR)** rule
- Showing certain files in a specific directory and path issue
- Using Python as a financial calculator
- Adding our project directory to the path

Writing a Python function without saving it

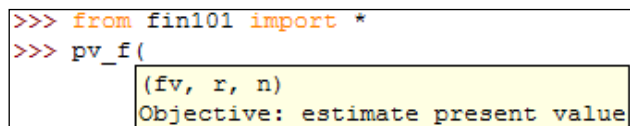
We start from the simplest way to write a Python program. The formula of estimating the present value for a given future cash flow is as follows:

$$PV = \frac{FV}{(1+R)^n} \quad (1)$$

In this equation, PV is the present value FV is the future value R is the periodic discount rate, and n is the number of periods. After launching Python, we just type the following two lines. After typing the second line, press the *Enter* key on our keyboard twice to return to the Python prompt of `>>>`:

```
>>>def pv_f(pv,r,n):  
    return fv/(1+r)**n  
  
>>>
```

The key word used to write a Python function is `def`. The function name is `pv_f`. The input variables are enclosed in the parentheses. Notice that upon pressing the *Enter* key after we type colon (:), the next line is automatically indented. Now, we are ready to call this present value function easily just as any Python built-in function. One of the wonderful features is that after typing the function name and the left parenthesis, that is, `pv_f (`, we will be given a list of input variables as shown in the following screenshot. This feature is not true for the Python Version 2.7.



```
>>> from fin101 import *  
>>> pv_f(  
    (fv, r, n)  
    Objective: estimate present value
```

To execute the function, just enter a set of ordered input values, as shown in the following code:

```
>>>pv_f(100,0.1,1)
90.9090909090909
>>>pv_f(80,0.05,6)
59.69723173093019
```

Default input values for a function

Sometimes, we set up a default input value to call our function more efficiently. Here, we use the `dir2()` function, which we created in the last section as an example. If the most frequently called directory is in C: Python32, we could set it as our default input value. This means after we activate this function and issue `>>>dir2()`, the contents under this directory will be displayed automatically, as shown in the following code:

```
def dir2(path="c:\python32"):
    from os import listdir
    print(listdir(path))
```

By the way, when a function needs inputs and there are no default input values, we would receive an error message when we don't have input values.

Indentation is critical in Python

Indentation plays a vital role in Python. Let's look at an R program. Anything between a pair of curly braces belongs to the same logic block. If we have multiple lines, the indentation is not important for R programs, as shown in the following code:

```
pv_f<-function(fv,r,n) {    # this is an R program
    pv<-fv*(1+r)^(-n)
pv
}
```

To achieve the same result in Python, we use indentations instead. This means that all the lines with the same indentation belong to the same scope, as shown in the following code:

```
def pv_f(fv,r,n):
    pv=fv/(1+r)**n
    return pv
```


The following are the ways to input values:

- In the preceding example, `pv_f(100, 0.1, 1)`, we input three values, 100, 0.1, and 1. There is no ambiguity that 100 is the future value, 0.1 is the discount rate, and 1 is the number of periods since the input variables are arranged this way. This is the first way to input values into a function.
- The second way to input values is based on key words. The advantage of this so-called *key word method* is that the order of input values does not play a role anymore. This could reduce our careless mistakes because we don't have to remember the input order when we call a function. This is especially true when we are dealing with many functions (programs) written by different developers/authors:

```
>>>pv_f(r=0.1,fv=100,n=1)
90.9090909090909
>>>pv_f(n=1,fv=100,r=0.1)
90.9090909090909
```

- The third way is the combination of the preceding two methods: ordered input first and then inputs with keywords, as shown in the following code:

```
>>>pv=pv_f(100,r=0.1,n=1)
>>>pv2=pv_f(100,n=1,r=0.1)
```

A word of caution is that the third method is the ordered inputs first, then input(s) with key words, and not the other way around.

Checking the existence of our functions

Again, we can use the `dir()` function to detect the existence of our just covered `pv_f()` function, as shown in the following code:

```
>>>dir()
['__builtins__', '__doc__', '__name__', '__package__', 'pv_f']
```

To save our file, we need to perform the following simple steps:

1. Navigate to **File | New Window Ctrl + N** and type the following two-liner code. A careful user would notice that after pressing the *Enter* key at the end of the first line, the second line will be indented automatically. While writing just two lines of code, it is not obvious. However, for a block of code with multiple lines, a correct indentation is critical. Later in the chapter, we will show and discuss this issue in more detail:

```
def pv_f(fv,r,n):
    return fv/(1+r)**n
```

2. Click on **File** and then save (*Ctrl + S*) to save the preceding two lines of code. Assume that we name the file as `test01.py`. The default directory is `Python33` in `C:` if we have installed Python 3.3.

When we save our function, the name of the file is not related to the name of the function. This is especially true when our saved file contains multiple functions.

To activate our saved function, we have the two most used ways, discussed in the *Defining a function from our Python editor* and *Activating our function using the import function* sections.

While writing a Python program, we can use any editor, such as the Python editor, Notepad, or even MS Word. If we are using MS Word, we have to remember to save our program in the `.txt` format. However, the R editor is preferred since its automatic indentation and colorful highlighting, among other features, make our debugging job easier.

Defining functions from our Python editor

After saving our previously discussed two-line code, click on **Run** and then on **Run Module F5**. If there is no error, the following line will appear. By the way, if we click on **Run** before we save the program, we will be asked to save it:

```
>>>=====RESTART =====
```

To check whether the `pv_f()` function is present in the memory, we type `dir()`, as shown in the following code:

```
>>>dir()
['_builtins_', '__doc__', '__file__', '__loader__', '__name__', '__package__', 'pv_f']

>>>
```

Now, we can execute this Python program by entering a set of three input values, as shown in the following code:

```
>>>pv_f(100,0.1,1)
90.9090909090909
```

As we discussed before, we could use the *key word* method to input values. After we quit and relaunch Python, the `pv_f()` function will be no longer available.

Activating our function using the import function

In the previous chapter, we learned that we could issue the `import math` command to upload the `math` module in order to use its included functions. Similarly, we can use the `import` function here. In other words, we have to upload or import it. Since we have the `test01.py` file saved under our default directory (Python33 in C:), we will use it, as shown in the following code:

```
>>>import test01
>>>dir()
['__builtins__', '__doc__', '__file__', '__loader__', '__name__', '__package__', 'test01']
>>>test01.pv_f(100,0.1,1)
90.9090909090909
```

Since `test01` could be treated the same way as the `math` module discussed in *Chapter 2, Using Python as an Ordinary Calculator*, we have to use `test01.pv()` instead of `pv_f()`. See the following comparison. The `ceil()` function offers the smallest integer that is bigger than the input value:

```
>>>import math
>>>math.ceil(3.5)
4
>>>import test01
>>>test01.pv_f(100,0.1,1)
90.9090909090909
```

Debugging a program from a Python editor

The preceding two sections show the two ways to activate our program, that is, from a Python editor or using the `import` function. Usually, the choice should depend on a user's preference. However, while debugging, activating our program from our Python editor is much better than the second method. If we use the second method, our program is not updated as we normally expect.

The following example contains a typo since we use both `r` (lower case) and `R` (capital letter) in the program (assume that we save it under `C:\Python33` with the name `test02.py`):

```
def pv_f(fv,r,n):
    return fv/(1+R)**n    # a typo of r
```

After issuing `from test02 import *` and calling the function, we will see an error message, as shown in the following code:

```
>>>from test02 import *
>>>pv_f(100,0.1,1)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    pv_f(100,0.1,1)
  File ".\test02.py", line 3, in pv_f
    return fv/(1+R)**n
NameError: global name 'R' is not defined
```

After correcting the typo by replacing the capital `R` with a lowercase `r`, saving our file, and repeating the first two lines of the preceding code, that is, `from test01 import *` and `pv_f(100,0.1,1)`, we will still have an error message. Only after quitting and restarting Python, can we call the updated Python program. This feature makes our debugging task difficult if we use the second way to load our updated function.

Two ways to call our `pv_f()` function

To call our `pv_f()` function included in the Python program `test01.py`, we can use `import test01` or `from test01 import *`. Obviously, it is more convenient to use `pv_f()` instead of `test01.pv_f()`. To call the function directly, we use `from test01 import *`. Refer to the following parallel structures:

```
>>>from math import *
>>>sqrt(3.5)
1.8708286933869707
>>>from test01 import *
>>>pv_f(100,0.1,2)
82.64462809917354
```

When we are sure about the existence of a specific function in `test01.py`, we can import it specifically as follows:

```
>>>from math import ceil,sqrt,pi
>>>from test01 import pv_f
```

The `del()` built-in function is used to remove a function or variable, as shown in the following code:

```
>>>del pv_f
>>>dir()
['__builtins__', '__doc__', '__loader__', '__name__',
'__package__']
```

Generating our own module

The `math` module has more than two dozen functions, such as the `pow()`, `sin()`, and `ceil()` functions. It is definitely a good idea to have just one program or file or package or module to include all of them. Let's start from the simplest case of two functions. The first function is the `pv_f()` function we discussed before. Our second function is the present value of perpetuity, which has constant cash flow at the same interval forever. If the first cash flow occurs at the end of the first period, we have the following formula:

$$PV(\text{perpetuity}) = \frac{c}{R} \quad (2)$$

Here, c is the constant periodic cash flow occurring at the end of each period, and R is the periodic discount rate. For example, if we are expected to receive \$10 at the end of each year forever, and the first cash flow would happen at the end of the first year, then the present value of such a perpetuity is \$100 ($10 / 0.1$) if the annual discount rate is 10 percent.

Again, we need to navigate to **File | (Choose) A new Window (Ctrl+N)**, and then type the following two functions:

```
def pv_f(fv,r,n):
    return fv*(1+r)**n

def pv_perpetuity(c,r):
    return c/r
```

After executing the preceding two functions, navigate to **File | Save**, and give the name `fin101.py`. After issuing `from fin101 import *`, both the functions will be available:

```
>>>from fin101 import *
>>>pv_perpetuity(100,0.1)
1000.0
```

Types of comments

When we write a complex program, the flow of logic is very important. At the same time, some good comments or explanations will help other programmers, other users, and even ourselves greatly. For a program that is not well documented, its author might have a hard time understanding it a few months later. We could add comments in many places. For example, at the beginning of the program, we could write the name of the program, objective, input variables, output variables, author or authors of the program, version of the program, and contact information. Some comments could be long, while others could be just a phrase. To satisfy various needs, Python has different types of methods to add comments. When the underlying software compiles the program, those comments could be ignored automatically.

The first type of comment

In Python, anything after `#` is a comment:

```
>>>fv=100 # this is comment
>>>fv
100
```

While writing a function, we could add several short comments such as definitions of input variables and one or two examples to explain the usage of our function:

```
# present value of perpetuity
def pv_perpetuity(c,r):
    # c is cash flow
    # r is discount rate
    return c/r
```

The second type of comment

If we have a one-line comment, it is quite convenient to use #. However, for multiple-line comments, it is cumbersome to add one # in front of each line. For those cases, we apply the second type of comments using a pair of triple quotation marks, `"""` and `"""`, to circle our comments. Thus, we could add a few lines to explain how our `pv_f()` function works, as shown in the following code:

```
def pv_f(fv,r,n):
    """
    Objective: estimate present value
               fv: future value
               r : discount periodic rate
               n : number of periods
    formula : fv/(1+r)**n
    e.g.,
    >>>pv_f(100,0.1,1)
    90.9090909090909
    >>>pv_f(r=0.1,fv=100,n=1)
    90.9090909090909
    >>>pv_f(n=1,fv=100,r=0.1)
    90.9090909090909
    """
    return fv/(1+r)**n
```

The alignment within our triple quotation marks is not important. Nevertheless, a good alignment even within our second type of comments makes our programs more readable.

Finding information about our `pv_f()` function

In the previous section, we added several lines of comments and two examples. The beauty is that we could use those comments to help other users who need more information about our function:

```
>>>help(pv_f)
Help on function pv_f in module fin101:

pv_f(fv, r, n)
    Objective: estimate present value
```

```

    fv: future value
    r : discount periodic rate
    n : number of periods
formula : fv/(1+r)**n
e.g.,
>>>pv_f(100,0.1,1)
90.9090909090909
>>>pv_f(r=0.1,fv=100,n=1)
90.9090909090909
>>>pv_f(n=1,fv=100,r=0.1)
90.9090909090909

```

Note that only the comments immediately under the first line of `def` would be shown after we type `help(pv_f)`. It means that the other, later comments will not be shown. It also means that if we add any line, such as `a=1`, before our comments, then nothing would be shown after we issue `help(pv_f)`.

The if() function

The present value of a growing perpetuity has the following formula:

$$PV(\text{growing perpetuity}) = \frac{C}{R - g} \quad (3)$$

Here, C is the first cash flow occurring at the end of the first period, R is the effective periodic rate, and g is the constant growth rate. The second and the third future cash flows will be $C(1+g)$ and $C(1+g)^2$, respectively. A necessary condition for the correctness of equation (3) is that the discount rate should be greater than the growth rate, that is, R should be greater than g . What is the present value if C is \$10, R is 10 percent, and g is 12 percent? The wrong answer is -500. For these similar cases, we could use the `if()` function to print an error message instead of offering the wrong answer, as shown in the following code:

```

def pv_growing_perpetuity(c,r,g):
    if(r<g):
        print("r<g !!!!")
    else:
        return(c/(r-g))

```


We could try different sets of input values, as shown in the following code:

```
>>>pv_growing_perpetuity(10,0.1,0.08)
499.9999999999999
>>>pv_growing_perpetuity(10,0.1,0.12)
r<g !!!!!
```

Annuity estimation

An annuity is the same periodic cash flows occurring at the same interval for n periods. There are two types of annuity: ordinary annuity when cash flows occur at the end of each period and annuity due when cash flows happen at the beginning of each period. Here is an example. We are going to receive \$100 at the end of each year for the next 7 years. The formulae to estimate the present value and the future value of an annuity are as follows when their first cash flows happens at the end of the first period:

$$PV(annuity) = \frac{PMT}{R} \left[1 - \frac{1}{(1+R)^n} \right] \quad (4A)$$

$$PV(annuity\ due) = \frac{PMT}{R} \left[1 - \frac{1}{(1+R)^n} \right] (1+R) \quad (4B)$$

$$FV(annuity) = \frac{PMT}{R} \left[(1+R)^n - 1 \right] \quad (5A)$$

$$FV(annuity\ due) = \frac{PMT}{R} \left[(1+R)^n - 1 \right] (1+R) \quad (5B)$$

Here, PV is the present value, PMT is the equal periodic payment, R is the periodic discount rate, and n is the number of periods. In the preceding annuity formulae, PMT , R , and n should be consistent. It means that PMT , R , and n should possess the same frequency. For example, for mortgage estimation, PMT is the monthly payment, R is an effective monthly rate, and n is the number of months. If the annuity enjoys a constant growth rate of g , its present value is as follows:

$$PV(\text{growing annuity}) = \frac{PMT}{R - g} \left[1 - \left(\frac{1+g}{1+R} \right)^n \right] \quad (6)$$

Similarly, the future value of a growing annuity is as follows:

$$PV(\text{growing annuity}) = \frac{PMT}{R - g} \left[(1+R)^n - (1+g)^n \right] \quad (7)$$

Converting the interest rates

Assume that bank A offers 5 percent compounding monthly, while bank B offers 5.1 percent compounding quarterly. Which bank should we borrow from in order to enjoy a lower interest rate? These examples are associated with conversion between different interest rates. First, let's look at the following formula used to estimate **effective annual rate (EAR)** for a given **Annual Percentage Rate (APR)**.

$$EAR = \left(1 + \frac{APR}{m} \right)^m - 1 \quad (8)$$

Here, m is the compounding frequency within one year. For example, if the annual rate is 5 percent compounding semiannually, its equivalent effective annual rate will be 5.0625 percent. From the two banks' offers, we would choose the offer of bank A since the cost of borrowing (effective annual rate) is cheaper, as shown in the following code:

```
>>> (1+0.05/2)**2-1
0.05062499999999992
>>> (1+0.051/4)**4-1
0.051983692114066615
```

For a mortgage estimate, if the annual rate is 5 percent, compounding monthly, the effective monthly rate will be 0.41667 (0.05/12). However, if the given rate is 5 percent compounding semiannually, what is the corresponding effective monthly rate? To convert one effective interest rate to another effective interest rate, and from one APR to another APR, we have to perform the following steps. First, we have to estimate an effective rate from a given APR and compounding frequency according to the following formula:

$$R_m^{effective} = \frac{APR}{m} \quad (9)$$

Here, R_m is the effective rate, APR/m is the annual percentage rate compounded m times per year, and m is the annual compounding frequency. For example, if a given APR is 5 percent compounding semiannually, the effective semiannual rate is 2.5 percent. Combining equations (8) and (9), we have the following equivalency:

$$\left(1 + R_{m_1}^{effective}\right)^{m_1} = \left(1 + R_{m_2}^{effective}\right)^{m_2} \quad (10)$$

Or, we can write the following formula:

$$\left(1 + \frac{APR_1}{m_1}\right)^{m_1} = \left(1 + \frac{APR_2}{m_2}\right)^{m_2} \quad (11)$$

Here, APR_1 and APR_2 are the annual percentage rates and m_1 and m_2 are their compounding frequencies. To find out the effective rate with compounding frequency m_2 for a given APR_1 and m_1 , we have the following formula:

$$R_{m_2}^{effective} = \left(1 + \frac{APR_1}{m_1}\right)^{\frac{m_1}{m_2}} - 1 \quad (12)$$

Assume that we plan to borrow \$300,000 to buy a house with a 30-year loan. What is the monthly payment if our bank offers us 5 percent annual rate compounding semiannually? From equation (4), we know that if we know R , then we can calculate our monthly payment (pmt) since pV is 300000 and n is 30×12 . By applying equation (12), we would have a monthly mortgage rate of 0.4123915 percent, as shown in the following code:

```
>>>r=(1+0.05/2)**(2/12)-1
>>>r
```

```

0.0041239154651442345
>>>pv=300000
>>>n=30*12
>>>pmt=pv*r/(1-1/(1+r)**n)
>>>pmt
1601.0720364262665

```

Based on the preceding estimation, the effective monthly rate is 0.41239155 percent and the monthly payment is \$1601.07. At the end of this chapter, we have several related exercises; refer to 3.18, 3.19, and 3.20.

Continuously compounded interest rate

In the previous section, our compounding frequency could be annual ($m=1$), semiannual (2), quarterly (4), monthly (12), or daily (365). If the compounding frequency increases further and further, such as by the hour, minute, and second, the limit is called continuously compounded. The following is the conversion formula:

$$R_c = m * \ln \left(1 + \frac{APR}{m} \right) \quad (13)$$

Here, R_c is the continuously compounded rate, $\ln()$ is a natural log function, APR is the annual percentage rate, and m is the compounding frequency per year. For the natural log function, refer to the following code:

```

>>>import math
>>>math.e
2.718281828459045
>>>math.log(math.e)
1.0

```

For example, if a given APR of 5 percent is compounded semiannually, its corresponding continuously compounded rate will be 4.9385225 percent, as shown in the following code:

```

>>>import math
>>>2*math.log(1+0.05/2)
0.04938522518074283

```

In the next chapter, for a call option, the risk-free rate used is continuously compounded.

A data type – list

A list is another type of data. Unlike a tuple, which uses parentheses, lists use square brackets, [and], as shown in the following code. A list could include different types of data, such as string, integer, float, and a list itself:

```
>>>record=['John',21,'Engineer',3]
>>>record
['John', 21, 'Engineer', 3]
```

Like tuples, the first data item starts with a subscript of zero. If we want to list all the data items from subscript 1 to the end of the list, we use `record[1:]`, and to list all the data items from subscript 2 to the end of the list, we use `record[2:]`, as shown in the following code:

```
>>>len(record)
4
>>>record[0]
'John'
>>>record[2:]
['Engineer', 3]
```

Unlike tuples, which are immutable, lists can be modified.

```
record[0]='Mary'
>>>record[0]
'Mary'
```

Net present value and the NPV rule

Net present value (NPV) is defined as the difference between the present value of all the benefits and costs, as shown in the following formula:

$$NPV = PV(\text{benefits}) - PV(\text{costs}) \quad (14)$$

Assume that we have a 5-year project with an initial investment of \$100 million. The future cash flows at the end of each year for the next five years are \$20m, \$40m, \$50m, \$20m, and \$10m, respectively. If the discount rate for such type of investments is 5 percent per year, should we take the project? First, we have to estimate the NPV of our project. Second, we have to apply the following decision rule (the NPV rule):

$$\begin{cases} \text{If } NPV(\text{project}) > 0 \text{ accept} \\ \text{If } NPV(\text{project}) < 0 \text{ reject} \end{cases} \quad (15)$$

If we manually estimate the NPV, we can do the following calculations

```
>>>-100 + 20/(1+0.05)+40/(1+0.05)**2 +50/(1+0.05)**3+20/(1+0.05)**4+10/
(1+0.05)**5
22.80998927303707
```

Since the NPV of our project is positive, we should accept it.

It is quite tedious to type each value. For example, we typed 0.05 (the r value) five times. To make our typing a little easier, we could assign a value to r , as shown in the following code:

```
>>>r=0.05
>>>-100 + 20/(1+r)+40/(1+r)**2 +50/(1+r)**3+20/(1+r)**4+10/(1+r)**5
22.80998927303707
```

A much better way is to generate an NPV function by entering the discount rate and all cash flows including today's investment. After launching Python, navigate to **File** | **New Window Ctrl+N**, and then type the following lines. Navigate to **Run** | **Run Module 5**. Note that while asking for a filename, you could enter `npv_f.py`:

```
def npv_f(rate, cashflows):
    total = 0.0
    for i, cashflow in enumerate(cashflows):
        total += cashflow / (1 + rate)**i
    return total
```

In the preceding function, the first line defines a function with the `def` key word. The function name is `npv_f` instead of `npv`. On the other hand, if we choose `npv` as our function name, and when a user chooses `npv` as a variable, the function would not be available any more. The second line defines a `total` variable and initializes its value as 0. Based on their indentations, the third and fourth lines could be considered as one block. The `for` loop has two intermediate variables `i` (from 0 to 5) and `cashflow` (for values -100, 20, 40, 50, 20, and 10). Notice that the `cashflow` and `cashflows` variables are different. The Python command of `x+=v` is equivalent to `x=x+v`. We will discuss the `for` loop and other loops in more detail in *Chapter 10, Python Loops and Implied Volatility*. If there is no error message, we could use the `npv_f()` function easily. To find information about the `enumerate()` function, we could use `help enumerate()`.

```
>>>r=0.05
>>>cashflows=[-100,20,40,50,20,10]
>>>npv_f(r,cashflows)
22.80998927303707
```

To make our function more user friendly, we could add the definitions of those two input variables along with one or two examples.

Defining the payback period and the payback period rule

A payback period is defined as the number of years we need to recover our initial investment. In the preceding example, we needed more than two years but less than three years to recover our \$100 million investment since we recovered \$60 million two years and \$110 million in three years.

If the revenue is assumed to be distributed evenly within a year, the payback period of this project will be 2.8 years, as shown in the following code:

```
>>>40/50+2
2.8
```

The payback rule is that if the estimated payback period of our project is less than a critical value ($T_{critical}$), we accept the project. Otherwise, we reject it, as given in the following conditions:

$$\begin{cases} \text{If } Payback(project) < T_{critical} & \text{accept} \\ \text{If } Payback(project) > T_{critical} & \text{reject} \end{cases} \quad (16)$$

Compared with the *NPV* rule, the payback period rule has many shortcomings, including the fact that it ignores the time value of money and cash flows after the payback period, and the benchmark of the critical value is ad hoc. The advantage is that this rule is very simple.

Defining IRR and the IRR rule

IRR is the discount rate resulting in a zero NPV. The IRR rule is that if our project's IRR is bigger than our cost of capital, we accept the project. Otherwise, we reject it, as shown in the following conditions:

$$\begin{cases} \text{If } IRR(\text{project}) > R_{\text{capital}} & \text{accept} \\ \text{If } IRR(\text{project}) < R_{\text{capital}} & \text{reject} \end{cases} \quad (17)$$

The Python code to estimate an IRR is as follows:

```
def IRR_f(cashflows, iterations=100):
    rate=1.0
    investment=cashflows[0]
    for i in range(1, iterations+1):
        rate*=(1-npv_f(rate, cashflows)/investment)
    return rate
```

At this stage, this program is quite complex. If a user cannot grasp its meaning, it this won't impact on them understanding the rest of the chapter. The `range(1, 100+1)` statement will give us the range from 1 to 101. The `i` variable takes values from 1 to 101. In other words, the fifth line will repeat 101 times. An assumption behind the fifth line is that R and NPV are negatively correlated. In other words, an increase in discount rate R leads to a smaller NPV value.

The key is the fifth line, `rate*=(1-npv_f(rate, cashflows)/investment)`. Let us simplify it as the following equation:

$$R_{i+1} = R_i * (1 - k)$$

If R_i leads to a positive NPV value, we would increase our discount rate, that is, R_{i+1} will be bigger than R_i , that is, k will be a small negative number. On the other hand, if R_i leads to a negative NPV value, we would reduce our discount rate, that is, a positive k . The following result is based on the first round of the loop when R is 100 percent:

```
>>>cashflows=[-100,20,40,50,20,10] # cash flows
>>>npv_f(1,cashflows)                # R(1) is 100%
-72.1875                               # negative NPV
>>>cashflows[0]                       # we would reduce R
-100
>>>k=npv_f(1,cashflows)/cashflows[0]
>>>k                                   # k is positive
0.721875
>>>1*(1-k)
0.27812499999999996                  # R(2) will be 0.278
```

The `IRR_f()` function depends on the `npv_f()` function we generated before, as shown in the following code:

```
>>>from npv_f import *
>>>cashflows=[-100,20,40,50,20,10]
>>>x=IRR_f(cashflows)
>>>x
0.13601259394401546
```

Thus, if our cost of capital is 5 percent, we accept the project. We can verify the preceding result by using the `npv_f` function and use it as our new discount rate, as shown in the following code:

```
>>>npv_f(r,x)
-1.4210854715202004e-14
```

Showing certain files in a specific subdirectory

Sometimes we want to know which files are available under a specific directory or subdirectory. Assume that we save both `npv_f.py` and `pv_f.py` at `C:\Python33\`. To double check their existence, we have the following code:

```
>>>from os import listdir
>>>listdir("c:\Python33")
```

Actually, we can create a function called `dir2()` to mimic the `dir()` function. The difference is that the `dir()` function lists variables and functions in the memory, while the `dir2()` function shows files in a given directory. Thus, the `dir()` function does not need an input, while our `dir2()` function needs an input value, that is, a directory, as shown in the following code:

```
def dir2(path):
    from os import listdir
    print(listdir(path))
```

After we save `dir2.py` at `C:\Python33\`, we issue the following command to view it:

```
>>>from dir2 import *
>>>path='c:\python33'
```

Using Python as a financial calculator

Based on what we have learned in this chapter, we are ready to put together about two dozen functions related to finance 101 or other finance courses, and call our final big program `fin101.py`. After debugging all the errors, we could call this module (program) easily by issuing the command `from fin101 import *`. A more detailed procedure of how to generate such an R-based financial calculator is as follows:

1. Create an empty Python file called `fin101.py` and save it under our default directory, that is, `Python33` in `C:`, or other designated directory.
2. Add the `pv_f()` function to `fin101.py`, and debug the program until it is error free.
3. Repeat the previous step by adding one function at a time until `fin101.py` includes all our functions.
4. Generate a function called `fin101`, which is used to offer a list of all our included functions. A few lines are shown in the following code. Assuming that our `fin101.py` file has only two functions, we could generate a very simple `fin101()` function, as shown in the following code:

```
def fin101():
    """
    1) Basic functions:
        PV:  pv_f, pv_annuity, pv_perpetuity
        FV:  fv_f, fv_annuity, fv_annuity_due
    2) How to use pv_f?
        >>>help(pv_f)
    """
```

5. To use this function, we have the following code. Assume here that the file called `fin101.py` includes three functions, `pv_f()`, `fv_f()`, and `fin101()`, as shown in the following code:

```
>>>from fin101 import *
>>>help(fin101)
```

Adding our project directory to the path

In the previous discussion, we assumed that all our programs are saved under our default directory, that is, `Python33` in `C:` or other similar directories. Obviously, it is not convenient if we plan to save our Python programs for a specific project under our designated directory. Assume that all of our programs related to our investment courses are located at `C:\yan\Teaching\Python_for_Finance\codes_chapters\`. To include it in our path, we have the following Python code:

```
import sys
myFolder="C:\\Yan\\Teaching\\Python_for_Finance\\codes_chapters"
if myFolder not in sys.path:
    sys.path.append(myFolder)
```

To double check, we use the `print(sys.path)` command, as shown in the following code:

```
>>>import sys
>>>print(sys.path)
['', 'C:\\Python33\\Lib\\idlelib', 'C:\\windows\\system32\\python33.zip', 'C:\\Python33\\DLLs', 'C:\\Python33\\lib', 'C:\\Python33', 'C:\\Python33\\lib\\site-packages', 'C:\\Yan\\Teaching\\Python_for_Finance\\codes_chapters']
```

An alternative way is to use the `path` function, as shown in the following code (only a few lines are shown to save space):

```
>>>import os
>>>help(os.path)
Help on module ntpath:
NAME
    ntpath - Common pathname manipulations, WindowsNT/95 version.
```

The following table summarizes the functions that we can include in our big Python program to have most of the functions of a financial calculator. The table has the following notations:

- PV is the present value
- FV is the future value
- R is the rate of interest for this period (discount rate)
- n is the number of periods
- C is a recursive cash flow for a perpetuity or annuity
- PMT is a recursive cash flow for a perpetuity or annuity (same as C)
- g is the growth rate for a growing perpetuity (annuity)
- APR is the annual percentage rate
- R_c is continuously compounded rate
- m is the compounding frequency each year

Note that C , R , and n should be consistent, that is, with the same frequency (unit). The recommended formulae for a Python financial calculator are as follows:

$FV = PV(1 + R)^n$	$PV = \frac{FV}{(1 + R)^n}$
$PV(\text{perpetuity}) = \frac{c}{R}$	Assume that the first cash flow occurs at the end of the first period
$PV(\text{growing perpetuity}) = \frac{c}{R - g}$	Assume that the first cash flow occurs at the end of the first period. $R > g$
$PV(\text{annuity}) = \frac{PMT}{R} \left[1 - \frac{1}{(1 + R)^n} \right]$	Assume that the first cash flow occurs at the end of the first period
$FV(\text{annuity}) = \frac{PMT}{R} \left[(1 + R)^n - 1 \right]$	Assume that the first cash flow occurs at the end of the first period
$PV(\text{perpetuity due}) = PV(\text{perpetuity}) * (1 + R)$	Due: cash flow occurs at the beginning of each period
$PV(\text{annuity due}) = PV(\text{annuity}) * (1 + R)$	$FV(\text{annuity due}) = FV(\text{annuity}) * (1 + R)$

PV (bond)=PV(coupons) + PV(face value)	$PV(bond) = \frac{c}{R} \left[1 - \frac{1}{(1+R)^n} \right] + \frac{FV}{(1+R)^n}$
$EAR = \left(1 + \frac{APR}{m} \right)^m - 1$	• <i>EAR</i>: It is the effective annual rate • <i>APR</i>: It is the annual percentage rate • <i>m</i>: It is the compounding frequency per year
From one APR to another APR	
For example, APR_1 , m_1 and m_2 are given, what is APR_2 ?	$\left(1 + \frac{APR_1}{m_1} \right)^{m_1} = \left(1 + \frac{APR_2}{m_2} \right)^{m_2}$
From one effective rate to another effective rate	$\left(1 + R_{m_1}^{effective} \right)^{m_1} = \left(1 + R_{m_2}^{effective} \right)^{m_2}$
From one APR to continuously compounded rate, R_c	$R_c = m * \ln \left(1 + \frac{APR}{m} \right)$
From R_c to APR	$APR = m * \left(e^{\frac{R_c}{m}} - 1 \right)$

Summary

In this chapter, we learned how to write simple functions, such as functions to estimate the present value of one future cash flow, the future value of one present value, the present value of annuity, the future value of annuity, the present value of perpetuity, the price of a bond, and **Internal Rate of Return (IRR)**. Obviously, it is difficult and time consuming to activate several dozens of small functions separately. How to combine many small functions into a single Python program was the focus of this chapter. In other words, we planned to generate a Python program (module) called `fin101.py` and used it as a financial calculator. After launching Python, we issued one command `from fin101 import *` to activate or load or import all of our functions. To sum it up, after reading this chapter, you should be able to write a Python program (module), including almost all the formulae used in courses such as corporate finance and investment.

In *Chapter 4, 13 Lines of Python to Price a Call Option*, we will show how to write a Python program to price a call option. In total, there are only 13 lines of code. To make it suitable for any and all backgrounds, there are no mathematic formulae related to the option theory.

Exercises

1. How do we generate a Python program without saving it? Please generate a function that triples any input value.
2. How do we use comments effectively when we write a Python program?
3. What are the advantages and disadvantages of using a default input value or values?
4. In this chapter, while writing a present value function, we use `pv_f()`. Why not use `pv()`, the same as the following formula?

$$PV = \frac{FV}{(1+R)^n} \quad (1)$$

Here PV is the present value, FV is the future value, R is the periodic discount rate, and n is the number of periods.

5. How do we debug a complex Python program?
6. What is the efficient way to test a Python program?
7. Why is indentation critical in Python?
8. How to put two formulae together, such as the present value of one future cash flow and the present value of an annuity?
9. How many types of comments are available? How do we use them effectively?
10. Write a `fin101.py` program and put together as many formulae as possible, such as `pv_f()`, `pv_perpetuity()`, `pv_perpetuity_due()`, `dpv_annuity()`, `dpv_annuity_due()`, `fv_annuity()`, among others.
11. Assume that we have a set of small programs put together called `fin101.py`. What is the difference between the two Python commands, `import fin101` and `from fin101 import *`?
12. How to prevent erroneous inputs such as negative interest rate?

13. We know that the following code works. Here we assume that C:\Python33 exists.

```
>>>from os import listdir
>>>listdir("c:\python33")
```

However, the following function does not work. Why?

```
def dir3(path):
    from os import listdir
    listdir(path)
```

14. Assume that both npv_f.py and irr_f.py exist under C:\Python32\. What is wrong with the following code?

```
>>>from irr_f import *
>>>import npv_f
>>>dir()
['IRR_f', '__builtins__', '__doc__', '__name__', '__package__', 'glob',
'npv_f']
>>>IRR_f(0.04, [-100, 50, 50, 50])
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    IRR_f(0.04, [-100, 50, 50, 50])
  File "C:\Python32\irr_f.py", line 3, in IRR_f
    investment = cashflows[0]
TypeError: 'float' object is not sub
```

15. Write a Python program to estimate payback period. For example, the initial investment is \$256, and the expected future cash inflows in the next 7 years will be \$34, \$44, \$55, \$67, \$92, \$70, and \$50. What is the project's payback period in years?

16. If the discount rate is 7.7 percent per year, what is the discounted payback period? Note: The discount payback period looks at how to recover our initial investment by checking the summation of present values of future cash flows.

17. Assume that we have a directory C:\python32 and we plan to list all Python programs under it with a .py extension. We could use the following code to achieve this:

```
>>>import glob
>>>glob.glob("c:\python33\*.py")
```

Write a Python function called `dir2()` with an input string variable, that is, we can call it `dir2("c:\python32\*.py")`.

18. Write a Python program to convert a given annual percent rate with compounding frequency to another effective rate and APR.

$$\left(1 + \frac{APR_1}{m_1}\right)^{m_1} = \left(1 + \frac{APR_2}{m_2}\right)^{m_2}$$

20. Write a Python conversion code to convert one rate to another by combining the following equations:

$$\left(1 + \frac{APR_1}{m_1}\right)^{m_1} = \left(1 + \frac{APR_2}{m_2}\right)^{m_2}$$

$$e^{R_c} = \left(1 + \frac{APR}{m}\right)^m$$

3.21 Based on the following code, write a Python program to add our path, such as `addPath('c:\my_project')`:

```
>>>import sys
>>>myFolder="C:\\Python_for_Finance\\codes_chapters"
>>>if myFolder not in sys.path:
    sys.path.append(myFolder)
```

22. Assume that we have 10 projects under 10 directories, such as `C:\teaching\python\`, `c:\projects\python\`, `c:\projects\portfolio\`, and `c:\projects\investments\`. Write a Python program (module) that has 10 functions with default input values of those 10 directories. After running the first function, the path of our first project will be automatically added to our path.

4

13 Lines of Python to Price a Call Option

To many readers, option theory is like rocket science. In order to make option theory less intimidating, we deliberately avoided any mathematical formula in this chapter. Literally, the focus of the whole chapter is around 13 lines of Python code. An option buyer pays to acquire the right to buy (or sell) something in the future while an option seller receives an upfront payment to bear an obligation to sell to (or buy from) the option buyer. A call option buyer has the right to buy a stock at a fixed price and at a fixed date in the future. A European option can only be exercised when the option expires, while an American option could be exercised any time before or at the maturity date.

In this chapter, we will cover the following topics:

- 13 lines of Python code to price a call option
- Writing a Python function without saving it
- Using the empty shell method to write a complex Python program
- Using the comment-all-out method to write a complex Python program
- How to debug other programs

We have the following five lines of Python code to price a European call option:

```
from math import *
def bs_call(S,X,T,r,sigma):
    d1 = (log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    d2 = d1-sigma*sqrt(T)
    return S*CND(d1)-X*exp(-r*T)*CND(d2)
```

The first line imports the `math` module since we need the `log()`, `sqrt()`, and `exp()` functions in our program. To price a call, we have five input variables: `S` is the current stock price, `X` is the exercise price (a fixed price), `T` is the maturity (in years), `r` is the continuously compounded risk-free rate, and `sigma` is the volatility of the underlying security (such as a stock). Since our imported `math` module does not include a **cumulative standard normal distribution (CND)** in the previous program, we have to write a Python program ourselves. Obviously, if we could import a module with the `CND` function, as shown in the following code, we could price a call option with just five lines! In *Chapter 6, Introduction to NumPy and SciPy*, we will show you how to achieve this by using a module called `SciPy`:

```
def CND(X):
    (a1,a2,a3,a4
,a5)=(0.31938153,-0.356563782,1.781477937,-1.821255978,1.330274429)
    L = abs(X)
    K=1.0/(1.0+0.2316419*L)
    w=1.0-1.0/sqrt(2*pi)*exp(-L*L/2.)*(a1*K+a2*K*K+a3*pow(K,3)+a4*pow(K,4)+
a5*pow(K,5))
    if X<0:
        w = 1.0-w
    return w
```

For the `CND` function, `X` is the input value. The second line assigns five values to `a1`, `a2`, `a3`, `a4`, and `a5`. A tuple is used to save space. After launching Python, click on **File | New Window Ctrl + N**, then type the previous 13 lines of code. After typing, we will save the file, click on **Run**, and then click on **Run Module F5**. If there is no error, we will see the following result:

```
>>>=====RESTART =====
```

Now, we are ready to use our just finished Python program to price a call option. With a set of input values of `S`, `X`, `T`, `r`, and `sigma`, we could estimate the Black-Scholes' call option easily. Based on the following set of input values, the call price is \$2.28:

```
>>>bs_call(40,42,0.5,0.1,0.2)
2.2777859030683096
```

Until now, we know that there are two separate functions associated with the pricing of a call option with a total of 13 lines. This is a perfect case based on which we could explain how to write a relatively complex Python program. For the rest of the chapter, we will show two ways of writing a Python program: the empty shell method and comment-all-out method.

Writing a program – the empty shell method

To vividly describe this method, we call it the empty shell method. The method works like this: generate an empty shell first and test it, then add one line and test it. If there is no error, add one more line and test the program. Repeat this procedure until you finish the whole program. The `CND` function is used as an example in the following case:

1. After launching Python, click on **File** then **New Window Ctrl + N**. Generate the following empty shell:

```
def CND(x):
    return x
```

2. Click on **File | Save**; for example, save it as `cnd.py`.
3. Click on **Run** and then click on **Run from module F5**. The following line will appear:

```
>>>=====RESTART =====
```

4. To test our program, we will enter various values. If we enter 1, the output would be 1. If we enter 5, then the output will be 5, as shown in the following example:

```
>>>CND(1)
1
```

5. We add one line as shown in the following code:

```
def CND(x):
    (a1,a2,a3,a4,a5)=(0.31938153,-0.356563782,1.781477937,-1.
821255978,1.330274429)
    return a1
```

6. Note that the return value is `a1` instead of `x`. Click on **Run** and then click on **Run from module F5**; you will see that the following line appears:

```
>>>=====RESTART =====
```

7. We could test this program by entering any value. Here is an example:

```
>>>CND(1)
0.31938153
```

8. Repeat the previous step until we complete this `CND()` function (program) as shown in the following code:

```
from math import *
def CND(X):
```

```
(a1,a2,a3,a4,a5)=(0.31938153,-0.356563782,1.781477937,
-1.821255978,1.330274429)

L = abs(X)

K=1.0/(1.0+0.2316419*L)

w=1.0-1.0/sqrt(2*pi)*exp(-L*L/2.)*(a1*K+a2*K*K+a3*pow(K,3)
+a4*pow(K,4)+a5*pow(K,5))

if X<0:
    w = 1.0-w

return w
```

9. The line `from math import *` is needed since we are using the `sqrt()` function contained in the `math` module of our `CND()` function. We could test this function with different input values as follows:

```
>>>CND(0)
0.5000000005248086
>>>CND(-2.3229)
0.010092238515047591
>>>CND(1.647)
0.9502209933627817
```

10. Since a standard normal distribution is symmetric, its cumulative distribution will be 0.5 at 0. It is also well known that a z value of -2.33 corresponds to 1 percent and 1.647 for 95 percent. We could use the Excel `normdist()` function to verify our `CND` function. The structure of the related Excel function is *normdist(x, mean, standard deviation, cumulative)*. The last one takes 0 for the normal distribution and 1 for the cumulative distribution as shown in the following screenshot:

f_x	=NORMDIST(0,0,1,1)			f_x	=NORMDIST(-2.3229,0,1,1)			f_x	=NORMDIST(1.647,0,1,1)		
	D	E	F		D	E	F		D	E	F
	0.5				0.010092				0.950221		

Writing a program – the comment-all-out method

Here is the logic behind this comment-all-out method: type all the lines and then comment them all. After that, we release one line at a time to debug. We use the Black-Scholes' call option as an example in the following case:

1. Launch Python, click on **File | New Window Ctrl + N**. Generate the five lines of code mentioned in step 3. We include several bugs on purpose.
2. Click on **File** and save it.
3. Comment out the entire program by using a pair of triple quotation marks. Since we need some output, we add a return line as follows:

```
def bs_call(S,X,T,r,sigma):
    """
    d1 = (lo(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    d2 = d1-sigma*sqrt(T)
    return SCN(d1)-X*exp(-r*T)*CND(d2)
    """
    return (X)
```

4. Again, click on **Run** and then **Run from module F5**. We will see that the following output appears:

```
>>>=====RESTART =====
```

5. We test it by using any set of input values as follows:

```
>>>bs_call(40,40,0.5,0.1,0.2)
40
>>>bs_call(40,42,0.5,0.1,0.2)
42
```

6. Note that the output takes the value of the second input variable since we designed it this way.
7. The last step is to release one line at a time. If there is an error or errors, modify the line accordingly:

```
def bs_call(S,X,T,r,sigma):
    d1 = (lo(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    """
```

```
d2 = d1-sigma*sqrt(T)
return SCN(d1) -X*exp(-r*T)*CND(d2)
"""
return(x)
```

8. When calling the function, we will see an error message meaning that the `lo()` function does not exist. Then we realize that we mistyped `log(S/X)` as `lo(S/X)` as shown in the following code:

```
>>>bs_call(40,40,0.5,0.1,0.2)
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    bs_call(40,40,0.5,0.1,0.2)
  File "<pyshell#49>", line 2, in bs_call
    d1 = (lo(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
NameError: global name 'lo' is not defined
```

9. Repeat step 4 until we finish the whole program.

Using and debugging other programs

Quite often than not, we might start our project based on a program or programs written by others, such as the programs by our fellow researchers, other students, teachers, programs downloaded from the Internet, or the old programs we wrote ourselves a long time ago. As the first step, we need to know whether our borrowed program contains any errors. These two methods could be used to debug such a program. In a sense, the second method, comment-all-out method, is preferred since it might save us some typing or copy-and-paste time. To debug other programs, the key is to find the locations of the errors. Here is a very useful Python program to get data from Yahoo! Finance: <http://goldb.org/ystockquote.html>. A beginner could download the program to try small functions contained in the program.

Summary

In this chapter, we deliberately avoided any mathematical formula related to the option theory. Thus, within a short period of time, such as less than two hours, a reader who has no clue about the option theory could price a European call option based on the famous Black-Scholes model.

In *Chapter 5, Introduction to Modules*, we will introduce modules formally, and it is the first chapter of a three-chapter block that focuses on modules. A module is a package or a set of programs written by one or a group of experts for a specific purpose. For example, in *Chapter 6, Introduction to NumPy and SciPy*, we will show that five lines, instead of 13 lines, could be used to price a call option since we could use the cumulative standard normal distribution function contained in the `SciPy` module.

Exercises

1. Write a Python program to price a call option.
2. Explain the empty shell method that is used while writing a complex Python program.
3. Explain the logic behind the so-called comment-all-out method when writing a complex Python program.
4. Explain the usage of a return value when we debug a program.
5. When we write the CND, we could define `a1`, `a2`, `a3`, `a4`, and `a5` separately. What are the differences between the following two approaches?

Current approach:

```
(a1,a2,a3,a4
,a5)=(0.31938153,-0.356563782,1.781477937,-1.821255978,1.330274429)
```

An alternative approach:

```
a1=0.31938153
a2=-0.356563782
a3=1.781477937
a4=-1.821255978
a5=1.330274429
```

6. What are the definitions of effective annual rate, effect semi-annual rate, and risk-free rate for the call option model? Assuming that the current annual risk-free rate is 5 percent, compounded semi-annually, which value should we use as our input value for the Black-Scholes call option model?
7. What is the call premium when the stock is traded at \$39, the exercise price is \$40, the maturity date is three months, the risk-free rate is 3.5 percent (compounding continuously), and the volatility is 0.15 per year?

8. Repeat the previous exercise if the risk-free rate is still 3.5 percent per year but compounded semiannually.
 9. What are the advantages and disadvantages of using other programs?
 10. How to debug other programs?
 11. Write a Python program to convert any given **annual percentage rate (APR)** that is compounded m times per year to a continuously compounded interest rate.
 12. How do you improve the accuracy of the cumulative normal distribution?
 13. What is the relationship between APR and a continuously compounded rate (R_c)?
 14. A stock has the current stock price of \$52.34. What is its call price if the exercise price is the same as its current stock price, matures in six months with a 0.16 annual volatility, and the risk-free rate is 3.1 percent (compounded continuously)?
 15. For a set of S , X , T , r , and σ , we could estimate a European call option by using those 13 lines of Python codes. When the current stock price, S , increases while other input values are the same, will the call price increase or decrease? Why?
 16. Show the previous result graphically.
 17. When the exercise price, X , increases, the value of a call will fall. Is this true? Why?
 18. If other input values are constant, the value of the call premium will increase if the σ of the stock increases. Is this true? Why?
 - 19*. For a set of input values of S , X , T , r , and σ , we could use the code in this chapter to price a European call option, that is, C . On the other hand, if we observe a real-world price of call premium (C_{obs}) with a set of values S , X , T , and r , we could estimate an implied volatility (σ). Specify a trial-and-error method to roughly estimate the implied volatility (if a new learner could not get this question, it is perfectly fine since we will devote a whole chapter to discuss how to do it).
 - 20*. According to the so-called put-call parity, holding a call option with enough cash at maturity (X dollars) is equivalent to holding a put option with a share of underlying stock in hand. Here, both call and put options have the same exercise price (X) with the same maturity (T) and both are European options. If the stock price is \$10, the exercise price is \$11, maturity is six months, and the risk-free rate is 2.9 percent (compounded semi-annually), what is the price of a European put option?*
- Indicates an advanced level question

5

Introduction to Modules

In this chapter, we will discuss modules, which are packages written by experts or any individual to serve special purposes. In this book, we will use about a dozen modules in total. Thus, knowledge related to modules is vitally important in our understanding of Python and its application to finance.

In particular, we will cover the following topics:

- What is a module and how do we import a module?
- Showing all functions contained in an imported module
- Adopting a short name for an imported module
- Comparing between `import math` and `from math import *`
- Deleting an imported module
- Importing a few functions from a module
- Finding out all built-in modules and all available (reinstalled) modules
- How to find a specific uninstalled module
- Finding the location of an imported module
- Module dependency
- One super package including many modules
- Online searching of modules and videos on how to install a module

What is a module?

A module is a package that is written by experts, users, or even a new beginner who is very good in a specific area to serve a specific objective. For example, a Python module is called `quant`, which is for quantitative financial analysis. The `quant` combines `SciPy` and `DomainModel`. The module contains a domain model that has exchanges, symbols, markets, and historical price, among other things. Modules are very important in Python. In this book, we will mention about a dozen modules implicitly or explicitly. In particular, we will discuss five modules in detail: `NumPy` and `SciPy` in *Chapter 6, Introduction to NumPy and SciPy*; `Matplotlib` in *Chapter 7, Visual Finance via Matplotlib*; and `Pandas` and `Statsmodels` in *Chapter 8, Statistic Analysis of Time Series*. As of November 6, 2013, there are 24,955 Python packages with different areas available according to the Python Package Index at <https://pypi.python.org/pypi?%3Aaction=browse>. For the financial and insurance industry, there are 687 modules currently available.

Importing a module

Assume that we want to estimate the square root of the number three. However, after issuing the following lines of code, we would encounter an error message:

```
>>>sqrt(3)
SyntaxError: invalid syntax
>>>
```

The reason is that the `sqrt()` function is not a built-in function. To use the `sqrt()` function, we need to import the `math` module first as follows:

```
>>>import math
>>>x=math.sqrt(3)
>>>round(x,4)
1.7321
```

To use the `sqrt()` function, we have to type `math.sqrt()` if we use the `import math` command to upload the `math` module. In addition, after issuing the command `dir()`, we will see the existence of the `math` module, which is the last one in the output shown as follows:

```
>>>dir()
['_builtins_', '__doc__', '__name__', '__package__', 'math']
```

In addition, when a module is reinstalled, we could use `import x_module` to upload it. For instance, the `math` module is a built-in module, and it is preinstalled. Later in the chapter, we show how to find all built-in modules. In the preceding output, after issuing the command `dir()`, we also observe `__builtins__`. This `__builtins__` module is different from other built-in modules, such as the `math` module. It is for all built-in functions and other objects. Again, we could issue `dir(__builtins__)` to list all built-in functions as shown in the following screenshot:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '__debug__', '__doc__', '__import__', '__name__', '__package__', '__path__', '__all__', '__any__', '__apply__', '__basestring__', '__bin__', '__bool__', '__buffer__', '__bufferarray__', '__bytes__', '__callable__', '__chr__', '__classmethod__', '__cmp__', '__coerce__', '__compile__', '__complex__', '__copyright__', '__credits__', '__delattr__', '__dict__', '__dir__', '__divmod__', '__enumerate__', '__eval__', '__execfile__', '__exit__', '__file__', '__filter__', '__float__', '__format__', '__frozenset__', '__getattr__', '__globals__', '__hasattr__', '__hash__', '__help__', '__hex__', '__id__', '__input__', '__int__', '__intern__', '__isinstance__', '__issubclass__', '__iter__', '__len__', '__license__', '__list__', '__locals__', '__long__', '__map__', '__max__', '__memoryview__', '__min__', '__next__', '__object__', '__oct__', '__open__', '__ord__', '__pow__', '__print__', '__property__', '__quit__', '__range__', '__raw_input__', '__reduce__', '__reload__', '__repr__', '__reversed__', '__round__', '__set__', '__setattr__', '__slice__', '__sorted__', '__staticmethod__', '__str__', '__sum__', '__super__', '__tuple__', '__type__', '__unichr__', '__unicode__', '__vars__', '__xrange__', '__zip__']
>>>
```

Adopting a short name for an imported module

Sometimes, the name of a module is long or difficult to type. To save some typing effort during our programming, we could use the command `import x_module as y_name` as shown in the following lines of code:

```
>>>import sys as s
>>>import time as tt
>>>import numpy as np
>>>import matplotlib as mp
```

When calling a specific function contained in an imported module, we use the module's short name as shown in the following lines of code:

```
>>>import time as tt
>>>tt.localtime()
time.struct_time(tm_year=2013, tm_mon=7, tm_mday=21, tm_hour=22, tm_min=39, tm_sec=41, tm_wday=6, tm_yday=202, tm_isdst=1)
```

Although we are free to choose any short names for our imported modules, we should respect some convention, such as using `np` for NumPy and `sp` for SciPy. One added advantage of using such commonly used short names is to make our programs more readable by others.

Showing all functions in an imported module

Assume that we are interested in finding all functions contained in the `math` module. For that, first we import it, and then we use `dir(math)` as shown in the following lines of code:

```
>>>import math
>>>dir(math)
['_doc_', '__loader__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>>
```

Comparing "import math" and "from math import *"

Although we have discussed this issue in the previous chapters, for the completeness of this chapter, we will mention it one more time. To make our program simpler, it is a good idea to use `from math import *`. This is especially true for a beginner who has just started to learn Python programming. Let's take a look at the following lines of code:

```
>>>from math import *
>>>sqrt(3)
1.7320508075688772
```

Now, all functions contained in the module will be available directly. On the other hand, if we use `import math`, we have to specify `math.sqrt()` instead of `sqrt()`. When we become more familiar with Python, it would be a good idea to use the `import module`. There are two reasons for this. First, we know exactly from which module we apply our function. Second, we might have written our own function with the same name as the function contained in another module. A module name ahead of a function will distinguish it from our own function as shown in the following lines of code:

```
>>>import math
>>>math.sqrt(3)
1.7320508075688772
```

Deleting an imported module

The `del()` function is used to remove an imported/uploaded module as shown in the following lines of code:

```
>>>import math
>>>dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'math']
>>>del math
>>>dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__']
```

However, if we use `from math import *`, we cannot remove all functions by issuing `del math`. We have to remove individual functions separately. The following two commands demonstrate such an effect:

```
>>>from math import *
>>>del math
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    del math
NameError: name 'math' is not defined
```

In the following code example, we remove the `sqrt()` function from our memory. It is obvious that there is no reason to do so. This operation is used purely for illustration purposes. The following two commands delete the `sqrt()` function first, then try to call it:

```
>>>del sqrt
>>>sqrt(2)
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    sqrt(2)
NameError: name 'sqrt' is not defined
```

Importing only a few needed functions

In *Chapter 4, 13 Lines of Python to Price a Call Option*, we need three functions, `log()`, `exp()`, and `sqrt()`, to price a call option. To make those three functions available, we issue the command `from math import *` to upload the `math` module which contains those functions. After issuing `from math import *`, all the functions included in the module will be available as shown in the following lines of code:

```
from math import *
def bs_call(S,X,T,r,sigma):
    d1 = (log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    d2 = d1-sigma*sqrt(T)
    return S*CND(d1)-X*exp(-r*T)*CND(d2)
```

On the other hand, if we need just a few functions, we could specify their names as shown in the following lines of code:

```
>>>from math import log, exp, sqrt
>>>round(log(2.3),4)
0.8329
>>>round(sqrt(3.7),4)
1.9235
```

Finding out all built-in modules

A tuple of strings gives the names of all modules that are compiled into this Python interpreter. The key word here is `builtin__module__names`, as shown in the following screenshot:

```
>>> import sys as s
>>> s.builtin__module__names
('builtin__', 'main__', '_ast', '_bisect', '_codecs', '_codecs_cn', '_codecs_hk', '_codecs_iso2022', '_codecs_jp', '_codecs_kr', '_codecs_tw', '_collections', '_csv', '_functools', '_heapq', '_hotshot', '_io', '_json', '_locale', '_lsprof', '_md5', '_multibytecodec', '_random', '_sha', '_sha256', '_sha512', '_sre', '_struct', '_subprocess', '_symtable', '_warnings', '_weakref', '_winreg', '_array', '_audioop', '_binascii', '_cPickle', '_cStringIO', '_cmath', '_datetime', '_errno', '_exceptions', '_future_builtins', '_gc', '_imageop', '_imp', '_itertools', '_marshal', '_math', '_mmap', '_msvcrt', '_nt', '_operator', '_parser', '_signal', '_strop', '_sys', '_thread', '_time', '_xxsubtype', '_zipimport', '_zlib')
```

Note that a built-in module does not mean that they are currently available. For example, from the preceding output, we know that the `math` module is preinstalled. If we want to call a function, such as `sin()`, contained within the `math` module, we have to import it first. On the other hand, the function called `modules.keys()` in the `sys` module only lists the imported modules, as shown in the following screenshot:

```
>>> s.modules.keys()
['heapq', 'functools', 'pyreadline.console.ansi', 'ctypes.os', 'sysconfig', 'logging.os', 'ctypes._endian', 'encodings.encodings', 'pyreadline.modes.basemode', 'logging.stat', 'logging.weakref', 'imp', 'collections', 'logging.thread', 'logging.socket', 'pyreadline.py3k_compat', 'zipimport', 'string', 'encodings.utf_8', 'pyreadline.lineeditor.lineobj', 'logging.logging', 'pyreadline.rlmain', 'signal', 'logging.handlers', 'threading', 'pyreadline.keysyms.common', 'ctypes.wintypes', 'pyreadline.console.console', 'cStringIO', 'logging.threading', 'locale', 'pyreadline.logger', 'pyreadline.lineeditor.wordmatcher', 'atexit', 'pyreadline.unicode_helper', 'encodings', 'logging.traceback', 'pyreadline.console', 'abc', 'ctypes.util', 'pyreadline.modes.notemacs', 're', 'pyreadline.keysyms.keysyms', 'ntpath', 'pyreadline.lineeditor', 'pyreadline.keysyms.winconstants', 'encodings.ascii', 'math', 'UserDict', 'ctypes', 'fnmatch', 'ctypes', 'pyreadline', 'ctypes.struct', 'codecs', 'logging.sys', 'struct', 'functools', '_locale', 'logging', 'socket', 'thread', 'StringIO', 'traceback', 'weakref', 'itertools', 'os', 'future', 'collections', 'sre', 'pyreadline.lineeditor.history', 'builtin', 'logging.errno', 'operator', 'logging.re', 'ctypes._ctypes', '_heapq', 'ctypes.sys', 'encodings.cp437', 'errno', '_socket', 'logging.struct', 'sre_constants']
```


Finding out all the available modules

To find all available modules, we need to activate the `help` window first. After that, we issue modules. The following graph illustrates the result after issuing such a `help` command:

```
>>> help(<>)

Welcome to Python 2.7! This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics". Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".
```

Then, we issue `modules` under the Python `help>` prompt as shown in the following screenshot:

```
ArgImagePlugin      _psutil_mswindows  inspect             sklearn
BaseHTTPServer      _pyio              io                  smtpd
Bastion              _pytest            isapi               smtplib
BdfFontFile         _random            itertools           sndhdr
Bio                  _sha               itsdangerous        socket
BioSQL              _sha256            Jinja2              sphinx
BmpImagePlugin       _sha512            json                spyderlib
BufRStubImagePlugin _socket            keyring             spyderplugins
CGIHTTPServer       _sqlite3           keyword             sqlalchemy
Canvas              _sre               kiva                 sqlite3
ConfigParser        _ssl               launcher            sre
ContainerIO         _strptime          lib2to3             sre_compile
Cookie              _struct            linecache            sre_constants
Crypto               _subprocess        llpython            sre_parse
CurlImagePlugin     _symtable          llvm                 ssl
Cython              _system_path       llvm_array          sspi
DcxImagePlugin      _testcapi          llvm_builder         sspicon
Dialog              _threading_local   llvmmath             stat
DocXMLRPCServer     _tkinter            llmppy              statsmodels
EpsImagePlugin      _warnings          locale               statvfs
ExifTags            _weakref           logging              storemagic
FileDialog           _weakrefset        lxml                 string
```

The last lines are shown as follows:

```
license      httplib      setuptools   xmlilib
locale      httpplib     sgmlib       xmlrpclib
lsprof      idlelib      sha          xxsubtype
markerlib   ihooks      shelve       yaml
md5         imageop     shlex        zipfile
msi         imaplib     shutil       zipimport
multibytecode imghdr      signal       zlib
multiprocessing imp         site         zmq
nsis        importlib   six
osx_support imputil     skimage

Enter any module name to get more help.  Or, type "modules spam" to search
for modules whose descriptions contain the word "spam".
help>
```

To find a specific module, we just type modules followed by the module's name. Assume that we are interested in the module called `Matplotlib`. Then, we issue `modules matplotlib` in the help window. The precondition is that the `matplotlib` module is preinstalled. If it is not, we could get an error message. The following graph shows the output after issuing the command of `modules matplotlib`:

```
help> modules matplotlib
Here is a list of matching modules.  Enter any module name to get more help.
IPython.core.magics.pylab - Implementation of magic functions for matplotlib/pyl
ab support.
IPython.core.pylabtools - Pylab (matplotlib) support utilities.
bokeh.plotting - Command-line driven plotting functions, a la Matplotlib / Matl
ab / etc.
matplotlib - This is an object-oriented plotting library.
matplotlib.cm - Nothing here but dictionaries for generating LinearSegmentedCol
ormaps.
matplotlib.cntr
matplotlib.delaunay
matplotlib.image
matplotlib.mathtext_data - font data tables for truetype and afm computer moder
n fonts
matplotlib.path
matplotlib.png
matplotlib.pylab_helpers - Manage figures for pyplot interface.
matplotlib.tri
matplotlib.windowing
```

Finding the location of an imported module

In *Chapter 6, Introduction to Numpy and SciPy*, we will show you how to download and install NumPy in detail. Here, we just assume that we have installed a module called NumPy. The following are the three ways to find the location of an imported module:

The first method (way) is to use `print(np.__file__)` as shown in the following lines of code:

```
>>>import numpy as np
>>>print(np.__file__)
c:\Anaconda\lib\site-packages\numpy\__init__.py
```

The second method is to use `np.__file__` without invoking the `print()` function as shown in the following lines of code:

```
>>>np.__file__
'C:\Anaconda\lib\site-packages\numpy\__init__.py'
```

The third method is to just type `np` after we have imported the module as shown in the following lines of code:

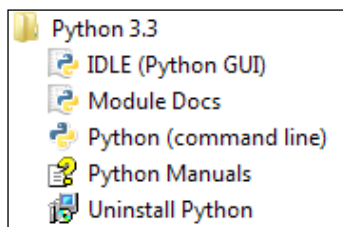
```
>>>np
<module 'numpy' from 'c:\Anaconda\lib\site-packages\numpy\__init__.py'>
```

Some readers might ask why we should care about the location of an installed module in the first place. For example, after finding out that the location of the NumPy module is `c:\Anaconda\lib\site_package\numpy\`, we could go to that subdirectory directly to find more information about this module. By going to the subdirectory shown earlier, you will find several interesting subdirectories, such as `doc`, `random`, and `tests`. From those specific subdirectories, a user, especially a new learner, could locate many useful Python programs. If we accidentally generate a module that has the same name as an existing module, we should know where to find the existing module for our debugging purposes. A good practice is to avoid using the same name in the first place.

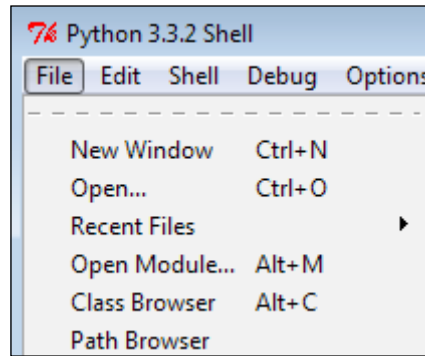
More information about modules

To get more information on modules, perform the following steps:

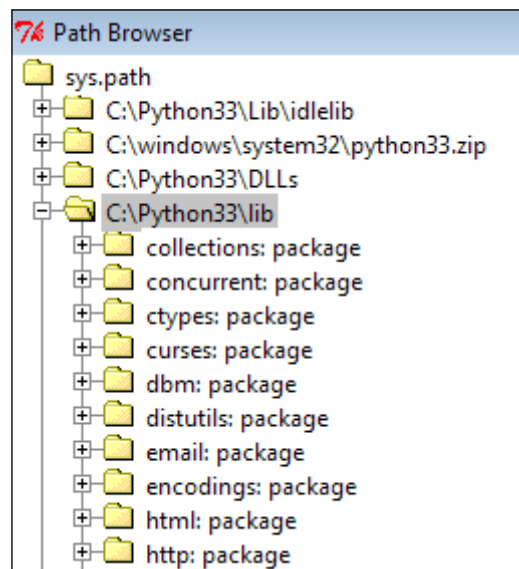
1. Navigate to **All Programs | Python 3.3 | Module Docs** as shown in the following screenshot:



2. We can also browse the path to find preinstalled modules. After launching Python, click on **File** and then on **Path Browser** as shown in the following screenshot:



3. After we click on **Path Browser**, we will be given a list of packages, that is, modules, as shown in the following screenshot:



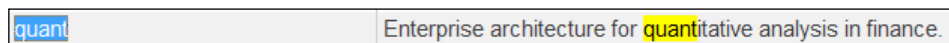
Finding a specific uninstalled module

Assume that we are interested in using a Python module called `quant` for quantitative analysis. Usually, we would try to import it first. Let's take a look at the following lines of code:

```
>>>import quant
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    import quant
ImportError: No module named 'quant'
>>>
```

Since we encounter an error message, it means that the module called `quant` is not preinstalled. The following are the steps by which we could locate this module:

1. Go to the Python Package Index at <https://pypi.python.org/>.
2. From this web page, choose **Browse all packages**.
3. Choose **Python** under programming languages (<https://pypi.python.org/pypi?%3Aaction=browse>).
4. Next, choose **Financial and Insurance Industry** (<https://pypi.python.org/pypi?%3Aaction=browse>).
5. Click on **show all** (<https://pypi.python.org/pypi?action=browse&c=33&c=214>).
6. Search the list by using the keyword `quant`.
7. Finally, we locate the package. The following screenshot shows what we would see at the end of the last discussed step:



After clicking on **quant**, we will find its related web page at <https://pypi.python.org/pypi/quant/0.8>. Copy this address to your hard drive and ensure that Python includes that path.

Module dependency

At the very beginning of this book, we argue that one of the advantages of using Python is that it is a rich source of hundreds of special packages called modules. To avoid duplicated efforts and to save time in developing new modules, later modules choose to use functions developed on early modules; that is, they depend on early modules.

The advantage is obvious because developers could save lots of time and effort when building and testing a new module. However, one disadvantage is that installation becomes difficult.

There are two competing approaches. The first approach is to bundle everything together and make sure that all parts play together nicely, thus avoiding the pain of installing n packages independently. This is wonderful assuming that it works. A potential issue is that the updating of individual modules might not be reflected in the super package. The second approach is to use minimal dependencies. It causes fewer headaches for the package maintainer, but for users who have to install several components, it can be more of a hassle. Linux has a better way: using the package installer. The publishers of the package can declare dependencies and the system tracks them down assuming they are in the Linux repository. `SciPy`, `NumPy`, and `quant` are all set up like that, and it works great.

In the next chapter, we will discuss the two most important modules, `NumPy` and `SciPy`. However, their installation individually is not a simple exercise. Instead, we choose a super package called `Anaconda`. After we install `Anaconda`, both `NumPy` and `SciPy` would be available.

The following table presents about a dozen Python modules related to finance:

Name of the module	Description
<code>Ystockquote</code>	Retrieves stock quote data from Yahoo! Finance
<code>Quant</code>	Enterprise architecture for quantitative analysis in finance
<code>trytond_currency</code>	Trytond module with currencies
<code>Economics</code>	Functions and data manipulation for economics data
<code>trytond_project</code>	Project module with project management
<code>trytond_analytic_account</code>	Financial and accounting module performs analytic accounting with any number of analytic charts and the analytic account balance report
<code>trytond_account_statement</code>	Financial and accounting module with Statement and Statement journal
<code>trytond_stock_split</code>	Trytond module to split stock move
<code>trytond_stock_forecast</code>	Trytond module with stock forecasts

Name of the module	Description
Finance	Calculates financial risks and is optimized for ease of use through class construction and operator overload
FinDates	Deals with dates in finance

The link that lists some of the most commonly used Python modules is available at <https://wiki.python.org/moin/UsefulModules>.

The Python Package Index is available at <https://pypi.python.org/pypi>. Note that you have to register first to view the complete list.

Summary

In this chapter, we discussed modules, such as finding all available or installed modules and how to install a new module. In this book, we will use a few dozen modules. Thus, an understanding of modules is vital. For example, a module called `Matplotlib`, which is useful in various graphs, will be used intensively in the next chapter, where we discuss the famous Black-Scholes-Merton option model.

In the next chapter, we will introduce the two most important modules: `NumPy` and `SciPy`. Those two modules are used intensively for scientific and financial computation. In this book, many chapters depend on these two modules. In addition, many other modules depend on these two modules.

Exercises

1. What is a module?
2. How do we find the number of functions in a module called `math`?
3. What is the difference between `import math` and `from math import *`?
4. How do we upload a few specific functions?
5. Where can we find manuals related to a module?
6. How do we remove a module?
7. If I am interested in just a few functions contained in the `math` module, how could I import just them?
8. What is module dependency? Why is this an issue when we install a module?
9. There is a module called `NumPy`. How many modules does it depend on?
10. How could we, as beginners, write a simple module?
11. How many modules are currently available in Python? How do we find a list of them?
12. Describe the major contents of a module called `zipimport`.

6

Introduction to NumPy and SciPy

In this chapter, we will introduce the two most important modules, called `NumPy` and `SciPy`, which are used intensively for scientific and financial computation based on Python. In this book, many chapters and modules depend on these two modules.

In particular, we will cover the following topics:

- Installation of `NumPy` and `SciPy`
- Launching Python from Anaconda
- Examples of using `NumPy` and `SciPy`
- Showing all functions in `NumPy` and `SciPy`
- Getting more information about a specific function
- Understanding the `list` data type
- Array in `NumPy`, logic relationship related to arrays
- Working with arrays of ones, zeros, and identity matrix
- Performing array operations: `+`, `-`, `*`, and `/`
- The `x.sum()` dot function
- Looping through an array
- A list of subpackages for `SciPy`
- Cumulative standard normal distribution
- Generating random numbers
- Statistic submodule (`stats`) from `SciPy`
- Interpolation, linear equations, and optimization

- Linear regression and **Capital Assets Pricing Model (CAPM)**
- Retrieving data from an external text file
- Installing NumPy independently
- Understanding the data types

Installation of NumPy and SciPy

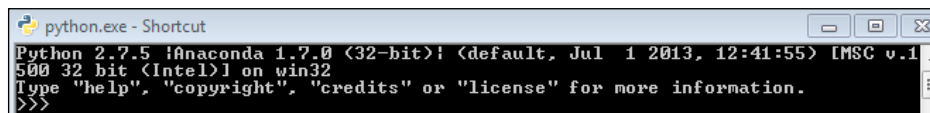
In the previous chapter, we discussed the module dependency and how it might be difficult to install a new module because it depends on many other modules. Fortunately, several super packages, such as Anaconda and Enthought Canoy, could be used to install several (or many) modules simultaneously. In this book, Anaconda is used, since it contains both NumPy and SciPy. To install it, we have to perform the following two steps:

1. Go to <http://continuum.io/downloads>.
2. According to your machine, choose the appropriate package, such as
Windows 32-bit / 280M / md5: 91a6398f63a8cc6fa3db3a1e9195b3bf.

After installation, we will see the anaconda folder in C: for the Windows system. The Python version accompanying Anaconda is 2.7. In addition to NumPy and SciPy, Anaconda contains the other three modules we plan to discuss in this book. For a complete list of modules, about 124, included in Anaconda, you can check these out at <http://docs.continuum.io/anaconda/pkgs.html>.

Launching Python from Anaconda

For the Window version, we locate the executable `python.exe` file at `c:\anaconda` and then click on it. The following Python window will appear:



To test whether we have correctly installed NumPy and SciPy, we need to type the following two commands to import them. If there is no error, it means that we have installed them correctly.

```
>>>import numpy as np
>>>import scipy as sp
```

In the last several chapters, we know that we could use `from numpy import *` instead of `import numpy as np` to make all functions included in the NumPy module into our namespace. However, most developers prefer to use `import numpy as np`. From now on, we will follow this tradition. The second reason is that using `sp.pv()` instead of `pv()` makes it clearer that the `pv()` function is from a module called `sp`. To generate a Python icon on the desktop, we generate a shortcut first, and then move it from `c:\anaconda` to our desktop.

Examples of using NumPy

In the following examples, the `np.size()` function from NumPy shows the number of data items of an array, and the `np.std()` function is for the standard deviation of an array:

```
>>>import numpy as np
>>>x= np.array([[1,2,3],[3,4,6]])    # 2 by 3 matrix
>>>np.size(x)                        # number of data items
>>>6
>>>np.size(x,1)                     # show number of columns
3
>>>np.std(x)
1.5723301886761005
>>>np.std(x,1)
Array([ 0.81649658,  1.24721913])
>>>total=x.sum()                    # pay attention to the format
>>>z=np.random.rand(50)# 50 random obs from [0.0, 1)
>>>y=np.random.normal(size=100)     # from standard normal
>>>r=np.array(range(0,100),float)/100 # from 0, .01,to .99
```

Compared with a Python array, a NumPy array is a contiguous piece of memory that is passed directly to LAPACK, which is a software library for numerical linear algebra, under the hood so that matrix manipulation is very fast in Python. An array in NumPy is like a matrix in MATLAB. Unlike `lists` in Python, an array should contain the same data type as shown in the following line of code:

```
>>>np.array([100,0.1,2],float)
```

The real type is `float64`, and the default for numerical values is also `float64`. In the preceding example, we could view that the `np.array()` function converts a list with the same data type, `integer` in this case, to an array. If we want to change the data type, we could specify it in the second input value as shown in the following lines of code:

```
>>>x=[1,2,3,20]
>>>y=np.array(x1,dtype=float)
>>>y
array([ 1.,  2.,  3., 20.] )
```

In the previous example, `dtype` is the keyword specifying the data type. For a `list`, we could have different data types without causing any problems. However, when converting a `list` containing different data types into an array, we will get an error message as shown in the following lines of code:

```
>>>x2=[1,2,3,"good"]
>>>x2
[1, 2, 3, 'good']
>>>y3=np.array(x2,float)
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    y3=np.array(x2,float)
ValueError: could not convert string to float: 'good'
. 1])
```

Examples of using SciPy

The following are a few examples based on the functions contained in the `SciPy` module. The first example is related to the **Net Present Value (NPV)** function:

```
>>>import scipy as sp
>>>cashflows=[50,40,20,10,50]
>>>npv=sp.npv(0.1,cashflows) #estimate NPV
>>>round(npv,2)
>>>144.56
```

The `np.npv()` function estimates the present values for a given set of future cash flows. The first input value is the discount rate, and the second input is an array of future cash flows. This `np.npv()` function mimics Excel's `NPV` function. Like Excel, `np.npv()` is not a true `NPV` function. It is actually a `PV` function. It estimates the present value of future cash flows by assuming the first cash flow happens at the end of the first period. An example of using an Excel `NPV()` function is as follows:

<i>f_x</i>	=NPV(0.1,E1:I1)				
D	E	F	G	H	I
-100	50	40	20	10	50
\$131.41					

While using just one future cash flow, the meaning of the `np.npv()` function is much clearer as shown in the following lines of code:

```
>>>c=[100]
>>>npv=np.npv(0.1,c)
>>>round(npv,2)
>>>90.91
>>>round(100/(1+0.1),2)
>>>90.91
```

Based on the preceding argument, if we have an initial cash outflow, such as 100, we have to modify our second input value, an array, accordingly as shown in the following lines of code:

```
>>>cashflows=[-100,50,40,20,10,50]
>>>npv=sp.npv(0.1,cashflows[1:])+cashflow[0]
>>>round(npv,2)
>>>31.41
```

The `sp.pmt()` function is used to answer the following question: What is the monthly cash flow to pay off a mortgage of \$250,000 over 30 years with an **annual percentage rate (APR)** of 4.5 percent, compounded monthly?

```
>>>payment=sp.pmt(0.045/12,30*12,250000)
>>>round(payment,2)
-1266.71
```

Similar to the `sp.npv()` function, the `sp.pmt()` function mimics the equivalent function in Excel, as we will see in the following screenshot. The input values are: the effective period rate, the number of the period, and the present value.

<i>f_x</i>	<code>=PMT(0.045/12,30*12,250000)</code>		
D	E	F	
(\$1,266.71)			

The `sp.pv()` function replicates the Excel `PV()` function. The format for `sp.pv()` is `sp.pv(rate,nper,mpt,fv=0.0,when='end')`. The discount rate is `rate`, `nper` is the number of periods, and `fv` is the future value with a default value of zero. The last input variable specifies whether the cash flows are at the end of each time period or at the beginning of each period. By default, it is at the end of each period. The following commands show how to call this function:

```
>>>pv1=sp.pv(0.1,5,0,100) # pv of one future cash flow
>>>round(pv1,2)
-92.09
>>>pv2=sp.pv(0.1,5,100) # pv of annuity
>>>round(pv2,2)
-379.08
```

The `sp.fv()` function has settings similar to that of `sp.pv()`. In finance, we estimate both arithmetic and geometric means, which are defined in the following formulas. For n numbers of $x_1, x_2, x_3, x_2, x_3, \dots$ and x_n , we have:

$$\text{Arithmetic mean} = \frac{\sum_{i=1}^n x_i}{n} \quad (1)$$

$$\text{Geometric mean} = \left(\prod_{i=1}^n x_i \right)^{1/n} \quad (2)$$

Here, $\prod_{i=1}^n x_i = x_1 * x_2 * \dots * x_n$. Assume that we have three numbers a , b , and c . Then their arithmetic mean is $(a+b+c)/3$, while their geometric mean is $(a*b*c)^{(1/3)}$. For three values of 2, 3, and 4, we have the following two means:

```
>>>(2+3+4)/3.
>>>3.0
>>>geo_mean=(2*3*4)**(1./3)
>>>round(geo_mean,4)
2.8845
```

If we have n returns, the formula to estimate their arithmetic mean remains the same. However, the geometric mean formula for returns is different as shown in the following screenshots:

$$\text{Arithmetic mean} = \frac{\sum_{i=1}^n R_i}{n} \quad (3)$$

$$\text{Geometric mean} = \left[\prod_{i=1}^n (1 + R_i) \right]^{1/n} - 1 \quad (4)$$

We could use the `sp.prod()` function, which gives us the products of all data items to estimate the geometric means for a given set of percentage returns as shown in the following lines of code:

```
>>>import scipy as sp
>>>ret=sp.array([0.1,0.05,-0.02])
>>>sp.mean(ret)                                # arithmetic mean
0.04333
>>>pow(sp.prod(ret+1),1./len(ret))-1 # geometric mean
0.04216
```

Two other useful functions are `sp.unique()` and `sp.median()` as shown in the following code:

```
>>>sp.unique([2,3,4,6,6,4,4])
Array([2,3,4,6])
>>>sp.median([1,2,3,4,5])
3.0
```

The Python `sp.npv()`, `sp.pv()`, `sp.fv()`, and `sp.pmt()` functions behave like Excel's `npv()`, `pv()`, `fv()`, and `pmt()` functions, respectively. They have the same sign convention: the sign of the present value is the opposite of that of the future value. In the following example to estimate a present value, if we enter a positive future value, we will end up with a negative present value:

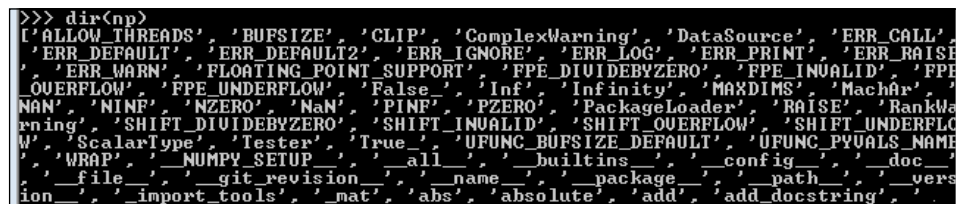
```
>>>import scipy as sp
>>>round(sp.pv(0.1,5,0,100),2)
>>>-62.09
>>>round(sp.pv(0.1,5,0,-100),2)
>>>62.09
```


Showing all functions in NumPy and SciPy

There are several ways to find out all the functions contained in a specific module. First, we can read related manuals. Second, we can issue the following lines of code:

```
>>>import numpy as np
>>>dir(np)
```

To save space, only a few lines of code are shown in the following screenshot:



```
>>> dir(np)
['ALLOW_THREADS', 'BUFSIZE', 'CLIP', 'ComplexWarning', 'DataSource', 'ERR_CALL',
 'ERR_DEFAULT', 'ERR_DEFAULT2', 'ERR_IGNORE', 'ERR_LOG', 'ERR_PRINT', 'ERR_RAISE',
 'ERR_WARN', 'FLOATING_POINT_SUPPORT', 'FPE_DIVIDEBYZERO', 'FPE_INVALID', 'FPE_
OVERFLOW', 'FPE_UNDERFLOW', 'False', 'Inf', 'Infinity', 'MAXDIMS', 'MachAr', '
NaN', 'NINF', 'NZERO', 'NaN', 'PINF', 'PZERO', 'PackageLoader', 'RAISE', 'RankWa
rning', 'SHIFT_DIVIDEBYZERO', 'SHIFT_INVALID', 'SHIFT_OVERFLOW', 'SHIFT_UNDERFLO
W', 'ScalarType', 'Tester', 'True', 'UFUNC_BUFSIZE_DEFAULT', 'UFUNC_PYVALS_NAME',
 'WRAP', 'NUMPY_SETUP', 'all', 'builtins', 'config', 'doc',
 '_file', '_git_revision', '_name', '_package', '_path', '_vers
ion', '_import_tools', '_mat', 'abs', 'absolute', 'add', 'add_docstring', ...]
```

Actually, a better way is to generate an array with all of those functions as follows:

```
>>>x=np.array(dir(np))
>>>len(x)          # showing the length of the array
571
>>>x[200:210]      # showing 10 lines starting from 200
```

The following screenshot shows the output:



```
>>> x[200:210]
array(['equal', 'errstate', 'exp', 'exp2', 'expand_dims', 'expm1',
       'extract', 'eye', 'fabs', 'fastCopyAndTranspose', 'fft',
       'fill_diagonal', 'find_common_type', 'finfo', 'fix', 'flatiter',
       'flatnonzero', 'flexible', 'fliplr', 'flipud', 'float', 'float16',
       'float32', 'float64', 'float_', 'floating', 'floor', 'floor_divide',
       'fmax', 'fmin', 'fmod', 'format_parser', 'frexp', 'frombuffer',
       'fromfile', 'fromfunction', 'fromiter', 'frompyfunc', 'fromregex',
       'fromstring', 'fv', 'generic', 'genfromtxt', 'get_array_wrap',
       'get_include', 'get_numarray_include', 'get_printoptions',
       'getbuffer', 'getbufsize', 'geterr'],
      dtype='<S22')
>>>
```

Similarly, to find all the functions in SciPy, we use the `dir()` function after we load the module as shown in the following lines of code:

```
>>>import scipy as sp
>>>dir(sp)
```

The first part is shown in the following screenshot:

```
>>> dir(sp)
['ALLOW_THREADS', 'BUFSIZE', 'CLIP', 'ComplexWarning', 'DataSource', 'ERR_CALL',
'ERR_DEFAULT', 'ERR_DEFAULT2', 'ERR_IGNORE', 'ERR_LOG', 'ERR_PRINT', 'ERR_RAISE',
'ERR_WARN', 'FLOATING_POINT_SUPPORT', 'FPE_DIVIDEBYZERO', 'FPE_INVALID', 'FPE_
OVERFLOW', 'FPE_UNDERFLOW', 'False_', 'Inf', 'Infinity', 'MAXDIMS', 'MachAr', '
NAN', 'NINF', 'NZERO', 'NaN', 'PIHF', 'PZERO', 'PkgLoader', 'RAISE', 'RankWa
rning', 'SHIFT_DIVIDEBYZERO', 'SHIFT_INVALID', 'SHIFT_OVERFLOW', 'SHIFT_UNDERFLO
W', 'ScalarType', 'Tester', 'True_', 'UFUNC_BUFSIZE_DEFAULT', 'UFUNC_PYVALS_NAME',
'__file__', '__name__', '__numpy_version__', '__package__', '__path__', '__ver
sion__', 'absolute', 'absolute_import', 'add', 'add_docstring', 'add_newdoc', 'a
dd_newdoc_ufunc', 'add_newdocs', 'alen', 'all', 'allclose', 'alltrue', 'alterdot
s', 'amax', 'amin', 'angle', 'any', 'append', 'apply_along_axis', 'apply_over_axe
s', 'arange', 'arccos', 'arccosh', 'arcsin', 'arcsinh', 'arctan', 'arctan2', 'ar
ctanh', 'argmax', 'argmin', 'argsort', 'argwhere', 'around', 'array', 'array2str
ing', 'array_equal', 'array_equiv', 'array_repr', 'array_split', 'array_str', 'a
```

More information about a specific function

After issuing `dir(np)`, we find the `std()` function among many others. To find more information about this specific function, we use `help(np.std)`. The following screenshot shows only a few lines of code for brevity:

```
>>> import numpy as np
>>> help(np.std)
Help on function std in module numpy.core.fromnumeric:

std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)
    Compute the standard deviation along the specified axis.

    Returns the standard deviation, a measure of the spread of a distribution
    of the array elements. The standard deviation is computed for the
    flattened array by default, otherwise over the specified axis.

    Parameters
    -----
    a : array_like
        Calculate the standard deviation of these values.
    axis : int, optional
        Axis along which the standard deviation is computed. The default is
        to compute the standard deviation of the flattened array.
```

Understanding the list data type

In *Chapter 3, Using Python as a Financial Calculator*, tuples are introduced as one of the data types. Recall that a tuple is defined by using parentheses such as `x=(1,2,3,"Hello")`. In addition, after a tuple is defined, we cannot change its values. Like tuples, the lists data type could contain different types of data, and their first subscripts start from 0. The following Python commands generate a list for variable `x`:

```
>>>x=[1,2,"John", "M", "Student"]
>>>type(x)
<class 'list'>
```

From the preceding code statements, we know that a set of variables included will be closed by a pair of square brackets, that is []. To call specific data item(s), we could use different ways to achieve our goals. The following commands show how to pick up individual data items for different goals:

```
>>>x
[1, 2, 'John', 'M', 'Student']
>>>x[1]
2
>>>x[2:]
['John', 'M', 'Student']
```

Unlike tuples, we can modify the values of a list.

Working with arrays of ones, zeros, and the identity matrix

In the following code examples, we don't show values of all the variables, from a to i, to save space:

```
>>>import numpy as np
>>>a=np.zeros(10)                # array with 10 zeros
>>>b=np.zeros((3,2),dtype=float) # 3 by 2 with zeros
>>>c=np.ones((4,3),float)        # 4 by 3 with all ones
>>>d=np.array(range(10),float)   # 0,1, 2,3 .. up to 9
>>>e1=np.identity(4)            # identity 4 by 4 matrix
>>>e2=np.eyes(4)                # same as above
>>>e3=np.eyes(4,k=1)            # 1 start from k
>>>f=np.arange(1,20,3,float)    # from 1 to 19 interval 3
>>>g=np.array([[2,2,2],[3,3,3]]) # 2 by 3
>>>h=np.zeros_like(g)          # all zeros
>>>i=np.ones_like(g)           # all ones
```

Performing array manipulations

In finance-related research, quite often we need to change the dimensions of a matrix or an array. For example, converting a set of 100 random numbers into a 20 by 5 matrix or vice versa. For this purpose, we could use two NumPy functions, `flatten()` and `reshape()`, as follows:

```
>>>pv=np.array([[100,10,10.2],[34,22,34]]) # 2 by 3
>>>x=pv.flatten() # matrix becomes a vector
>>>vp2=np.reshape(x,[3,2]) # 3 by 2 now
```

Performing array operations with +, -, *, /

Plus and minus for an array would have their normal meaning. However, multiplication and division have quite different definitions. Using multiplication as an example, $A \times B$ arrays could have two meanings: either item by item (A and B should have the same dimensions, that is, both are n by m) or matrix multiplication (the second dimension of A should be the same as the first dimension of B , that is, A is n by m while B is m by p).

Performing plus and minus operations

When adding or subtracting two arrays, they must have the same dimensions, that is, both are n by m . If they have different dimensions, we will get an error message. The following example shows the summation of two cash flow arrays:

```
>>>cashFlows_1=np.array([-100,50,20])
>>>cashFlows_2=np.array([-80,100,120])
>>>cashFlows_1 + cashFlows_2
>>>array([-180, 150, 140])
```

Performing a matrix multiplication operation

For matrix multiplication, matrices A and B should be n by k and k by m . Assume that matrix A is n by k and that B is k by m as shown in the following formula:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{k1} & \cdots & b_{km} \end{pmatrix} \quad (5)$$

Our final matrix will have the dimensions n by m as shown in the following formula:

$$A * B = C = \begin{pmatrix} c_{11} & \cdots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nm} \end{pmatrix} \quad (6)$$

The individual data item, c_{ij} , in matrix C will have the following operation:

$$c_{ij} = \sum_{i=1}^n \sum_{j=1}^m \sum_{t=1}^k a_{i,t} b_{t,j}$$

Assume that we have two matrices/arrays x (n by k) and y (k by m); the dot product will generate a matrix with n by m as shown in the following lines of code:

```
>>>x=np.array([[1,2,3],[4,5,6]],float)    # 2 by 3
>>>y=np.array([[1,2],[3,3],[4,5]],float)  # 3 by 2
>>>np.dot(x,y)                             # 2 by 2
Array([[19., 23.],
       [43., 53.]])
```

Alternatively, we could convert arrays into matrices first and then use $*$ of matrix multiplication as shown in the following lines of code:

```
>>>x=np.matrix("1,2,3;4,5,6")
>>>y=np.matrix("1,2;3,3;4,5")
>>>x*y
Array([[19., 23.], [43., 53.]])
```

Actually, we could convert an array into a matrix easily as follows:

```
>>>x1=np.array([[1,2,3],[4,5,6]],float)
>>>x2=np.matrix(x1)    # from array to matrix
>>>x3=np.array(x2)     # from matrix to array
```

Performing an item-by-item multiplication operation

When two arrays have the same dimensions, the product of $x*y$ will give an item for item multiplication. When both A and B have the same dimensions, n by m , their item by item multiplication will have the following form:

$$c_{ij} = \sum_{i=1}^n \sum_{j=1}^m a_{ij} b_{ij}$$

The following lines of code are an example of this:

```
>>>x=np.array([[1,2,3],[4,5,6]],float)
>>>y=np.array([[2,1,2],[4,0,5]],float)
>>>x*y
Array([[2.,  2.,  6.,  ],
       [16.,  0.,  30. ]])
```

The x.sum() dot function

After x is defined as a NumPy array, we could use $x.function()$ to conduct related operations such as $x.sum()$ as shown in the following lines of code:

```
>>>import numpy as np
>>>x=np.array([1,2,3])
>>>x.sum()
6
>>>np.sum(x)
6
```

If x is a NumPy array, we could have other functions with the same dot format as well: $x.mean()$, $x.min()$, $x.max()$, $x.var()$, $x.argmax()$, $x.clip()$, $x.copy()$, $x.diagonal()$, $x.reshape()$, $x.tolist()$, $x.fill()$, $x.transpose()$, $x.flatten()$, and $x.argmax()$. Those dot functions are useful because of the convenience they offer. The following commands show two such examples:

```
>>>cashFlows=np.array([-100,30,50,100,30,40])
>>>np.min(cashFlows)
```

```
-100
>>>np.argmax(cashFlows)
0
```

The `np.min()` function shows the minimum value, while the `np.argmin()` function gives the location (that is, index) of the minimum value.

Looping through an array

We could iterate over an array. The following code example prints each cash flow:

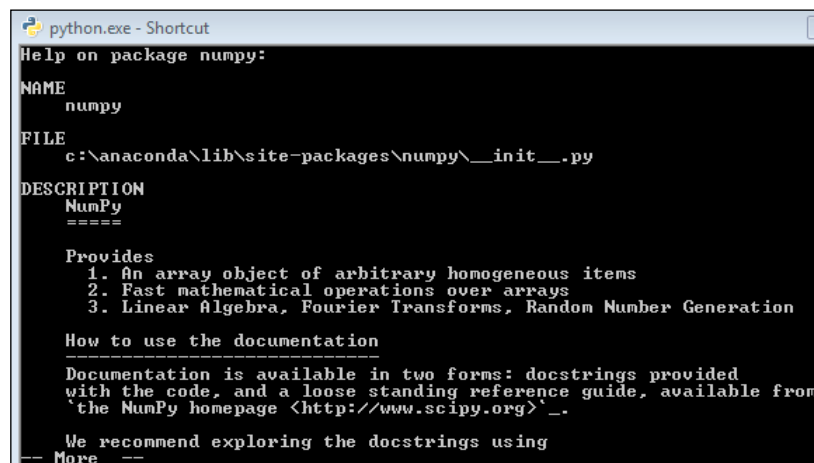
```
>>>import numpy as np
>>>cash_flows=np.array([-100,50,40,30,25,-10,50,100])
>>>for cash in cash_flows:
    print x
```

Using the help function related to modules

We could use the `help()` function to find out more information about NumPy and SciPy as shown in the following lines of code:

```
>>>help()
help>numpy
```

The first few lines are shown in the following screenshot:



Similarly, we issue `scipy` in the help window as follows:

```
>>>help()
help> scipy      # to save space, the output is not shown
```

A list of subpackages for SciPy

SciPy has about a dozen subpackages. The following table shows a list of subpackages contained in SciPy:

Subpackage	Description
Cluster	Clustering algorithms
Constants	Physical and mathematical constants
Fftpack	Fast Fourier Transform routines
Integrate	Integration and ordinary differential equation solvers
Interpolate	Interpolation and smoothing splines
Io	Input and output
Linalg	Linear algebra
Ndimage	N-dimensional image processing
Odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions
weave	C

Cumulative standard normal distribution

In *Chapter 4, 13 Lines of Python to Price a Call Option*, we used 13 lines of Python codes to price a call option since we have to write our own cumulative standard normal distribution. Fortunately, the cumulative standard normal distribution is included in the submodule of SciPy. The following example shows the value of the cumulative standard normal distribution at zero:

```
>>>from scipy.stats import norm
>>>norm.cdf(0)
0.5
```


Thus, we could simplify our call option model considerably using just five lines. The following code is a typical example of the benefits we can enjoy using various modules:

```
from scipy import log,exp,sqrt,stats
defbs_call(S,X,T,r,sigma):
    d1=(log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    d2 = d1-sigma*sqrt(T)
    return S*stats.norm.cdf(d1)-X*exp(-r*T)*stats.norm.cdf(d2)
```

Now, we could use the following function by inputting a set of values:

```
>>>price=bs_call(40,40,1,0.03,0.2)
>>>round(price,2)
3.77
```

Logic relationships related to an array

An array could contain `true` and `false` as shown in the following lines of code. This data type is called Boolean.

```
>>>import numpy as np
>>>x=np.array([True,False,True,False],bool)
>>>a=any(x)    # if one item is TRUE then return TRUE
>>>b=all(x)    # if all are TRUE then return TRUE
>>>cashFlows=np.array([-100,50,40,30,100,-5])
>>>a=cashFlows>0    # [False,True,True,True,True,False]
>>>np.logical_and(cashFlows>0, cashFlows<60)
Array([False,True,True,False,False],dtype=bool)
```

The `logical_and()`, `logical_or()`, and `logical_not()` functions could be used to compare each data item included in an array as shown in the previous code example. In addition, we could save the index or subscripts of the logic comparison and call the array later as shown in the following lines of code:

```
>>>cashFlows=np.array([-100,50,40,30,100,-5])
>>>index=(cashFlows>0) # index is a Boolean variable
>>>cashFlows[index]    # retrieve positive cash flows
Array([50., 40., 30., 100. ])
```

Statistic submodule (stats) from SciPy

One special module called `stats` contained in the `SciPy` module is worthy of special attention, since many of our financial problems depend on this module. To find out all the contained functions, we have the following lines of code:

```
>>>from scipy import stats
>>>dir(stats)
```

To save space, only a few lines are shown in the following screenshot:

```
>>> from scipy import stats
>>> dir(stats)
['_Tester', '_all_', '_builtins_', '_doc_', '_file_', '_name_', '_pack
age_', '_path_', 'binned_statistic', 'rank', 'support', 'tukeylambda_stat
s', 'absolute_import', 'alpha', 'anderson', 'anglit', 'ansari', 'arcsine', 'bart
lett', 'bayes_mvs', 'hernoulli', 'beta', 'betai', 'betaprime', 'binned_statistic
', 'binned_statistic_2d', 'binned_statistic_dd', 'binom', 'binom_test', 'boltzma
nn', 'boxcox', 'boxcox_1lf', 'boxcox_normmax', 'boxcox_normplot', 'bradford', 'b
urr', 'callable', 'cauchy', 'chi', 'chi2', 'chi2_contingency', 'chisqprob', 'chi
square', 'circmean', 'circstd', 'circvar', 'cmedian', 'contingency', 'cosine', 'chi
cumfreq', 'describe', 'dgamma', 'distributions', 'division', 'dlaplace', 'dweibu
ll', 'entropy', 'erlang', 'expon', 'exponpow', 'exponweib', 'f', 'f_oneway', 'f_
value', 'f_value_multivariate', 'f_value_wilks_lambda', 'fastsort', 'fatiguelife
', 'find_repeats', 'fisher_exact', 'fisk', 'fligner', 'foldcauchy', 'foldnorm',
', 'fprob', 'frechet_l', 'frechet_r', 'friedmanchisquare', 'futil', 'gamma', 'gauss
hyper', 'gaussian_kde', 'genexpon', 'genextreme', 'gengamma', 'genhalflogistic',
', 'genlogistic', 'genpareto', 'geom', 'gilbrat', 'glm', 'gmean', 'gompertz', 'gum
bel_l', 'gumbel_r', 'halfcauchy', 'halflogistic', 'halfnorm', 'histogram', 'hist
```

From the output, not shown completely in the preceding screenshot, we could find a `ttest_1samp()` function. To use the function, we generate 100 random numbers drawn from a standard normal distribution, zero mean, and unit standard deviation. For the one-sample T-test, we test whether its mean is 0. Based on the t-value (1.18) and p-value (0.24), we could not reject the null hypothesis. This means that the mean of our x is zero as shown in the following lines of code:

```
>>>import numpy as np
>>>from scipy import stats
>>>np.random.seed(124) # get the same random values
>>>x=np.random.normal(0,1,100) # mean=0,std=1
>>>skew=stats.skew(x) # skewness is -0.2297
>>>stats.ttest_1samp(x,0) # if the mean is zero
(array(1.176), 0.24228) # T-value and P-value
```

From NumPy, we could draw random numbers from various distributions; see a few more examples:

```
>>>import numpy as np
>>>s=np.random.standard_t(10, size=1000) # from standard-T,df=10
```

```
>>>x=np.random.uniform(low=0.0,high=1.0,size=100) # uniform
>>>stocks=np.random.random_integers(1,500,20)
>>>stocks
array([371,  15, 158, 468, 299, 470, 257, 481,  76, 196, 355, 386, 438,
      484,  41,  39, 222, 377, 455,  46])
```

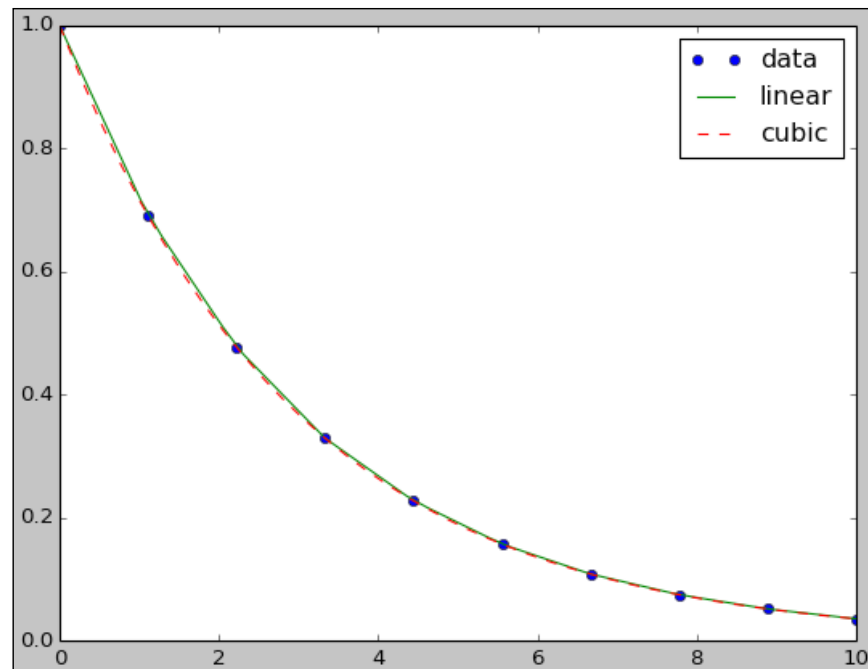
The two commands pick up 20 stocks randomly from 500 available stocks.

Interpolation in SciPy

In the following code example, x can be viewed as the x axis with a set of values from 0 to 10, while the vertical axis is y , where $y = \exp(-x/3)$. We intend to interpolate between different $y(i)$ values by applying two methods: linear and cubic. The following lines of code are an example from *SciPy Reference Guide*:

```
>>>import numpy as np
>>>import matplotlib.pyplot as plt
>>>from scipy.interpolate import interp1d
>>>x = np.linspace(0, 10, 10)
>>>y = np.exp(-x/3.0)
>>>f = interp1d(x, y)
>>>f2 = interp1d(x, y, kind='cubic')
>>>xnew = np.linspace(0, 10, 40)
>>>plt.plot(x,y,'o',xnew,f(xnew),'-', xnew, f2(xnew),'--')
>>>plt.legend(['data', 'linear', 'cubic'], loc='best')
>>>plt.show()
```

In the preceding program, we use the `np.linspace()` function to generate evenly spaced numbers—40 values—over a specified interval, from 0 to 10 in this case. The related output is shown as follows:



Solving linear equations using SciPy

Assume that we have the following three equations:

$$\begin{cases} x + 2y + 5z = 10 \\ 2x + 5y + z = 8 \\ 2x + 3y + 8z = 5 \end{cases} \quad (7)$$

We define A and B as follows:

$$A = \begin{pmatrix} 1 & 2 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{pmatrix} \quad B = \begin{pmatrix} 10 \\ 8 \\ 5 \end{pmatrix} \quad (8)$$

The solution is as follows:

$$A = \begin{pmatrix} 1 & 2 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{pmatrix} \quad B = \begin{pmatrix} 10 \\ 8 \\ 5 \end{pmatrix} \quad (9)$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = A^{-1} * b \quad (10)$$

The related Python code is as follows:

```
>>>import scipy as sp
>>>import numpy as np
>>>A=sp.mat('[1 2 5; 2 5 1; 2 3 8]')
>>>b = sp.mat('[10;8;5]')
>>>A.I*b
Matrix([-22.45, 10.09, 2.45])
>>>np.linalg.solve(A,b) # offer the same solution
```

Generating random numbers with a seed

One of the major assumptions about option theory is that stock prices follow a log-normal distribution and returns follow a normal distribution. The following lines of code show an example of this:

```
>>>importscipy as sp
>>>x=sp.random.rand(10) # 10 random numbers from [0,1)
>>>y=sp.random.rand(5,2) # random numbers 5 by 2 array
>>>z=sp.random.rand.norm(100) from a standard normal
>>>
```

After issuing the preceding function, the software would pick up a set of random numbers depending on a user's computer time. However, sometimes we need a fixed set of random numbers, and this is especially true when testing our models and code, and for teaching. To satisfy this need, we will have to set up the seed value before generating our random numbers, as shown in the following lines of code:

```
>>>importscipy as sp
>>>sp.random.seed(12456)
>>>sp.random.rand(5)
[0.92961609, 0.3163755, 0.18391881, 0.20456028]
```

If we want to generate the exact same random numbers, we have to use the same seed before we call the `random.rand()` function.

There are about two dozen distributions available, such as beta, binomial, chisquare, exponential, f, gamma, geometric, lognormal, poisson, uniform, and weibull. After issuing `help(np.random)`, we get the following output a (part of the all output):

```
Univariate distributions
=====
beta          Beta distribution over ``[0, 1]``.
binomial      Binomial distribution.
chisquare     :math:`\chi^2` distribution.
exponential   Exponential distribution.
f            F (Fisher-Snedecor) distribution.
gamma        Gamma distribution.
geometric     Geometric distribution.
gumbel        Gumbel distribution.
hypergeometric Hypergeometric distribution.
laplace       Laplace distribution.
logistic      Logistic distribution.
lognormal     Log-normal distribution.
logseries     Logarithmic series distribution.
negative_binomial Negative binomial distribution.
noncentral_chisquare Non-central chi-square distribution.
noncentral_f  Non-central F distribution.
normal        Normal / Gaussian distribution.
pareto        Pareto distribution.
poisson       Poisson distribution.
power         Power distribution.
rayleigh      Rayleigh distribution.
triangular    Triangular distribution.
uniform       Uniform distribution.
vonmises      Von Mises circular distribution.
wald          Wald (inverse Gaussian) distribution.
weibull       Weibull distribution.
zipf          Zipf's distribution over ranked data.
=====
```

Finding a function from an imported module

We could assign all related functions from NumPy to a variable such as `x`. Then, we loop through it to print each individual function as shown in the following lines of code:

```
>>>import numpy as np
>>>x=np.array(dir(np))
>>>for k in x:
    if (k.find("uni")!=-1):
        print k

unicode
unicode0
unicode_
union1d
unique
```

Understanding optimization

In finance, many issues depend on optimization, such as choosing an optimal portfolio with an objective function and with a set of constraints. For those cases, we could use a SciPy optimization module called `scipy.optimize`. Assume that we want to estimate the x value that minimizes the value of y , where $y = 3 + x^2$. Obviously, the minimum value of y is achieved when x takes a value of 0.

```
>>>import scipy.optimize as optimize
>>>def my_f(x):
    Return 3 + x**2
>>>optimize.fmin(my_f,5)    # 5 is initial value

Optimization terminated successfully
Current function values: 3:000000
Iterations: 20
Function evaluations: 40
Array([ 0. ])
```

To find a list of all input variables to this `fmin()` function and their meanings, issue `help(optimize.fmin)`. To list all the functions included in `scipy.optimize`, issue `dir(optimize)`.

Linear regression and Capital Assets Pricing Model (CAPM)

According to the famous CAPM, the returns of a stock are linearly correlated with its market returns. Usually, we consider the relationship of the excess stock returns versus the excess market returns.

$$R_i - R_f = a + \beta_i (R_{mkt} - R_f) \quad (11)$$

Here R_i is the stock i 's return; β_i is the slope (market risk); R_{mkt} is the market return and R_f is the risk-free rate. Eventually, the preceding equation could be rewritten as follows:

$$y = \alpha + \beta * x \quad (12)$$

The following lines of code are an example of this:

```
>>>from scipy import stats
>>>stock_ret = [0.065, 0.0265, -0.0593, -0.001,0.0346]
>>>mkt_ret = [0.055, -0.09, -0.041,0.045,0.022]
>>>beta, alpha, r_value, p_value, std_err =
stats.linregress(stock_ret,mkt_ret)
>>>print beta, alpha
0.507743187877 -0.00848190035246
>>>print "R-squared=", r_value**2
R-squared =0.147885662966
>>>print "p-value =", p_value
0.522715523909
```


Retrieving data from an external text file

When retrieving data from an external data file, the variable generated will be a list.

```
>>>f=open("c:\\data\\ibm.csv","r")
>>>data=f.readlines()
>>>type(data)
<class 'list'>
```

The first few lines of the input file are shown in the following lines of code. In *Chapter 7, Visual Finance via Matplotlib*, we will discuss how to download this input file from Yahoo! Finance.

```
>>>Date,Open,High,Low,Close,Volume,Adj Close
2013-07-26,196.59,197.37,195.00,197.35,2485100,197.35
2013-07-25,196.30,197.83,195.66,197.22,3014300,197.22
2013-07-24,195.95,197.30,195.86,196.61,2957900,196.61
2013-07-23,194.21,196.43,194.10,194.98,2863800,194.98
2013-07-22,193.40,195.79,193.28,194.09,3398000,194.09
2013-07-19,197.91,197.99,193.24,193.54,6997600,193.54
```

After we generate a variable called `stock`, we can view its first two observations as follows:

```
>>>data[1]
'2013-07-26,196.59,197.37,195.00,197.35,2485100,197.35\n'
>>>data[2]
'2013-07-25,196.30,197.83,195.66,197.22,3014300,197.22\n'
```

The `loadtxt()` and `genfromtxt()` functions

The `loadtxt()` function included in the NumPy module could be used to input a text or a CSV file as shown in the following lines of code:

```
>>>import numpy as ny
>>>ny.loadtxt("c:/data/ibm.csv",delimiter=',')
```

The function `genfromtxt()` in the NumPy module is more powerful but is a slower function to input data. The advantage of this function compared with `loadtxt()` is that the former treats non-standard values, such as 3.5 percent, as NA (Python missing code).

Installing NumPy independently

To install NumPy independently, we have to perform the following two steps:

1. Go to <http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy>.
2. Choose an appropriate package to download and install, such as `numpy-MKL-1.7.1.win32-py3.3.exe`.

Understanding the data types

In the following table, most of the types of data are given:

Data type	Description
<code>Bool</code>	Boolean (True or False) stored as a byte
<code>int</code>	Platform integer (normally either <code>int32</code> or <code>int64</code>)
<code>int8</code>	Byte (-128 to 127)
<code>int16</code>	Integer (-32768 to 32767)
<code>int32</code>	Integer (-2147483648 to 2147483647)
<code>int64</code>	Integer (9223372036854775808 to 9223372036854775807)
<code>uint8</code>	Unsigned integer (0 to 255)
<code>uint16</code>	Unsigned integer (0 to 65535)
<code>uint32</code>	Unsigned integer (0 to 4294967295)
<code>uint64</code>	Unsigned integer (0 to 18446744073709551615)
<code>float</code>	Short and for <code>float64</code>
<code>float32</code>	Single precision float: sign bit 23 bits mantissa; 8 bits exponent
<code>float64</code>	52 bits mantissa
<code>complex</code>	Shorthand for <code>complex128</code>
<code>complex64</code>	Complex number; represented by two 32-bit floats (real and imaginary components)
<code>complex128</code>	Complex number; represented by two 64-bit floats (real and imaginary components)

Summary

In this chapter, we introduced the two most important modules, called NumPy and SciPy, which are used intensively for scientific and financial computation. NumPy is for numerical methodology, and SciPy could be viewed as an extension of NumPy. In this book, many chapters depend on these two modules. In addition, many other modules depend on these two modules. For instance, the Matplotlib (for graph) module, which will be discussed in *Chapter 7, Visual Finance via Matplotlib*, and Statsmodels (for statistical/financial modeling), which will be discussed in *Chapter 8, Statistic Analysis of Time Series*, depend on these two modules.

It is a very useful tool for visualization. We are going to use this module intensively when we explain option theory.

Exercises

1. What is module dependency?
2. Why is it difficult to install NumPy independently?
3. What are the advantages and disadvantages of writing a module that depends on other modules?
4. What are the advantages of using a super package to install many modules simultaneously?
5. How do we find all the functions contained in NumPy and SciPy?
6. What is wrong with the following operation?

```
>>>x=[1,2,3]
>>>x.sum()
```

7. How can we print all the data items for a given array?
8. What is wrong with the following lines of code?

```
>>>import np
>>>x=np.array([True,false,true,false],bool)
```

9. How can we iterate through an array?

10. Write a Python program to price a European call option using the cumulative standard normal distribution included in the `SciPy` module. Compare your result with the code in *Chapter 4, 13 Lines of Python to Price a Call Option*.
11. Find out the meaning of `skewtest` included in the `stats` submodule (`SciPy`), and give an example of using this function.
12. How do we find all the functions of `SciPy` and `NumPy`?
13. We have the following simultaneous equations. 'What are the values of x , y , and z ?

$$\begin{cases} 2x - y + 2.5z = 2.3 \\ x + 3.4y - z = 4.2 \\ -x + 2.9y + 1.8z = 3.1 \end{cases}$$

14. Debug the following lines of code, which are used to estimate a geometric mean for a given set of returns:

```
>>>import scipy as sp
>>>ret=np.array([0.05,0.11,-0.03])
>>>pow(np.prod(ret+1),1/len(ret))-1
```

15. Write a Python program to estimate both arithmetic and geometric means for a given set of returns.
16. In finance, we use the standard deviation of returns to measure the risk level of a security or portfolio. Based on the latest five year daily prices from Yahoo! Finance, what is the total risk for IBM? Note that we use the following formula to annualize a volatility (variance) based on daily returns:

$$\sigma_{annual}^2 = 252\sigma_{daily}^2$$

17. Find out the meaning of `zscore()` included in the `stats` submodule (`SciPy`), and offer a simple example of using this function.
18. What is the market risk (beta) for IBM in 2010? (Hint: the source of data could be from Yahoo! Finance.)
19. What is wrong with the following lines of code?

```
>>>c=20
>>>npv=np.npv(0.1,c)
```

20. The correlation coefficient function from NumPy is `np.corrcoef()`. Find more about this function. Estimate the correlation coefficient between IBM, DELL, and W-Mart.
21. Why is it claimed that the `sn.npv()` function from `SciPY()` is really a Present Value (PV) function?
22. Design a true NPV function using all cash flows, including today's cash flow.
23. The Sharpe ratio is used to measure the trade-off between risk and return:

$$Sharpe = \frac{\bar{R} - \bar{R}_f}{\sigma}$$

Here, $\bar{R}$ is the expected returns for an individual security, and $\bar{R}_f$ is the expected risk-free rate. σ is the volatility, that is, standard deviation of the return on the underlying security. Estimate Sharpe ratios for IBM, DELL, Citi, and W-Mart by using their latest five-year monthly data.

7

Visual Finance via Matplotlib

Graphs and other visual representations have become more important in explaining many complex financial concepts, trading strategies, and formulae. In this chapter, we discuss the module `matplotlib`, which is used to create various types of graphs. In addition, the module will be used intensively in *Chapter 9, The Black-Scholes-Merton Option Model*, when we discuss the famous Black-Scholes-Merton option model and various trading strategies. The `matplotlib` module is designed to produce publication-quality figures and graphs. The `matplotlib` module depends on `NumPy` and `SciPy` which were discussed in *Chapter 6, Introduction to NumPy and SciPy*. There are several output formats, such as PDF, Postscript, SVG, and PNG.

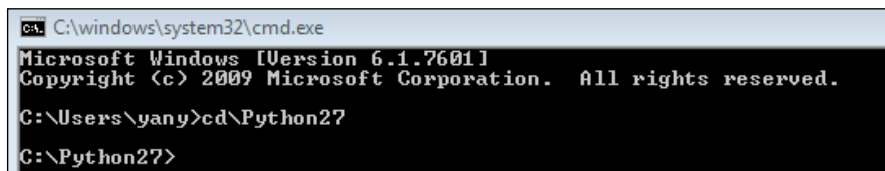
In particular, we will cover the following:

- Several ways to install `matplotlib`
- Simple examples of using `matplotlib`
- **Net Present Value (NPV)** profile, DuPont identity, stock returns, and histogram
- Total risk, market risk (beta), and firm-specific risk
- Stock co-movement and correlation
- Portfolio diversification
- Presentation of trading volume and price movement
- Return versus risk graph with several stocks
- Very complex examples of using `matplotlib`

Installing matplotlib via ActivePython

The first way to install the matplotlib module is via ActivePython. We install ActivePython first and then install matplotlib. In the process of installing matplotlib, NumPy would be installed as well since matplotlib depends on both NumPy and SciPy. The whole procedure has four steps as follows:

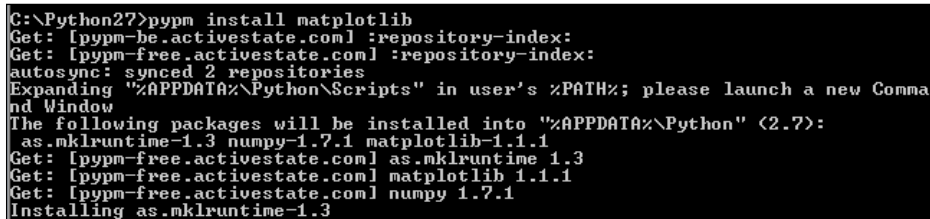
1. Go to <http://www.activestate.com/activepython/downloads>.
2. Choose an appropriate executable file to download.
3. For Windows, navigate to **All Programs | Accessories**, and then click on **Command Prompt**. You will see the following window:



```
C:\windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

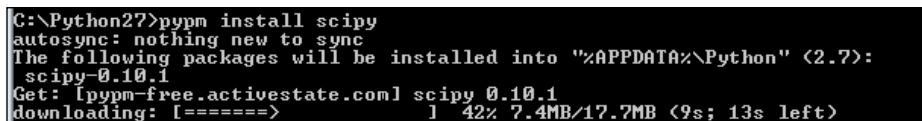
C:\Users\yany>cd\Python27
C:\Python27>
```

4. After going to the appropriate directory, such as C:\Python27, type `pypm install matplotlib` as shown in the following screenshot:



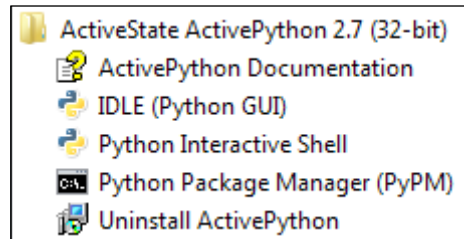
```
C:\Python27>pypm install matplotlib
Get: [pypm-be.activestate.com] :repository-index:
Get: [pypm-free.activestate.com] :repository-index:
autosync: synced 2 repositories
Expanding "%APPDATA%\Python\Scripts" in user's %PATH%; please launch a new Command Window
The following packages will be installed into "%APPDATA%\Python" (2.7):
  as.mklruntime-1.3 numpy-1.7.1 matplotlib-1.1.1
Get: [pypm-free.activestate.com] as.mklruntime 1.3
Get: [pypm-free.activestate.com] matplotlib 1.1.1
Get: [pypm-free.activestate.com] numpy 1.7.1
Installing as.mklruntime-1.3
```

The matplotlib module depends on both NumPy and SciPy. Since the NumPy module will be installed automatically when we install matplotlib, we need to install SciPy; see the following similar procedure:



```
C:\Python27>pypm install scipy
autosync: nothing new to sync
The following packages will be installed into "%APPDATA%\Python" (2.7):
  scipy-0.10.1
Get: [pypm-free.activestate.com] scipy 0.10.1
downloading: [=====] 1 42% 7.4MB/17.7MB (9s; 13s left)
```

To launch Python, navigate to **All Programs | ActivateStateActive Python**, and then click on **IDLE (Python GUI)**. For convenience, we could generate a shortcut on our desktop as shown in the following screenshot:



Alternative installation via Anaconda

In *Chapter 6, Introduction to NumPy and SciPy*, we discussed the dependency of a module. Because of such a dependency, it might be very difficult to install a module independently since it depends on many other modules. In this book, we use the so-called super-packages. If one of them is installed, most of our modules are installed simultaneously. We choose Anaconda. To install Anaconda, we have the following two steps:

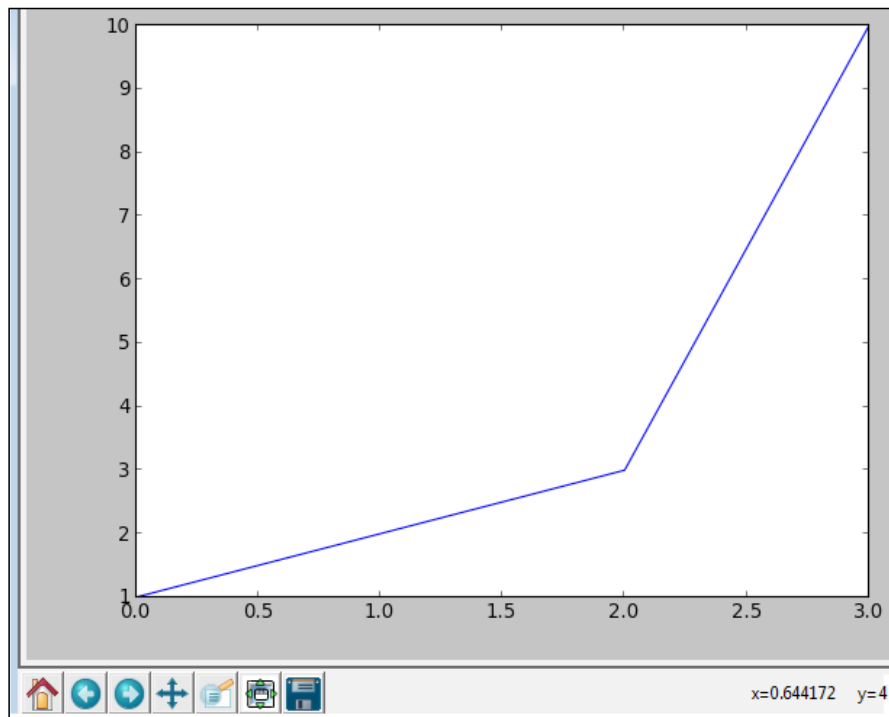
1. Go to <http://continuum.io/downloads>.
2. Choose an appropriate package to download and install.

Understanding how to use matplotlib

The best way to understand the usage of the `matplotlib` module is through examples. The following example could be the simplest one since it has just three lines of Python code. The objective is to link several points. By default, the `matplotlib` module assumes that the *x* axis starts at zero and moves by one on every element of the array. The following command lines illustrate this situation:

```
>>>from matplotlib.pyplot import *
>>>plot([1,2,3,10])
>>>show()
```


After we press the *Enter* key after typing the last command of `show()`, the following graph will appear:

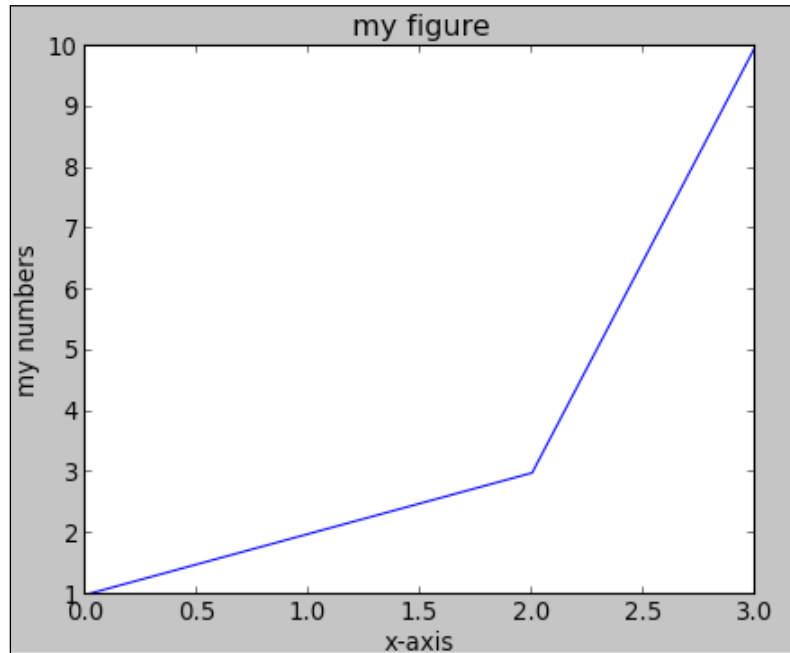


At the bottom of the graph, we can find a set of icons, and based on them, we could adjust our image and other functions, such as saving our image. After closing the preceding figure, we could return to Python prompt. On the other hand, if we issue `show()` the second time, nothing will happen. To repeat the preceding graph, we have to issue both `plot([1, 2, 3, 10])` and `show()`.

We could add labels for both the *x* axis and *y* axis as follows:

```
>>>from matplotlib.pyplot import *
>>>plot([1,2,3,10])
>>>xlabel("x- axis")
>>>ylabel("my numbers")
>>>title("my figure")
>>>show()
```

The corresponding graph is shown in the following screenshot:

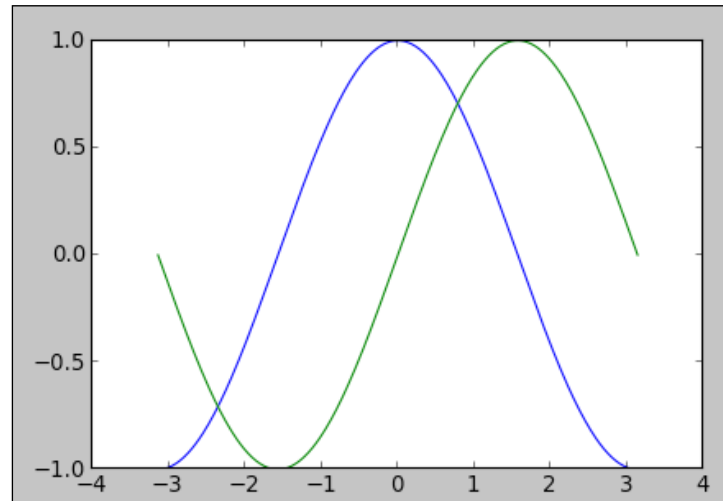


The next example presents two cosine functions:

```
>>>from pylab import *
>>>x = np.linspace(-np.pi, np.pi, 256,endpoint=True)
>>>C,S = np.cos(x), np.sin(x)
>>>plot(x,C),plot(x,S)
>>>show()
```

In the preceding code, the `linspace()` function has four input values: `start`, `stop`, `num`, and `endpoint`. In the preceding example, we start from `-3.1415916` and stop at `3.1415926`, with 256 values between. In addition, the endpoints will be included. By the way, the default value of `num` is 50.

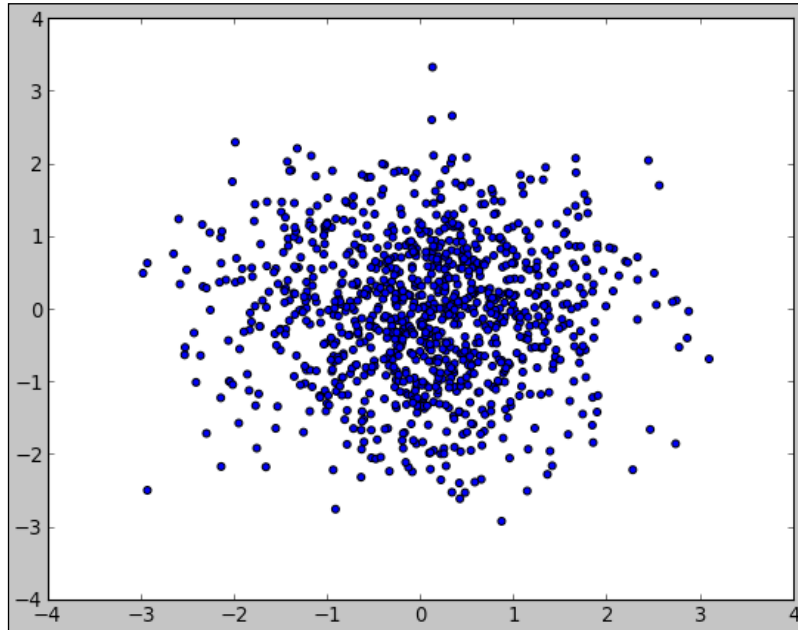
The corresponding graph is shown in the following screenshot:



The following example shows the scatter pattern. First, the `np.random.normal()` function is used to generate two sets of random numbers. Since `n` is 1024, we have 1,024 observations for both `x` and `y` variables. The key function is `scatter(x,y)` as follows:

```
>>>from pylab import *
>>>n = 1024
>>>X = np.random.normal(0,1,n)
>>>Y = np.random.normal(0,1,n)
>>>scatter(X,Y)
>>>show()
```

The corresponding output graph is given as follows:



We could check the scatter pattern to visually perceive the relationship between two stocks. For example, we have two time series of returns for stocks A and B. Assume that they are strongly, positively correlated, that is, stock A has a lower return of -1 percent and stock B has a quite similar low return. This is true for a higher return, such as 20 percent. The scatter points of their matched returns should be distributed along a 45-degree straight line.

Understanding simple and compounded interest rates

Many students and practitioners are confused with the difference between simple interest and compound interest. Simple interest does not consider interest on interest while compound interest does. It is a good idea to represent them with a graph. For instance, we borrow \$1,000 today for 10 years with an annual interest of 8 percent per year. What are the future values if 8 percent is the simple interest and compounded interest rate? The formula for payment of a simple interest rate is as follows:

$$FV(\text{simple interest}) = PV(1 + R * n) \quad (1)$$

The future value for compounded interest is as follows:

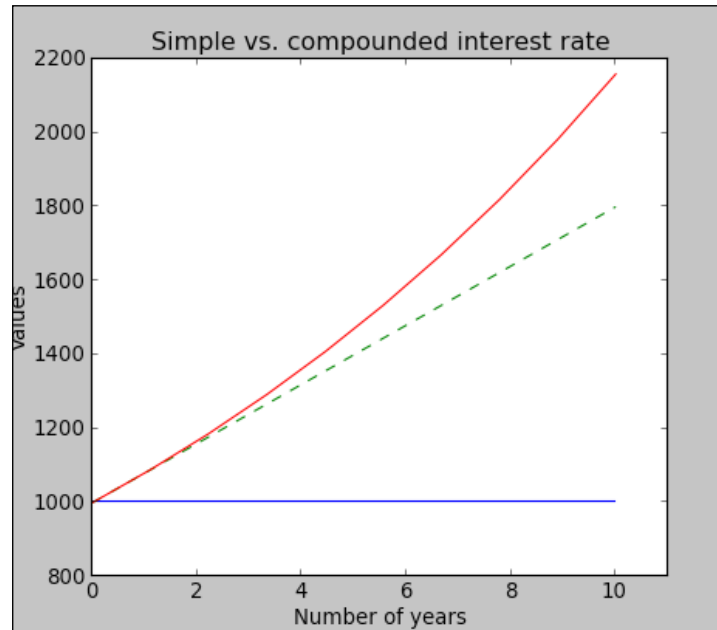
$$FV(\text{compounded interest}) = PV(1 + R)^n \quad (2)$$

Here, PV is the load we borrow today, that is, present value, R is the period rate, and n is the number of periods. Thus, those two future values will be \$1,800 and \$2,158.93. The following program offers a graphic representation of a principal, simple interest payment, and the future values:

```
import numpy as np
from matplotlib.pyplot import *
from pylab import *
pv=1000
r=0.08
n=10
t=linspace(0,n,n)
y1=np.ones(len(t))*pv # this is a horizontal line
y2=pv*(1+r*t)
y3=pv*(1+r)**t
title('Simple vs. compounded interest rates')
xlabel('Number of years')
ylabel('Values')
xlim(0,11)
ylim(800,2200)
plot(t, y1, 'b-')
plot(t, y2, 'g--')
plot(t, y3, 'r-')
show()
```

In the preceding program, the `xlim()` function would set the range of the x axis. This is true for the `ylim()` function. The third input variable for both the `xlim()` and `ylim()` functions is for color and for the line. The letter `b` is for black, `g` is for green, and `r` is for red.

The corresponding output graph for the previous code is given as follows:

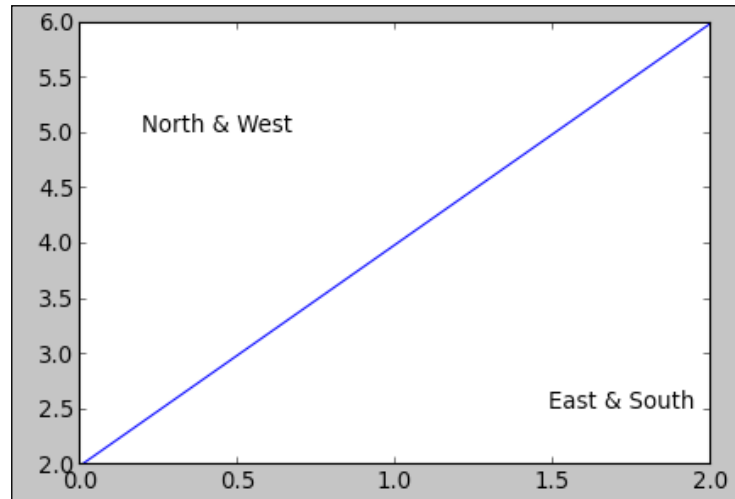


Adding texts to our graph

In the following example, we simply insert a text. Remember that the x and y scale is from 0 to 1:

```
>>>from pylab import *
>>>x = [0,1,2]
>>>y = [2,4,6]
>>>plot(x,y)
>>>figtext(0.2, 0.7, 'North & West')
>>>figtext(0.7, 0.2, 'East & South')
>>>show()
```

The corresponding output graph is given as follows:



Let's make it more complex. From the National Bureau of Economic Research web page at <http://www.nber.org/cycles.html>, we can find the following table showing the business cycle in the past two decades:

Turning Point Date	Peak or Trough	Announcement Date
June 1, 2009	Trough	September 20, 2010
December 1, 2007	Peak	December 1, 2008
November 1, 2001	Trough	July 17, 2003
March 1, 2001	Peak	November 26, 2001
March 1, 1991	Trough	December 22, 1992
July 1, 1990	Peak	April 25, 1991
November 1, 1982	Trough	July 8, 1983
July 1, 1981	Peak	June 1, 1982
July 1, 1980	Trough	July 8, 1981
January 1, 1980	Peak	June 3, 1980

Working with DuPont identity

In finance, we could find useful information from a firm's financial statements such as annual income statement, balance sheet, and cash flow statement. Ratio analysis is one of the commonly used tools to compare the performance among different firms and for the same firm over the years. DuPont identity is one of them. DuPont identity divides **Return on Equity (ROE)** into three ratios: Gross Profit Margin, Assets Turnover, and Equity Multiplier:

$$ROE = \frac{\text{Net Income}}{\text{Sales}} * \frac{\text{Sales}}{\text{Total Assets}} * \frac{\text{Total Assets}}{\text{Book value of Equity}} \quad (3)$$

The following code will show those three ratios with different colors. Here we have the following information about some firms:

Ticker	Fiscal Year Ending Date	ROE	Gross Profit Margin	Assets Turnover	Equity Multiplier
IBM	December 31, 2012	0.8804	0.1589	0.8766	6.3209
DELL	February 1, 2013	0.2221	0.0417	1.1977	4.4513
WMT	January 31, 2013	0.2227	0.0362	2.3099	2.6604

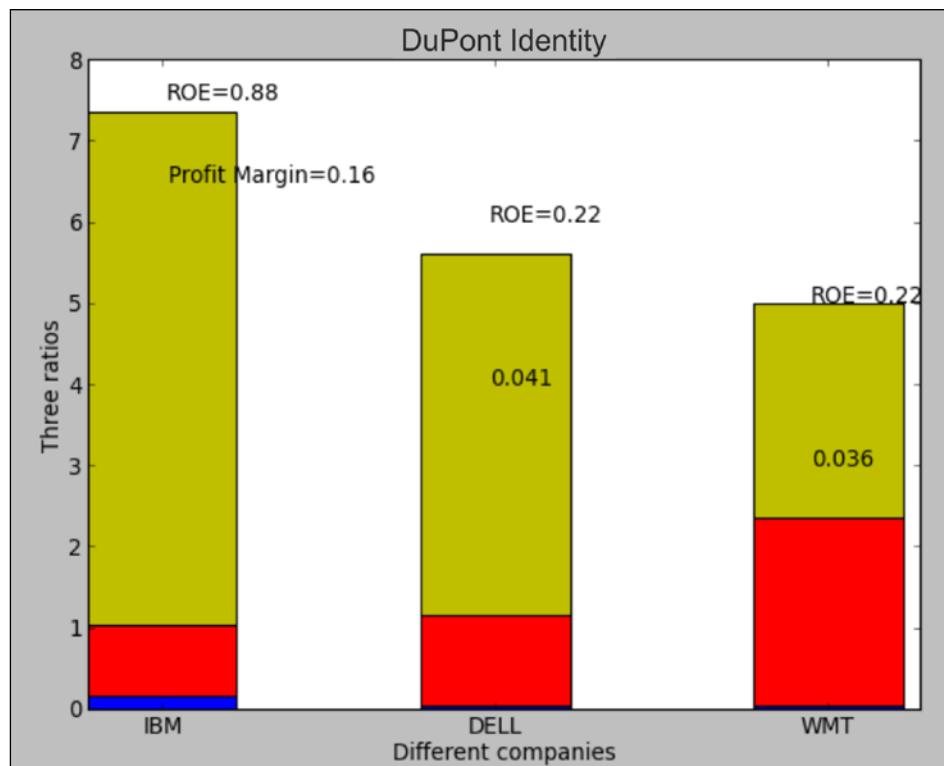
The Python code is as follows:

```
import numpy as np
import matplotlib.pyplot as plt
ind = np.arange(3)
plt.title("DuPont Identity")
plt.xlabel("Different companies")
plt.ylabel("Three ratios")
ROE=[0.88,0.22,0.22]
a = [0.16,0.04,0.036]
b = [0.88,1.12,2.31]
c = [6.32,4.45,2.66]
width = 0.45
plt.figtext(0.2,0.85,"ROE=0.88")
```



```
plt.figtext(0.5,0.7,"ROE=0.22")
plt.figtext(0.8,0.6,"ROE=0.22")
plt.figtext(0.2,0.75,"Profit Margin=0.16")
plt.figtext(0.5,0.5,"0.041")
plt.figtext(0.8,0.4,"0.036")
p1 = plt.bar(ind, a, width, color='b')
p2 = plt.bar(ind, b, width, color='r', bottom=a)
p3 = plt.bar(ind, c, width, color='y', bottom=[a[j] + b[j] for j in plt.
xticks(ind+width/2., ('IBM', 'DELL', 'WMT')) )
plt.show()
```

In the previous program, `plt.figtext(x,y,'text')` adds a text message at x-y location with x and y both having a range from 0 to 1. The `plt.bar()` function is used to generate three bars. The three bars are shown in the following figure:



Understanding the Net Present Value (NPV) profile

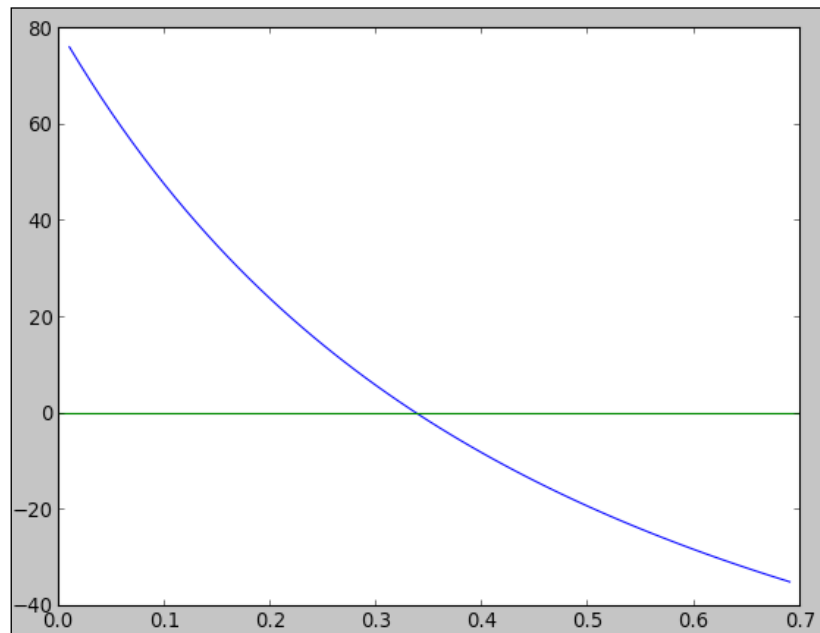
Gradually, we use graphs and other visual representations to explain many complex financial concepts, formulae, and trading strategies. An NPV profile is the relationship between a project's NPV and its discount rate (cost of capital). For a normal project, where cash outflows first then cash inflows, its NPV will be a decreasing function of the discount rate. The reason is that when the discount rate increases, the present value of the future cash flows (most time benefits) will decrease more than the current or the latest cash flows (most time costs). The NPV is defined by the following formula:

$$NPV = PV(\text{benefits}) - PV(\text{all costs})$$

The following program demonstrates a negative correlation between NPV and the discount rate:

```
>>>import scipy as sp
>>>from matplotlib.pyplot import *
>>>cashflows=[-100,50,60,70]
>>>rate=[]
>>>npv=[]
>>>x=(0,0.7)
>>>y=(0,0)
>>>for i in range(1,70):
    rate.append(0.01*i)
    npv.append(sp.npv(0.01*i,cashflows[1:])+cashflows[0])
>>>plot(rate,npv),plot(x,y)
>>>show()
```

In the preceding program, we plan to draw two lines: a straight line at $y=0$ and an NPV profile. The NPV profile indicates the relationship between NPV and discount rate as shown in the following graph:



In the previous example, we see just one Internal Rate of Return (IRR) defined as the discount rate makes NPV equal to zero. However, for abnormal projects, with cash inflows first and then cash outflows, or for the projects with more than one change in direction of cash flows, we could not tell whether we could have a unique **IRR**. This scenario is represented as follows:

```
>>>from import scipy as sp
>>>cashflows=[-100,50,60,70]
>>>rate=0.1
>>>npv=sp.npv(rate,cashflows[1:])+cashflows[0]
>>>round(npv,2)
47.62
```

As we discussed in the previous chapter, the NPV function from `scipy` mimics the Excel NPV function, and it is actually a PV function using the following program:

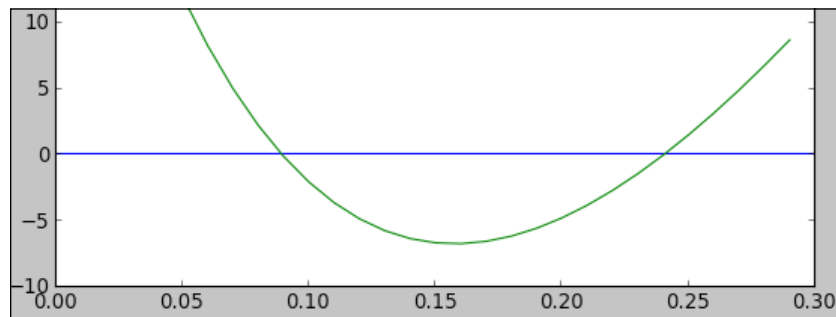
```
>>>import scipy as sp
>>>import matplotlib.pyplot as plt
>>>cashflows=[504,-432,-432,-432,832]
```

```

>>>rate=[]
>>>npv=[]
>>>x=[0,0.3]
>>>y=[0,0]
>>>for i in range(1,30):
    rate.append(0.01*i)
    npv.append(sp.npv(0.01*i,cashflows[1:])+cashflows[0])
>>>plt.plot(x,y),plt.plot(rate,npv)
>>>plt.show()

```

The output corresponding to this code is given as follows:



In the previous example, we know that there exist multiple IRRs. From the previous chapter, we know that we could use the `np.irr()` function to find out those multiple IRRs using the following program:

```

>>>import numpy as np
>>>cashflows=[504,-432,-432,-432,832]
>>>np.irr(cashflows)
array([ 0.08949087,  0.24006047])

```

Using colors effectively

To make our graphs or lines more eye-catching, we could use different colors. For example, we have 4 years' EPS (Diluted EPS Excluding Extraordinary Items) from Yahoo! Finance for the companies W-Mart and DELL. EPS is Earnings per Share. We could contrast their EPS with different colors using the following program:

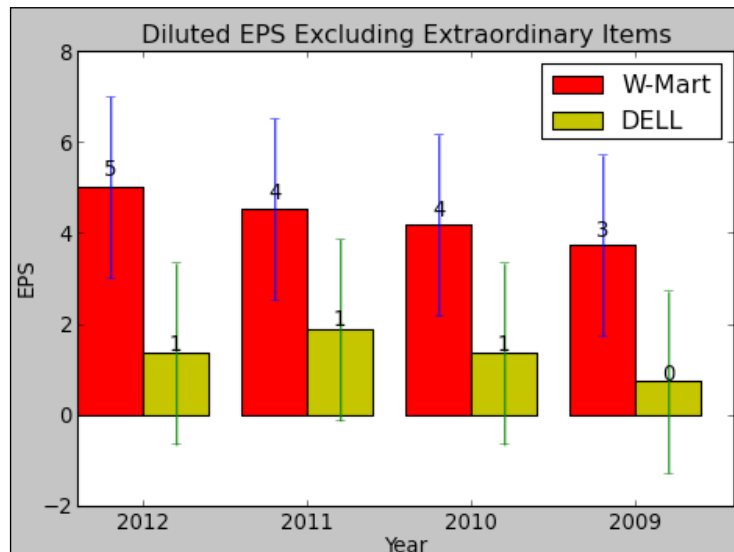
```

import matplotlib.pyplot as plt
A_EPS = (5.02, 4.54, 4.18, 3.73)
B_EPS = (1.35, 1.88, 1.35, 0.73)
ind = np.arange(len(A_EPS)) # the x locations for the groups

```

```
width = 0.40          # the width of the bars
fig, ax = plt.subplots()
A_Std=B_Std=(2,2,2,2)
rects1 = ax.bar(ind, A_EPS, width, color='r', yerr=A_Std)
rects2 = ax.bar(ind+width, B_EPS, width, color='y', yerr=B_Std)
ax.set_ylabel('EPS')
ax.set_xlabel('Year')
ax.set_title('Diluted EPS Excluding Extraordinary Items ')
ax.set_xticks(ind+width)
ax.set_xticklabels( ('2012', '2011', '2010', '2009') )
ax.legend( (rects1[0], rects2[0]), ('W-Mart', 'DELL') )
def autolabel(rects):
    for rect in rects:
        height = rect.get_height()
        ax.text(rect.get_x()+rect.get_width()/2., 1.05*height,
            '%d'%int(height),
                ha='center', va='bottom')
autolabel(rects1)
autolabel(rects2)
plt.show()
```

The command of `np.arange(3)` will generate four values from 0 to 3 while `ax.set_xticks()` generates ticks. The corresponding output is shown in the following figure:



In the preceding code, we color W-Mart's EPS red, `color='r'`, and DELL's EPS yellow, `color='y'`. The eight different colors and their representing letters are given in the following table:

Letter	Color	Letter	Color
'b'	Blue	'm'	Magenta
'g'	Green	'y'	Yellow
'r'	Red	'k'	Black
'c'	Cyan	'w'	White

Using different shapes

To make our graphs more eye-catching, we could use different shapes. In the following table, various shapes and their corresponding symbols are presented:

Character	Description	Character	Description
"_"	Solid line style	"3"	tri_left marker
"_ _"	Dashed line style	"4"	tri_right marker
"_ ."	Dash-dot line style	"s"	Square marker
"_ . ."	Dotted line style	"p"	Pentagon marker
"_ ."	Point marker	"*"	Star marker
"_ ,"	Pixel marker	"h"	Hexagon1 marker
"_ o"	Circle marker	"H"	Hexagon2 marker
"_ v"	triangle_down marker	"+"	Plus marker
"_ ^"	triangle_up marker	"x"	X marker
"_ <"	triangle_left marker	"D"	Diamond marker
"_ >"	triangle_right marker	"d"	Thin_diamond marker
"_ 1"	tri_down marker	" "	Vline marker
"_ 2"	tri_up marker	"_ "	Hline marker

Graphical representation of the portfolio diversification effect

In finance, we could remove firm-specific risk by combining different stocks in our portfolio. First, let us look at a hypothetical case by assuming that we have 5 years' annual returns of two stocks as follows:

Year	Stock A	Stock B
2009	0.102	0.1062
2010	-0.02	0.23
2011	0.213	0.045
2012	0.12	0.234
2013	0.13	0.113

We form an equal-weighted portfolio using those two stocks. Using the `mean()` and `std()` functions contained in NumPy, we can estimate their means, standard deviations, and correlation coefficients as follows:

```
>>>import numpy as np
>>>A=[0.102,-0.02, 0.213,0.12,0.13]
>>>B=[0.1062,0.23, 0.045,0.234,0.113]
>>>port_EW=(np.array(ret_A)+np.array(ret_B))/2.
>>>round(np.mean(A),3),round(np.mean(B),3),round(np.mean(port_EW),3)
(0.109, 0.146, 0.127)
>>>round(np.std(A),3),round(np.std(B),3),round(np.std(port_EW),3)
(0.075, 0.074, 0.027)
```

In the preceding code, we estimate mean returns, their standard deviations for individual stocks, and an equal-weighted portfolio. The volatility (standard deviation) of such an equal-weighted portfolio is 2.7 percent, considerably smaller than those of the individual stock (7.5 percent for stock A and 7.4 percent for stock B). In the following program, we use a graph to represent such an effect:

```
import numpy as np
import matplotlib.pyplot as plt
year=[2009,2010,2011,2012,2013]
ret_A=[0.102,-0.02, 0.213,0.12,0.13]
ret_B=[0.1062,0.23, 0.045,0.234,0.113]
```

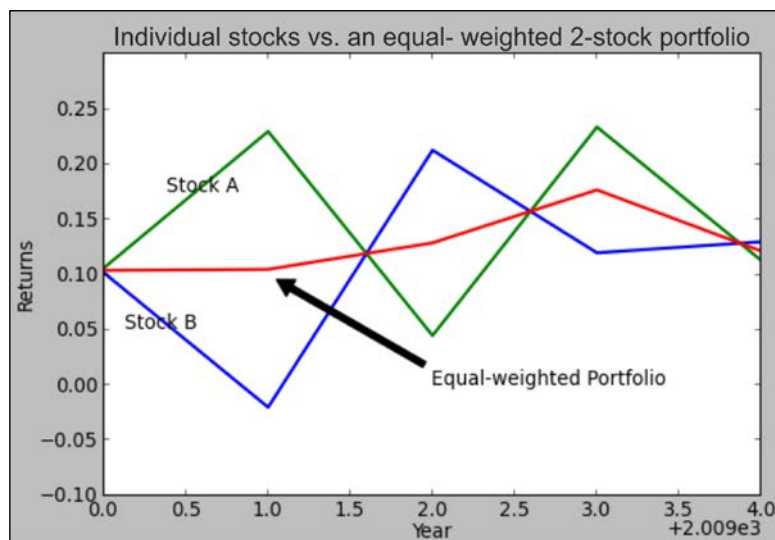
```

port_EW=(np.array(ret_A)+np.array(ret_B))/2.

plt.figtext(0.2,0.65,"Stock A")
plt.figtext(0.15,0.4,"Stock B")
plt.xlabel("Year")
plt.ylabel("Returns")
plt.plot(year,ret_A,lw=2)
plt.plot(year,ret_B,lw=2)
plt.plot(year,port_EW,lw=2)
plt.title("Indiviudal stocks vs. an equal-weighted 2-stock portfolio")
plt.annotate('Equal-weighted Portfolio', xy=(2010, 0.1), xytext=(2011.,0)
,arrowprops=dict(facecolor='black',shrink=0.05),
plt.ylim(-0.1,0.3)
plt.show()

```

The output graph corresponding to this code is given as follows:



In the preceding code, we add an arrow to indicate which line is associated with our equal-weighted, two-stock portfolio by using the function called `annotate()`. The pair of values for `xy=(2010, 0.1)` is for the destination of the arrow, and `xytext=(2011, 0)` is the starting point of the arrow. The color of the arrow is black. For more detail about the function, just type `help(plt.annotate)` after issuing `import matplotlib.pyplot as plt`. From the preceding graph, we see that the fluctuation, uncertainty, or risk of our equal-weighted portfolio is much smaller than those of individual stocks in its portfolio. We can also estimate their means, standard deviation, and correlation coefficient. The correlation coefficient between those two stocks is -0.75, and this is the reason why we could diversify away firm-specific risk by forming an even equal-weighted portfolio as shown in the following code:

```
>>>import scipy as sp
>>>sp.corrcoef(A,B)
array([[ 1.          , -0.74583429],
       [-0.74583429,  1.          ]])
```

In the preceding example, we use hypothetical numbers (returns) for two stocks. How about IBM and W-Mart? First, we have to know how to retrieve historical price data from Yahoo! Finance.

Number of stocks and portfolio risk

We know that when we increase the number of stocks in a portfolio, we would diversify away firm-specific risk. However, how many stocks do we need to diversify away from most of the firm-specific risk? Statman (1987) argues that we need at least 30 stocks. The title of his paper is *How Many Stocks Make a Diversified Portfolio?* in the *Journal of Financial Quantitative Analysis*. Based on his relationship between n (number of stocks) and the ratio of the portfolio standard deviation to the standard deviation of a single stock, we have the graph showing the relationship between the two. The values in the following table are from Statman (1987) where n is the number of stocks in a portfolio, $\bar{\sigma}_p$ is the standard deviation of the annual portfolio returns, and $\bar{\sigma}_1$ is the average of the standard deviation of a one-stock portfolio:

n	$\bar{\sigma}_p$	$\frac{\bar{\sigma}_p}{\bar{\sigma}_1}$	n	$\bar{\sigma}_p$	$\frac{\bar{\sigma}_p}{\bar{\sigma}_1}$
1	49.236	1.00	45	20.316	0.41
2	37.358	0.76	50	20.203	0.41
4	29.687	0.60	75	19.860	0.40
6	26.643	0.54	100	19.686	0.40

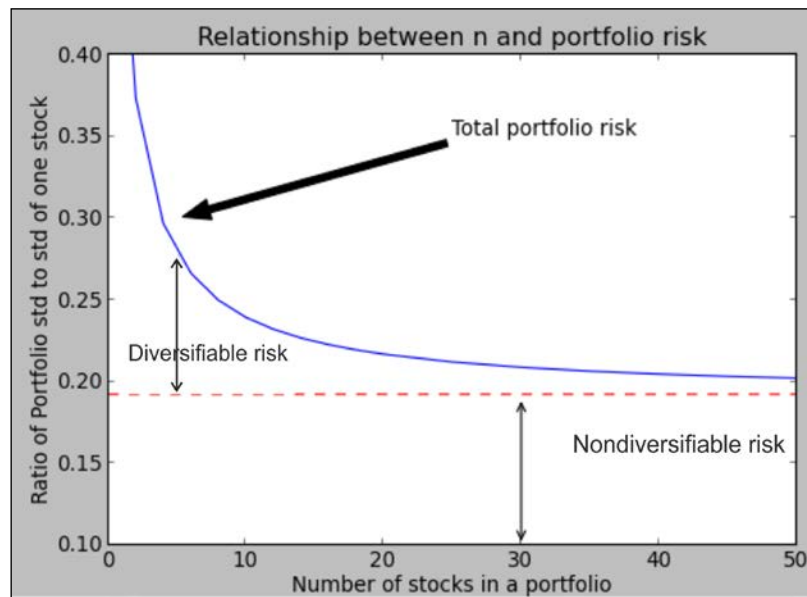
n	$\bar{\sigma}_p$	$\frac{\bar{\sigma}_p}{\bar{\sigma}_1}$	n	$\bar{\sigma}_p$	$\frac{\bar{\sigma}_p}{\bar{\sigma}_1}$
8	24.983	0.51	200	19.432	0.39
10	23.932	0.49	300	19.336	0.39
12	23.204	0.47	400	19.292	0.39
14	22.670	0.46	500	19.265	0.39
16	22.261	0.45	600	19.347	0.39
18	21.939	0.45	700	19.233	0.39
20	21.677	0.44	800	19.224	0.39
25	21.196	0.43	900	19.217	0.39
30	20.870	0.42	1000	19.211	0.39
35	20.634	0.42	∞	19.158	0.39
40	20.456	0.42			

The following is our program:

```
from matplotlib.pyplot import *
n=[1,2,4,6,8,10,12,14,16,18,20,25,30,35,40,45,50,75,100,200,300,400,500,600,700,800,900,1000]
port_sigma=[0.49236,0.37358,0.29687,0.26643,0.24983,0.23932,0.23204,0.22670,0.22261,0.21939,0.21677,0.21196,0.20870,0.20634,0.20456,0.20316,0.20203,0.19860,0.19686,0.19432,0.19336,0.19292,0.19265,0.19347,0.19233,0.19224,0.19217,0.19211,0.19158]
xlim(0,50)
ylim(0.1,0.4)
hlines(0.19217, 0, 50, colors='r', linestyle='dashed')
annotate('', xy=(5, 0.19), xycoords = 'data',xytext = (5, 0.28),
textcoords = 'data',arrowprops = {'arrowstyle':'<->'})
annotate('', xy=(30, 0.19), xycoords = 'data',xytext = (30, 0.1),
textcoords = 'data',arrowprops = {'arrowstyle':'<->'})
annotate('Total portfolio risk', xy=(5,0.3),xytext=(25,0.35),
arrowprops=dict(facecolor='black',shrink=0.02))
figtext(0.15,0.4,"Diversifiable risk")
figtext(0.65,0.25,"Nondiversifiable risk")
plot(n[0:17],port_sigma[0:17])
```

```
title("Relationship between n and portfolio risk")
xlabel("Number of stocks in a portfolio")
ylabel("Ratio of Portfolio std to std of one stock")
show()
```

In the preceding code, the values for n , that is, the number of stocks in a portfolio, and port_sigma , that is, the portfolio standard deviation, are from Statman (1987). The functions `ylim()` and `xlim()` set the lower and upper limits for the x axis and y axis respectively, as shown in the following figure:



Retrieving historical price data from Yahoo! Finance

The function called `quotes_historical_yahoo()` in the `matplotlib` module could be used to download historical price data from Yahoo! Finance. For example, we want to download daily price data for IBM over the period from January 1, 2012 to December 31, 2012, we have the following four-line Python code:

```
>>>from matplotlib.finance import quotes_historical_yahoo
>>>date1=(2012, 1, 1)
>>>date2=(2012, 12,31)
>>>price=quotes_historical_yahoo('IBM', date1, date2)
```

To download IBM's historical price data up to today, we could use the `datetime.date.today()` function as follows:

```
>>>import datetime
>>>import matplotlib.finance as finance
>>>import matplotlib.mlab as mlab
>>>ticker = 'IBM'
>>>begdate = datetime.date(2013,1,1)
>>>enddate = datetime.date.today()
>>>price = finance.fetch_historical_yahoo(ticker, begdate, enddate)
>>>r = mlab.csv2rec(price); price.close()
>>>r.sort()
```

The `r.sort()` function will sort the time series in ascending order since the original data from Yahoo! Finance is arranged in descending order. To check the number of observations, we use the `len()` function. To check the first observation and the last one, we use `r[0]` and `r[-1]`; see the following results:

```
>>>len(r)
217
>>>r[0:4]
rec.array([(datetime.date(2013, 1, 2), 194.09, 196.35, 193.8, 196.35,
4234100, 192.61),
          (datetime.date(2013, 1, 3), 195.67, 196.29, 194.44, 195.27,
3644700, 191.55),
          (datetime.date(2013, 1, 4), 194.19, 194.46, 192.78, 193.99,
3380200, 190.3),
          (datetime.date(2013, 1, 7), 193.4, 193.78, 192.34, 193.14,
2862300, 189.46)],
          dtype=[('date', '<O'), ('open', '<f8'), ('high', '<f8'), ('low',
'<f8'), ('close', '<f8'), ('volume', '<i4'), ('adj_close', '<f8')])
>>>
```

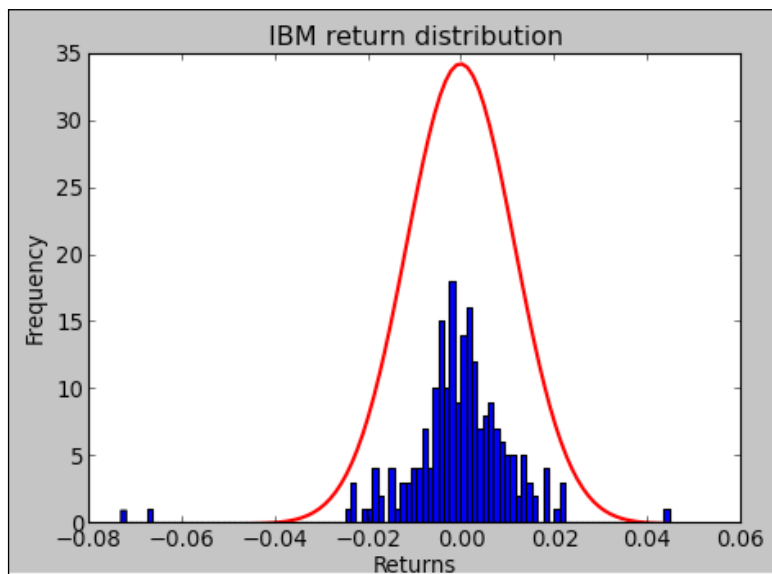
Histogram showing return distribution

In finance, we use mean returns to represent the expected returns and use the standard deviation of returns to represent the risk. A histogram could be used to show those two. If the location is on the right, it means the stock has a higher expected return, while the dispersion indicates the risk level: wider dispersion suggests a higher risk as shown in the following program:

```
from matplotlib.pyplot import *
from matplotlib.finance import quotes_historical_yahoo
```

```
import numpy as np
import matplotlib.mlab as mlab
ticker='IBM'
begdate=(2013,1,1)
enddate=(2013,11,9)
p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
ret = (p.aclose[1:] - p.aclose[:-1])/p.aclose[1:]
[n,bins,patches] = hist(ret, 100)
mu = np.mean(ret)
sigma = np.std(ret)
x = mlab.normpdf(bins, mu, sigma)
plot(bins, x, color='red', lw=2)
title("IBM return distribution")
xlabel("Returns")
ylabel("Frequency")
show()
```

The output corresponding to the preceding code is given as follows:



The next program makes the trading days more evenly distributed:

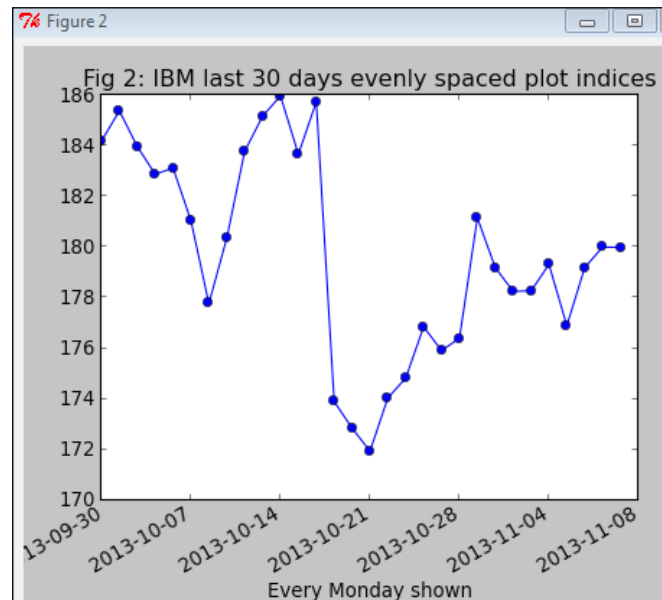
```

from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
import matplotlib.ticker as ticker
import datetime
import matplotlib.finance as finance

myticker = 'IBM'
begdate = datetime.date(2013,1,1)
enddate = datetime.date.today()
price = finance.fetch_historical_yahoo(myticker, begdate, enddate)
r = mlab.csv2rec(price); price.close()
r.sort()
r = r[-30:] # get the last 30 days
fig, ax = plt.subplots()
ax.plot(r.date, r.adj_close, 'o-')
ax.set_title('Fig. 1: IBM last 30 days with gaps on weekends')
fig.autofmt_xdate()
N = len(r)
ind = np.arange(N) # the evenly spaced plot indices
def format_date(x, pos=None):
    thisind = np.clip(int(x+0.5), 0, N-1)
    return r.date[thisind].strftime('%Y-%m-%d')
fig, ax = plt.subplots()
ax.plot(ind, r.adj_close, 'o-')
plt.xlabel("Every Monday shown")
ax.set_title('Fig 2: IBM last 30 days evenly spaced plot indices')
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
fig.autofmt_xdate()
plt.show()

```

Only the second figure is shown to save space:



Comparing stock and market returns

We could download daily price data from Yahoo! Finance for one stock and the market represented by S&P 500. Then estimate their returns and represent them via a graph using the following code:

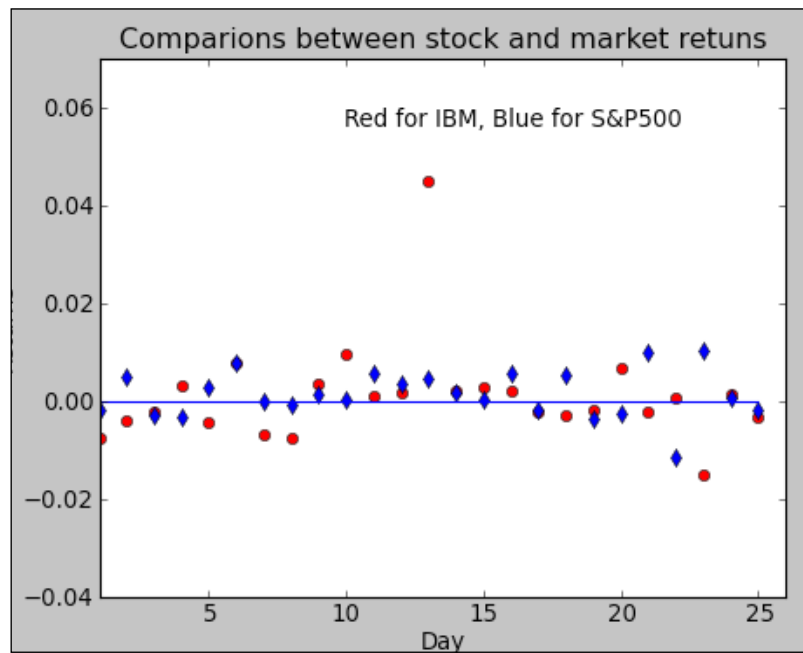
```
from matplotlib.pyplot import *
from matplotlib.finance import quotes_historical_yahoo
import numpy as np

def ret_f(ticker,begdate,enddate):
    p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
    adjusted=True)
    return((p.aclose[1:] - p.aclose[:-1])/p.aclose[:-1])

begdate=(2013,1,1)
enddate=(2013,2,9)
ret1=ret_f('IBM',begdate,enddate)
ret2=ret_f('^GSPC',begdate,enddate)
n=min(len(ret1),len(ret2))
```

```
s=np.ones(n)*2
t=range(n)
line=np.zeros(n)
plot(t,ret1[0:n], 'ro',s )
plot(t,ret2[0:n], 'bd',s)
plot(t,line,'b',s)
figtext(0.4,0.8,"Red for IBM, Blue for S&P500")
xlim(1,n)
ylim(-0.04,0.07)
title("Comparisons between stock and market returns")
xlabel("Day")
ylabel("Returns")
show()
```

The output corresponding to the preceding code is given as follows:

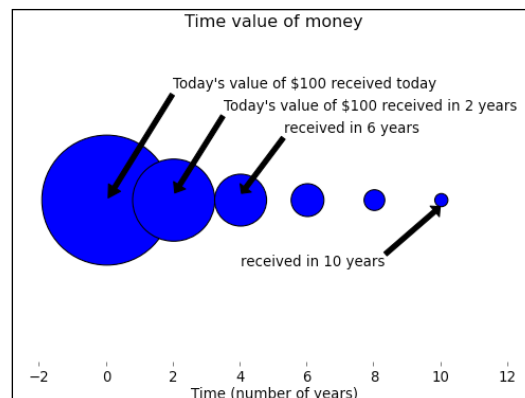


Understanding the time value of money

In finance, we know that \$100 received today is more valuable than \$100 received one year later. If we use size to represent the difference, we could have the following Python program to represent the same concept:

```
from matplotlib.pyplot import *
fig1 = figure(facecolor='white')
ax1 = axes(frameon=False)
ax1.set_frame_on(False)
ax1.get_xaxis().tick_bottom()
ax1.axes.get_yaxis().set_visible(False)
x=range(0,11,2)
x1=range(len(x),0,-1)
y = [0]*len(x);
annotate("Today's value of $100 received today",xy=(0,0),xytext=(2,0.001),
,arrowprops=dict(facecolor='black',shrink=0.02))
annotate("Today's value of $100 received in 2 years",xy=(2,0.00005),xytext=(3.5,0.0008),arrowprops=dict(facecolor='black',shrink=0.02))
annotate("received in 6 years",xy=(4,0.00005),xytext=(5.3,0.0006),arrowprops=dict(facecolor='black',shrink=0.02))
annotate("received in 10 years",xy=(10,-0.00005),xytext=(4,-0.0006),arrowprops=dict(facecolor='black',shrink=0.02))
s = [50*2.5**n for n in x1];
title("Time value of money ")
xlabel("Time (number of years)")
scatter(x,y,s=s);
show()
```

The output graph is shown as follows:



Candlesticks representation of IBM's daily price

We could use candlesticks to represent the daily opening, high, low, and closing prices. The vertical line represents high and low prices, while a rectangular bar represents open-close span. When the close price is higher than the opening price, we have a black bar. Otherwise, we would have a red bar. The following program will show exactly this:

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

from matplotlib.dates import DateFormatter, WeekdayLocator, HourLocator,
DayLocator, MONDAY

from matplotlib.finance import quotes_historical_yahoo, candlestick,\
    plot_day_summary, candlestick2

date1 = ( 2013, 10, 20)
date2 = ( 2013, 11, 10 )
ticker='IBM'

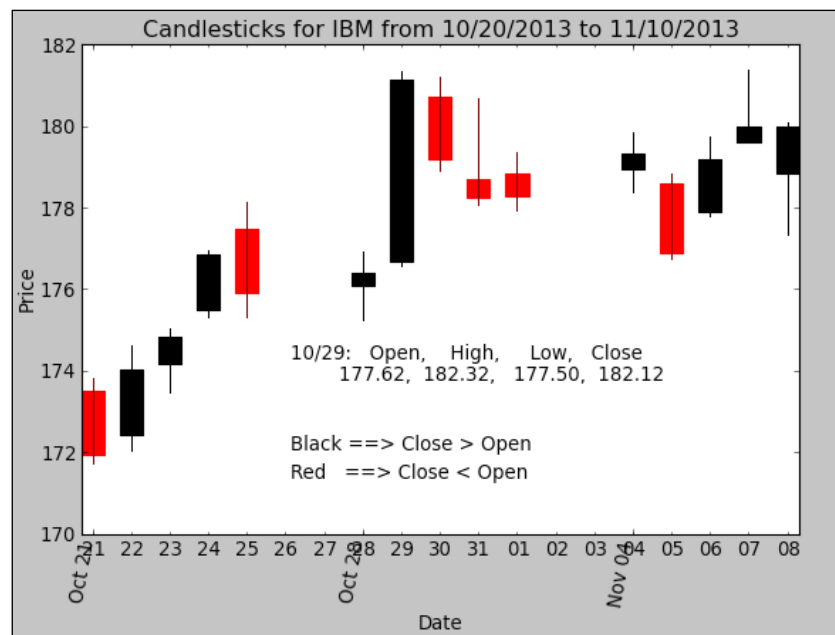
mondays = WeekdayLocator(MONDAY)          # major ticks on the mondays
alldays  = DayLocator()                   # minor ticks on the days
weekFormatter = DateFormatter('%b %d')   # e.g., Jan 12
dayFormatter = DateFormatter('%d')       # e.g., 12

quotes = quotes_historical_yahoo(ticker, date1, date2)
if len(quotes) == 0:
    raise SystemExit

fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.2)
ax.xaxis.set_major_locator(mondays)
ax.xaxis.set_minor_locator(alldays)
ax.xaxis.set_major_formatter(weekFormatter)
ax.xaxis.set_minor_formatter(dayFormatter)
plot_day_summary(ax, quotes, ticksize=3)
candlestick(ax, quotes, width=0.6)
ax.xaxis_date()
ax.autoscale_view()
plt.setp( plt.gca().get_xticklabels(), rotation=80,
horizontalalignment='right')
```

```
plt.figtext(0.35,0.45, '10/29:   Open,   High,   Low,   Close')
plt.figtext(0.35,0.42, '           177.62, 182.32, 177.50, 182.12')
plt.figtext(0.35,0.32, 'Black ==> Close > Open ')
plt.figtext(0.35,0.28, 'Red  ==> Close < Open ')
plt.title('Candlesticks for IBM from 10/20/2013 to 11/10/2013')
plt.ylabel('Price')
plt.xlabel('Date')
plt.show()
```

The output graph is shown as follows:

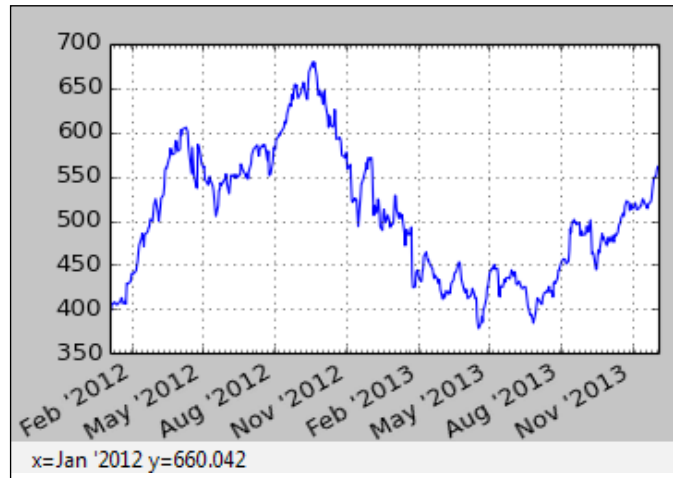


Graphical representation of two-year price movement

We could show the price movement for a given ticker from the first date to the second date. In the following program, we have three input values: `ticker`, `begdate` (first date), and `enddate` (second date):

```
import datetime
import matplotlib.pyplot as plt
from matplotlib.finance import quotes_historical_yahoo
from matplotlib.dates import MonthLocator, DateFormatter
ticker='AAPL'
begdate= datetime.date( 2012, 1, 2 )
enddate = datetime.date( 2013, 12,4)
months    = MonthLocator(range(1,13), bymonthday=1, interval=3) # every
3rd month
monthsFmt = DateFormatter("%b '%Y")
x = quotes_historical_yahoo(ticker, begdate, enddate)
if len(x) == 0:
    print ('Found no quotes')
    raise SystemExit
dates = [q[0] for q in x]
closes = [q[4] for q in x]
fig, ax = plt.subplots()
ax.plot_date(dates, closes, '-')
ax.xaxis.set_major_locator(months)
ax.xaxis.set_major_formatter(monthsFmt)
ax.xaxis.set_minor_locator(mondays)
ax.autoscale_view()
ax.grid(True)
fig.autofmt_xdate()
```

The output graph is shown as follows:



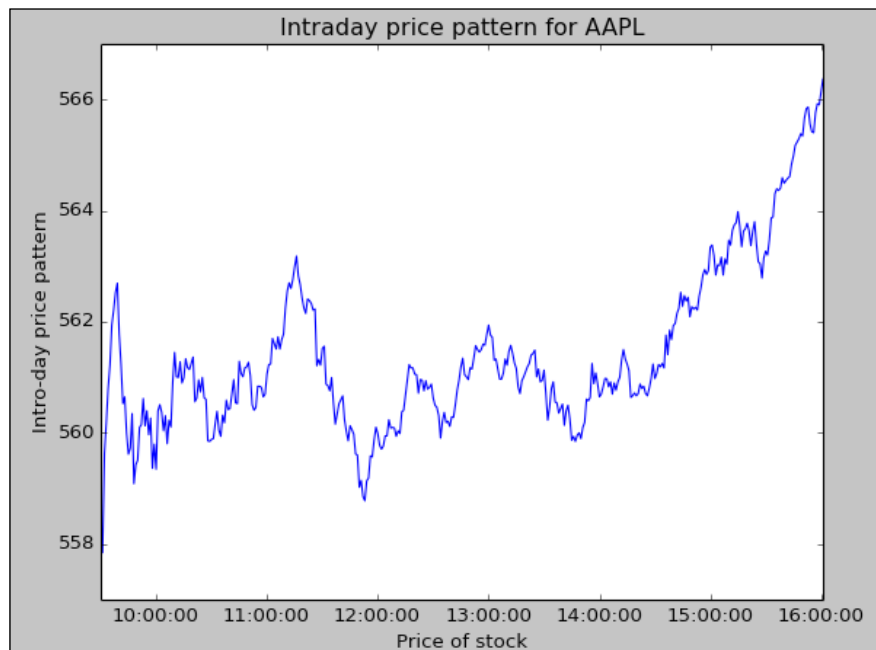
IBM's intra-day graphical representations

We could demonstrate the price movement of a stock for a given period, for example, from January 2009 to today. First, let's look at the intra-day price pattern. The following program will be explained in the next chapter:

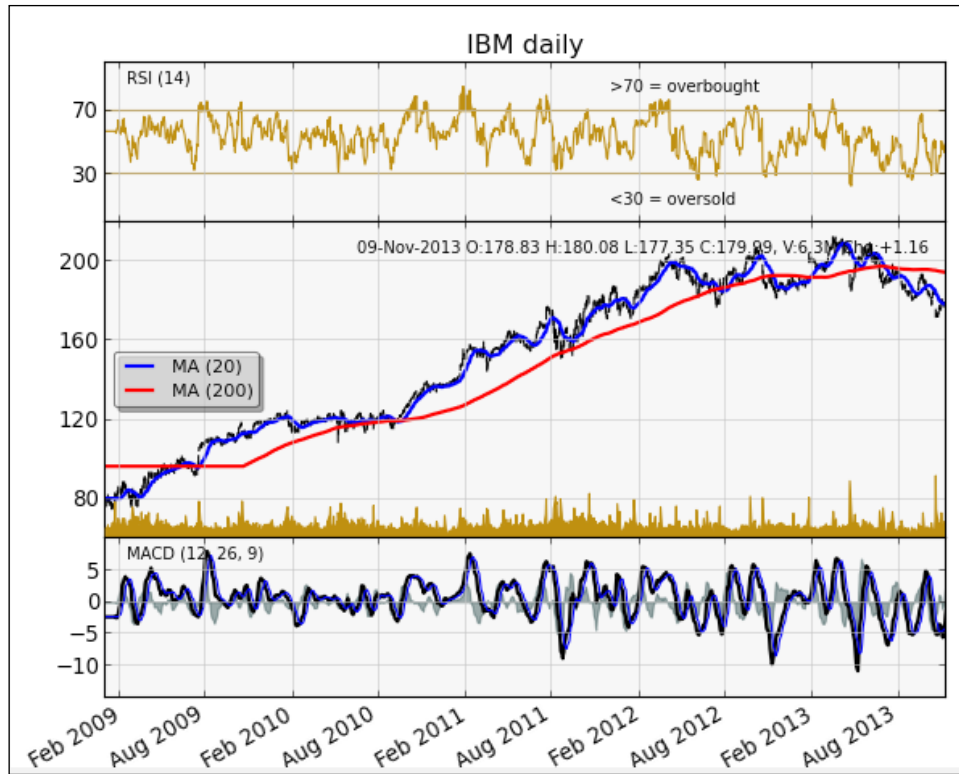
```
import pandas as pd, numpy as np, datetime
ticker='AAPL'
path='http://www.google.com/finance/getprices?q=ttt&i=60&p=1d&f=d,o,h,l,c,v'
p=np.array(pd.read_csv(path.replace('ttt',ticker),skiprows=7,header=None))
date=[]
for i in range(0,len(p)):
    if p[i][0][0]=='a':
        t= datetime.datetime.fromtimestamp(int(p[i][0].replace('a','')))
        date.append(t)
    else:
        date.append(t+datetime.timedelta(minutes =int(p[i][0])))
```

```
final=pd.DataFrame(p,index=date)
final.columns=['a','Open','High','Low','Close','Vol']
del final['a']
x=final.index
y=final.Close
title('Intraday price pattern for ttt'.replace('ttt',ticker))
xlabel('Price of stock')
ylabel('Intro-day price pattern')
plot(x,y)
show()
```

The graph is shown in the following image:



A more complex program that we can run to find the intra-day pattern could be found at http://matplotlib.org/examples/pylab_examples/finance_work2.html. Our program is quite similar to the original program posted at the link. The Python program used to generate the following graph is not shown to save space:



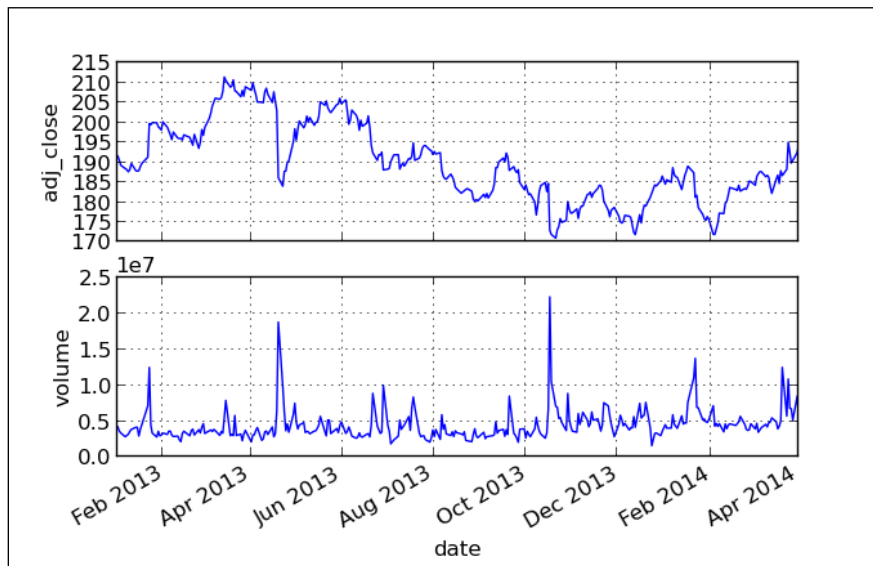
Presenting both closing price and trading volume

Sometimes, we like to view both price movement and trading volumes simultaneously. The following program accomplishes this:

```
>>>from pylab import plotfile, show
>>>import matplotlib.finance as finance
>>>ticker = 'IBM'
>>>begdate = datetime.date(2013,1,1)
>>>enddate = datetime.date.today()
```

```
>>>x= finance.fetch_historical_yahoo(ticker, begdate, enddate)
>>>plotfile(x, (0,6,5))
>>>show()
```

The output graph is shown as follows:



Adding mathematical formulae to our graph

In finance, we use many mathematical formulae. Occasionally, we need to add a mathematical formula to our figure. A set of formulae for a call option by using the matplotlib module is shown in the following program:

```
import numpy as np
import matplotlib.mathtext as mathtext
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rc('image', origin='upper')
parser = mathtext.MathTextParser("Bitmap")
r'$\left[\left\lfloor\frac{5}{\left(3\right)^4}y\right\right]$'
rgba1, depth1 = parser.to_rgba(r'$d_2=\frac{\ln(S_0/K)+(r-\sigma^2/2)T}{\sigma\sqrt{T}}=d_1-\sigma\sqrt{T}$', color='blue', fontsize=12, dpi=200)
```



```
rgba2, depth2 = parser.to_rgba(r'$d_1=\frac{\ln(S_0/K)+(r+\sigma^2/2)T}{\sigma\sqrt{T}}$', color='blue', fontsize=12, dpi=200)
rgba3, depth3 = parser.to_rgba(r'$c=S_0N(d_1)-Ke^{-rT}N(d_2)$',
color='red', fontsize=14, dpi=200)
fig = plt.figure()
fig.figimage(rgba1.astype(float)/255., 100, 100)
```

The program crucially depends on the LaTeX format, which is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTeX is the de facto standard for the communication and publication of scientific documents according to the web page of <http://latex-project.org/>. The following is the output:

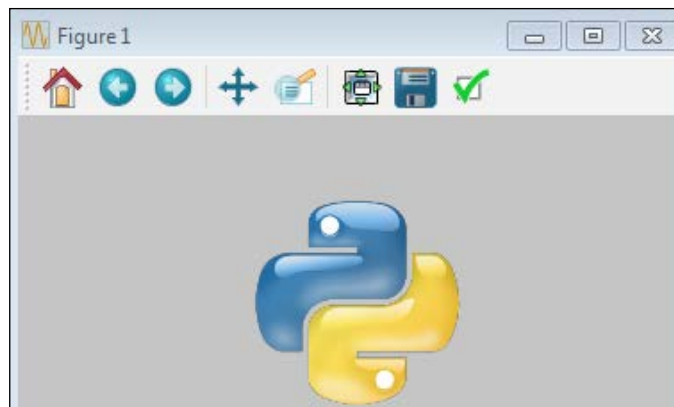
$$c = S_0 N(d_1) - K e^{-rT} N(d_2)$$
$$d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$
$$d_2 = \frac{\ln(S_0/K) + (r - \sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}$$

Adding simple images to our graphs

Assume that we have the Python logo saved under `C:\temp`. The logo could be downloaded at http://canisius.edu/~yany/python_logo.png. The following code could be used to retrieve it:

```
>>>import matplotlib.pyplot as plt
>>>import matplotlib.cbook as cbook
>>>image_file = cbook.get_sample_data('C:/temp/python_logo.png')
>>>image = plt.imread(image_file)
>>>plt.imshow(image)
>>>plt.axis('off')
>>>plt.show()
```

The `cbook` module is a collection of utility functions and classes. Many of them are from the Python cookbook. Thus, it is named `cbook`. The following is the corresponding graph:

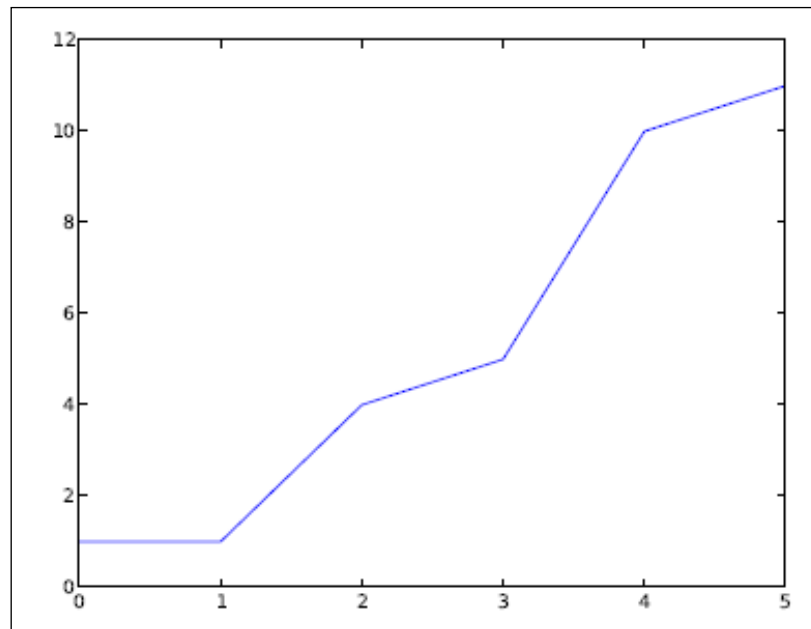


Saving our figure to a file

If we plan to save our figure as a .pdf file, we could use the following code:

```
>>>from matplotlib.pylab import *  
>>>plot([1,1,4,5,10,11])  
>>>savefig("c:/temp/test.pdf")
```

The following is the corresponding graph:



If we don't specify a particular path as done in the following code, the figure would be at the current working directory; usually it is under `C:\python27`:

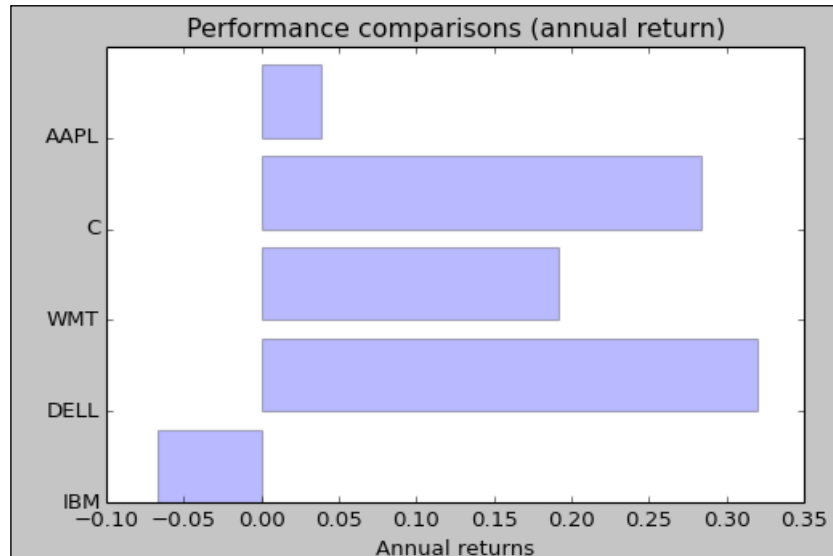
```
>>>savefig("test.pdf")
```

Performance comparisons among stocks

In the following program, we compare the performance of several stocks in terms of their returns in 2013:

```
import matplotlib.pyplot as plt; plt.rcdefaults()
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.finance import quotes_historical_yahoo
stocks = ('IBM', 'DELL', 'WMT', 'C', 'AAPL')
begdate=(2013,1,1)
enddate=(2013,11,30)
def ret_annual(ticker):
    x = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
    logret = log(x.aclose[1:]/x.aclose[:-1])
    return(exp(sum(logret))-1)
performance = []
for ticker in stocks:
    performance.append(ret_annual(ticker))
y_pos = np.arange(len(stocks))
plt.barh(y_pos, performance, left=0, alpha=0.3)
plt.yticks(y_pos, stocks)
plt.xlabel('Annual returns ')
plt.title('Performance comparisons (annual return)')
plt.show()
```

The related bar chart is shown in the following figure:



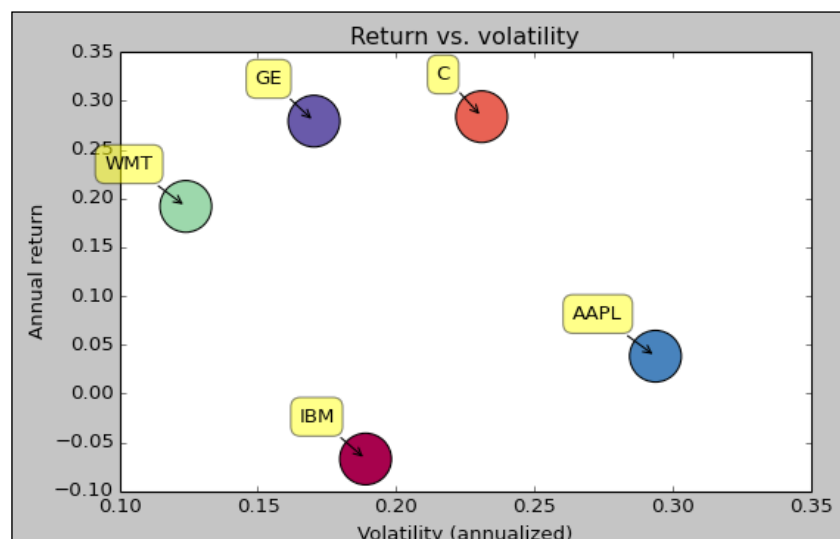
Comparing return versus volatility for several stocks

The following program shows the locations of five stocks on the return versus volatility graph:

```
import numpy as np
import matplotlib.pyplot as plt; plt.rcdefaults()
from matplotlib.finance import quotes_historical_yahoo
stocks = ('IBM', 'GE', 'WMT', 'C', 'AAPL')
begdate=(2013,1,1)
enddate=(2013,11,30)
def ret_vol(ticker):
    x = quotes_historical_yahoo(ticker,begdate,enddate,asobject=True,adjusted=True)
    logret = log(x.aclose[1:]/x.aclose[:-1])
    return(exp(sum(logret))-1,std(logret))
ret=[];vol=[]
```

```
for ticker in stocks:
    r,v=ret_vol(ticker)
    ret.append(r)
    vol.append(v*sqrt(252))
labels = ['{0}'.format(i) for i in stocks]
xlabel('Volatility (annualized)')
ylabel('Annual return')
title('Return vs. volatility')
plt.subplots_adjust(bottom = 0.1)
color=np.array([ 0.18,  0.96,  0.75,  0.3,  0.9])
plt.scatter(vol, ret, marker = 'o', c=color,s = 1000,cmap=plt.get_
cmap('Spectral'))
for label, x, y in zip(labels, vol, ret):
    plt.annotate(label,xy=(x,y),xytext=(-20,20),textcoords='offset
points',
                ha = 'right', va = 'bottom',bbox = dict(boxstyle = 'round,pad=0.5',
                fc = 'yellow', alpha = 0.5),arrowprops = dict(arrowstyle = '->',
                connectionstyle = 'arc3,rad=0'))
plt.show()
```

In the graph, each point represents one stock in terms of its annual return and annualized volatility. To make the picture more eye-catching, different colors are used as shown in the following figure:



Finding manuals, examples, and videos

The web page offering examples is <http://matplotlib.org/examples/index.html>. We believe it is a good idea to study the examples given on the web page before writing our own applications. The web pages related to matplotlib are as follows:

- <http://matplotlib.org/users/>
- <http://scipy-lectures.github.io/intro/matplotlib/matplotlib.html>

The web page of 5,000 examples is as follows:

- <http://matplotlib.org/examples/index.html>
- <http://www.youtube.com/watch?v=OfumUp3hZmQ>

How do we install ActivePython and matplotlib?

- <http://www.activestate.com/activepython/python-financial-scientific-modules> (5m, 37s)

Visual financial statements can be found at the following location:

- <http://www.youtube.com/watch?v=OfumUp3hZmQ>

Installing the matplotlib module independently

There are two steps to install the module:

1. Go to <http://matplotlib.org/downloads.html>.
2. Choose an appropriate package and download it, such as `matplotlib-1.2.1.win-amd64-py3.2.exe`.

Summary

In this chapter, we showed how to use the matplotlib module to vividly explain many financial concepts by using graph, pictures, color, and size. For example, in a two-dimensional graph, we showed a few stocks' returns and volatility, the NPV profile, multiple IRRs, and the portfolio diversification effect.

In *Chapter 8, Statistical Analysis of Time Series*, first we demonstrate how to retrieve historical time series data from several public data sources, such as Yahoo! Finance, Google Finance, Federal Reserve Data Library, and Prof. French's Data Library. Then, we discussed various statistical tests, such as T-test, F-test, and normality test. In addition, we presented Python programs to run **capital asset pricing model (CAPM)**, run a Fama-French three-factor model, estimate the Roll (1984) spread, estimate **Value at Risk (VaR)** for individual stocks, and also estimate the Amihud (2002) illiquidity measure, and the Pastor and Stambaugh (2003) liquidity measure for portfolios. For the issue of anomaly in finance, we tested the existence of the so-called January effect. For high-frequency data, we explained briefly how to draw intra-day price movement and retrieved data from the **Trade, Order, Report and Quote (TORQ)** database and the **Trade and Quote (TAQ)** database. The terms of use for Yahoo! Finance is at <http://finance.yahoo.com/badges/tos>.

Exercises

1. What is the potential usage of `matplotlib`?
2. How do we install `matplotlib`?
3. Does the `matplotlib` module depend on NumPy? Does it depend on SciPy?
4. Write a Python function to generate an NPV profile with a set of input cash flows.
5. Write a Python function to download daily price time series from Yahoo! Finance.
6. We have six-year return vectors for two stocks and intend to construct a simple equal-weighted portfolio. Interpret the following Python codes and explain the result for the portfolio:

```
>>>A=[0.09,0.02, -0.13,0.20,-0.09,-0.03]
>>>B=[0.10,-0.3, -0.02,0.14,-0.13,0.23]
>>>C=[0.08,-0.16, 0.033,-0.24,0.053,-0.39]
>>>port_EW=(A+B)/3.
```
7. What is the standard deviation in terms of stock daily returns for the stocks of IBM, DELL, WMT, and C and GE in 2011?
8. How do we estimate a moving beta for a set of given tickers?
9. How do we generate a histogram in terms of daily returns for IBM? You can use five-year daily data from Yahoo! Finance.

10. Which pair of stocks is more closely associated with each other among IBM, DELL, and WMT? Show the evidence. You can use the latest five-year data from Yahoo! Finance to support your arguments.
11. What is the Capital Market Line? How do we visualize this concept?
12. What is the Security Market Line? How do we visualize this concept?
13. Could you find empirical evidence to support or dispute the argument made by Statman (1987) that a well-diversifiable portfolio should at least be holding 30 stocks?
14. Construct an efficient frontier with the use of ten stocks from Yahoo! Finance. You can use either monthly or daily data.
15. How do we show the relationship between risk and returns?
16. What is the correlation between the US stock market and the Canadian stock market? What is the relationship between the US stock market and the Japanese stock market? For the US stock market, you can choose S&P500 (^GSPC for its ticker symbol from Yahoo! Finance). To search market indices, go to finance.yahoo.com first, type carat of (^) in the search bar, and press *Enter*. Your screen will look like the following screenshot:

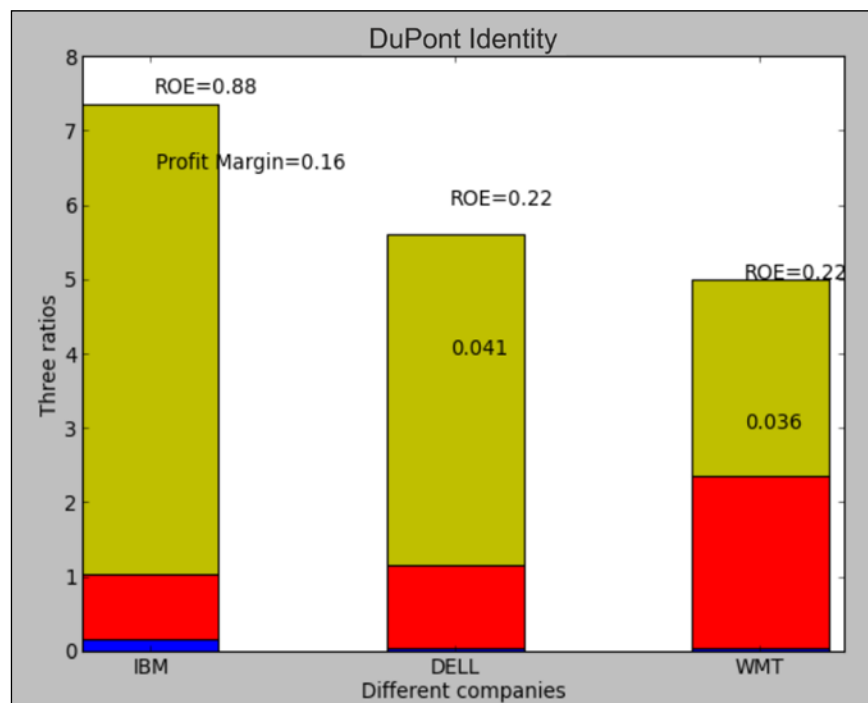
YAHOO! FINANCE		^		
Recent	% \$			
AAPL	+0.51%	^NSEI	CNX NIFTY	Index-NSE
DELL	0.00%	^DJI	Dow Jones Industrial Average	Index-DJI
IBM	+0.19%	^GSPC	S&P 500	Index-SNP
IBM.L	0.00%	^IXIC	NASDAQ Composite	Index-NIM
WMT	-0.97%	^VIX	VOLATILITY S&P 500	Index-Chicago Board.
^GSPC	-0.43%	^JKSE	Composite Index	Index-Jakarta
^DJI	-0.43%	^NDX	NASDAQ-100	Index-NIM
R	-0.51%	^N225	Nikkei 225	Index-Osaka Stock Ex.
		^TNX	CBOE Interest Rate 10-Year T-No	Index-Chicago Board.
		^HSI	HANG SENG INDEX	Index-Hong Kong

17. How do we randomly select ten stocks from a list of 100 tickers?
18. How do we draw an efficient frontier for ten stocks with five-year daily price data downloaded from Yahoo! Finance?
19. What might be the potential usages of a 3D graph for teaching finance?
20. Find more information about visual finance and offer your own comments.

21. Debut the following program:

```
>>>import scipy as sp
>>>import matplotlib.pyplot as plt
>>>cashflows=[100,-50,-50,-50,60]
>>>rate=npv=[]
>>>x=[0,0.8]
>>>y=[0,0]
>>>for i in range(1,30):
    rate.append(0.01*i)
    npv.append(sp.npv(0.01*i,cashflows[1:])+cashflows[0])
>>>plt.plot(x,y),plt.plot(rate,npv)
>>>plt.show()
```

22. There are some issues with the DuPont identity discussed in this chapter. For DELL and W-Mart, their ROE is the same, 0.22. However, the graph gives different heights. The reason is that the final value of ROE is the product of three components instead of summation. Find a way to overcome this issue, that is, your final output which is shown in the following graph should have the same heights:



8

Statistical Analysis of Time Series

Understanding the properties of financial time series is very important in finance. In this chapter, we will discuss many issues, such as downloading historical prices, estimating returns, total risk, market risk, correlation among stocks, correlation among different countries' markets from various types of portfolios, and a portfolio variance-covariance matrix; constructing an efficient portfolio and an efficient frontier; estimating Roll (1984) spread; and also estimating the Amihud (2002) illiquidity measure, and Pastor and Stambaugh's (2003) liquidity measure for portfolios. The two related Python modules used are `Pandas` and `statsmodels`.

In this chapter, we will cover the following topics:

- Installation of `Pandas` and `statsmodels`
- Using `Pandas` and `statsmodels`
- Open data sources, and retrieving data from Excel, text, CSV, and MATLAB files, and from a web page
- Date variable, `DataFrame`, and merging different datasets by date
- Term structure of interest rate, 52-week high and low trading strategy
- Return estimation and converting daily returns to monthly or annual returns
- Various tests, such as Durbin-Watson, T-test, and F-test
- **Capital asset pricing model (CAPM)**, rolling beta, and Fama-MacBeth regression
- Rolling volatility, correlation, forming ann-stock portfolio, variance-covariance matrix, portfolio optimization, and efficient frontier

- The Roll (1985) spread, Amihud's (2002) illiquidity, and Pastor and Stambaugh's (2003) liquidity measure
- Individual stock and portfolio's **Value at Risk (VaR)**
- January effect, size effect, and weekday effect
- Retrieving high-frequency data from Google Finance, **Trade, Order, Report, and Quotation (TORQ)**

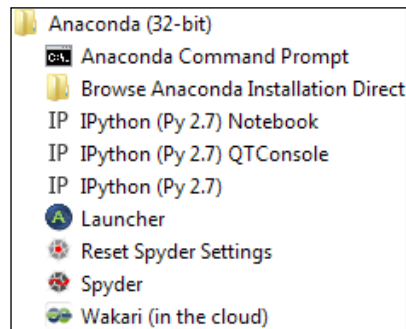
Installing Pandas and statsmodels

In the previous chapter, we used ActivePython. Although this package includes Pandas using PyPm to install, statsmodel is unavailable in PyPm. Fortunately, we could use Anaconda, introduced in *Chapter 4, 13 Lines of Python Code to Price a Call Option*. The major reason that we recommend Anaconda is that the package includes NumPy, SciPy, matplotlib, Pandas, and statsmodels. The second reason is its wonderful editor called Spyder.

To install Anaconda, perform the following two steps:

1. Go to <http://continuum.io/downloads>.
2. According to your machine, choose an appropriate package, such as **Anaconda-1.8.0-Windows-x86.exe** for a Windows version.

There are several ways to launch Python. After clicking on **Start | All Programs**, search **Anaconda**; we will see the following hierarchy:



In the following three sections, we show different ways to launch Python.

Launching Python using the Anaconda command prompt

For launching Python using the Anaconda command prompt, perform the following steps:

1. Click on **Start | All Programs**, search **Anaconda**, and then **Anaconda Command Prompt**; we will go to the directory that contains the Python executable file `python.exe`.
2. The exact path depends on individual installation. After typing `python`, we launch Python, see the first line of the next screenshot. To test whether `Pandas` and `statsmodels` are available, we import both of them. If there is no error, it means that we have them installed correctly:

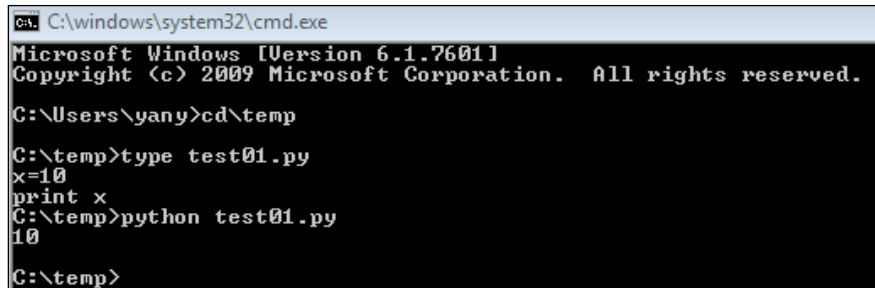
```
C:\Users\Yany\AppData\Local\Continuum\Anaconda>python
Python 2.7.5 [Anaconda 1.8.0 (32-bit)] (default, Jul  1 2013, 12:41:55) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Launching Python using the DOS window

We can launch Python from any directory. To add the directory of our Python executable file to the path, we perform the following steps:

1. First, launch Python via **Anaconda Command Prompt** (refer to the earlier steps), and then copy the full path. In the preceding example, it is `C:\Users\Yany\AppData\Local\Continuum\Anaconda`.
2. Then, click on **Start | Control Panel | View advanced system settings**, click on **Environment Variables**, find **PATH**, and then paste the full path given in the preceding paragraph. (Note that this is for Windows only.)
3. Now, we can launch Python from any directory or subdirectory. After clicking on **Start**, enter `cmd` in the **Search programs and files** textbox, and press the *Enter* key; a DOS window will appear. Just type `Python` to launch it. Assume that we have a two-line Python program called `test01.py` under the `C:\temp` directory. The two lines of code in that file are `x=10` and `print x`.

4. Again, click on **Start**, enter `cmd` in the **Search programs and files** textbox, and then press *Enter*. From the DOS window, we go to the correct directory. To show the program, issue `type test01.py`. To run the program, issue `python test01.py` as follows:



```
C:\windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

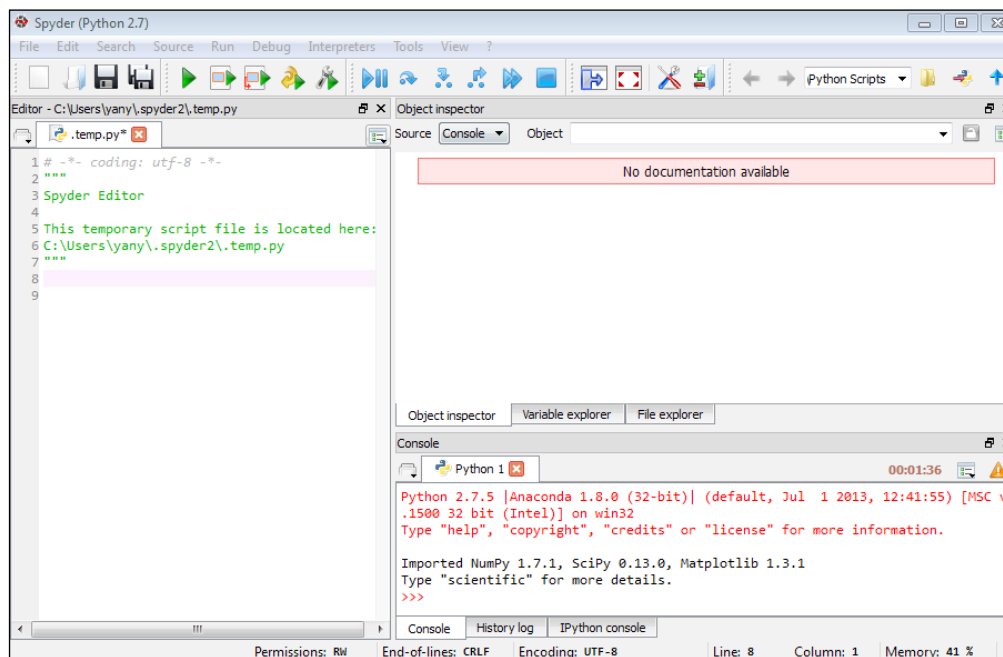
C:\Users\yany>cd\temp

C:\temp>type test01.py
x=10
print x
C:\temp>python test01.py
10
C:\temp>
```

Launching Python using Spyder

It is a much better choice to launch Python via Spyder, the editor accompanying Anaconda. For the Windows version, perform the following steps:

1. Click on **Start | All Files | Anaconda | Spyder**; the following three panels (windows) will appear. On the left-hand side panel, a default program called `temp.py` appears as shown in the following screenshot:



2. Let's try our first simple program. On the left-hand side panel, type `x=10`, press *Return*, and type `print x`. We can remove the contents of the default file before we type our two lines. In other words, our new program just has the following two lines:

```
x=10
Print x
```

3. After clicking on the green run button on the menu bar, you will be asked to save it. After that, the output will show on the bottom-right panel called **Console** as shown in the following screenshot:

```
>>> runfile('C:/temp/test01.py', wdir=r'C:/temp')
10
>>>
```

In the preceding output, the first line tells us the name of the program and the working directory (`wdir`). As for the left-hand side panel, we can upload our programs and modify them. One of the good features is that we can open many Python programs simultaneously.

Using Pandas and statsmodels

We give a few examples in the following section for the two modules we are going to use intensively in the rest of the book. Again, the `Pandas` module is for data manipulation and the `statsmodels` module is for the statistical analysis.

Using Pandas

In the following example, we generate two time series starting from January 1, 2013. The names of those two time series (columns) are A and B:

```
>>>import numpy as np
>>>import pandas as pd
>>>dates=pd.date_range('20130101',periods=5)
>>>np.random.seed(12345)
>>>x=pd.DataFrame(np.random.rand(5,2),index=dates,columns=('A','B'))
```

First, we import both NumPy and Pandas modules. The `pd.date_range()` function is used to generate an index array. The `x` variable is a Pandas' data frame with dates as its index. Later in this chapter, we will discuss `pd.DataFrame()`. The `columns()` function defines the names of those columns. Because the `seed()` function is used in the program, anyone can generate the same values. The `describe()` function offers the properties of those two columns, such as mean and standard deviation. Again, we call such a function as shown in the following code:

```
>>>x
          A          B
2013-01-01  0.929616  0.316376
2013-01-02  0.183919  0.204560
2013-01-03  0.567725  0.595545
2013-01-04  0.964515  0.653177
2013-01-05  0.748907  0.653570
>>>x.describe()
          A          B
count  5.000000  5.000000
mean   0.678936  0.484646
std    0.318866  0.209761
min    0.183919  0.204560
25%    0.567725  0.316376
50%    0.748907  0.595545
75%    0.929616  0.653177
max    0.964515  0.653570
>>>
```

Assume that we plan to replace missing values (NaN) with the mean of the time series. The two functions used are `mean()` and `fillna()`:

```
>>>import pandas as pd
>>>import numpy as np
>>>x=pd.Series([0.1,0.02,-0.03,np.nan,0.130,0.125])
>>>x
0    0.100
1    0.020
2   -0.030
3     NaN
```

```

4      0.130
5      0.125
dtype: float64
>>>m=np.mean(x)
>>>round(m,4)
0.069
>>>y=x.fillna(m)
>>>y
0      0.100
1      0.020
2     -0.030
3      0.069 # nan is replaced with the mean
4      0.130
5      0.125
dtype: float64
>>>

```

Examples from statsmodels

In statistics, **ordinary least square (OLS)** regression is a method for estimating the unknown parameters in a linear regression model. It minimizes the sum of squared vertical distances between the observed values and the values predicted by the linear approximation. The OLS method is used extensively in finance. Assume that we have the following equation where y is an n by 1 vector (array), and x is an n by $(m+1)$ matrix, a return matrix (n by m), plus a vector that contains 1 only. N is the number of observations, and m is the number of independent variables:

$$y = \beta * x + \epsilon_t \quad (1)$$

In the following program, after generating the x and y vectors, we run an OLS regression (a linear regression). The last line prints the parameters only:

```

>>>import numpy as np
>>>import statsmodels.api as sm
>>>y=[1,2,3,4,2,3,4]
>>>x=range(1,8)
>>>x=sm.add_constant(x)
>>>results=sm.OLS(y,x).fit()
>>>print results.params

```


The output is shown as follows:

```
>>>[ 1.28571429  0.35714286]
```

Open data sources

Since this chapter explores the statistical properties of time series, we need certain data. It is a great idea to employ publicly available economic, financial, and accounting data since every reader can download these time series with no cost. The free data sources are summarized in the following table:

Name	Web page
Yahoo! Finance	http://finance.yahoo.com Current and historical pricing, BS, IS, and so on
Google Finance	http://www.google.com/finance Current and historical trading prices
Federal Reserve Bank Data Library	http://www.federalreserve.gov/releases/h15/data.htm Interest rates, rates for AAA, AA rated bonds, and so on Financial statements
Russell indices	http://www.russell.com Russell indices
Prof. French's Data Library	http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html Fama-French factors, market index, risk-free rate, and industry classification
Census Bureau	http://www.census.gov/ http://www.census.gov/compendia/statab/hist_stats.html Census data
Bondsonline	http://www.bondsonline.com/ Bond data
U.S. Department of the Treasury	http://www.treas.gov U.S. Treasury ? yield
Bureau of Labor Statistics	http://download.bls.gov/ http://www.bls.gov/ Inflation, employment, unemployment, pay, and benefits Business cycles, vital statistics, and report of Presidents

We can easily download many time series from these sources. For example, to download IBM historical daily price data from Yahoo! Finance, we can perform the following steps:

1. Go to Yahoo! Finance at <http://finance.yahoo.com>.
2. Enter `IBM` in the search box.
3. Click on **Historical Prices**.
4. Select the starting and ending dates, and click on **Get Prices**.
5. Go to the bottom of the page, and click on **Download to Spreadsheet**.

The first and last several lines are given as follows:

```
>>>Date,Open,High,Low,Close,Volume,Adj Close
2013-07-26,196.59,197.37,195.00,197.35,2485100,197.35
2013-07-25,196.30,197.83,195.66,197.22,3014300,197.22
2013-07-24,195.95,197.30,195.86,196.61,2957900,196.61
2013-07-23,194.21,196.43,194.10,194.98,2863800,194.98

1962-01-09,552.00,563.00,552.00,556.00,491200,2.43
1962-01-08,559.50,559.50,545.00,549.50,544000,2.40
1962-01-05,570.50,570.50,559.00,560.00,363200,2.44
1962-01-04,577.00,577.00,571.00,571.25,256000,2.49
1962-01-03,572.00,577.00,572.00,577.00,288000,2.52
1962-01-02,578.50,578.50,572.00,572.00,387200,2.50
```

The second example involves downloading the Russell 3000 time series. At http://www.russell.com/indexes/data/us_equity/russell_us_index_values.asp, find the appropriate time series, then click on **Download File** under **Historical**. The download file will be a CSV file, the first few lines of which are shown as follows:

```
"Index Name","Date","Value Without Dividends","Value With Dividends"
"Russell 3000® Index","06/01/1995",555.15,1034.42
"Russell 3000® Index","06/02/1995",555.15,1034.56
"Russell 3000® Index","06/05/1995",558.72,1041.21
"Russell 3000® Index","06/06/1995",558.50,1041.04
"Russell 3000® Index","06/07/1995",556.45,1037.21
"Russell 3000® Index","06/08/1995",555.83,1036.18
"Russell 3000® Index","06/09/1995",551.66,1028.41
```

```
"Russell 3000® Index", "06/12/1995", 554.61, 1033.96
```

```
"Russell 3000® Index", "06/13/1995", 559.74, 1043.93
```

```
"Russell 3000® Index", "06/14/1995", 560.19, 1044.86
```

Retrieving data to our programs

To feed data to our programs, we need to understand how to input data. Since the data courses vary, we introduce several ways to input data, such as from clipboard, Yahoo! Finance, an external text or CSV file, a web page, and a MATLAB dataset.

Inputting data from the clipboard

In our everyday lives, we use Notepad, Microsoft Word, or Excel to input data. One of the widely used functionalities is copy and paste. The `pd.read_clipboard()` function contained in Pandas mimics this operation. For example, we type the following contents on Notepad:

```
x y
1 2
3 4
5 6
```

Then, highlight these entries, right-click on it, copy and paste in the Python console, and run the following two lines:

```
>>>import pandas as pd
>>>data=pd.read_clipboard()
>>>data
   x  y
1  1  2
3  3  4
5  5  6
```

This is true for copying data from Microsoft Word and Excel.

Retrieving historical price data from Yahoo! Finance

The following simple program has just five lines, and we can use it to retrieve DELL's historical prices data from Yahoo! Finance:

```
>>>from matplotlib.finance import quotes_historical_yahoo
>>>ticker='DELL'
>>>begdate=(2013,1,1)
>>>enddate=(2013,11,9)
>>>p=quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
```

To further check the variable called `p`, we could apply the `type()` and `size()` functions. To view a few lines, we print the first and the last few lines on screen. The `p[0]` array index is referred to in the first observation, while `p[-1]` is for the last one:

```
>>>type(p)
<class 'numpy.core.records.recarray'>
>>>size(p)
209
```

```
>>> p[0:3]
rec.array([(datetime.date(2013, 1, 2), 2013, 1, 2, 734870.0, 10.146067415730338, 10.5, 10.5196
62921348315, 10.126404494382024, 26421700.0, 10.5),
          (datetime.date(2013, 1, 3), 2013, 1, 3, 734871.0, 10.435557586837295, 10.75, 11.13322669
1042047, 10.406078610603291, 38131300.0, 10.75),
          (datetime.date(2013, 1, 4), 2013, 1, 4, 734872.0, 10.740692798541476, 10.78, 10.87826800
3646307, 10.622771194165907, 18706400.0, 10.78)],
          dtype=[('date', 'O'), ('year', '<i2'), ('month', 'i1'), ('day', 'i1'), ('d', '<f8'), ('op
en', '<f8'), ('close', '<f8'), ('high', '<f8'), ('low', '<f8'), ('volume', '<f8'), ('aclose', '
<f8')])
>>>
```

The output tells us that the data type of the output is an array. By looking at these two days, we realize that the dataset is sorted with the oldest date as the first observation. This is opposite to the date order where the first date is the latest date, shown on Yahoo! Finance. For the array, we have seven variables: `date`, `open`, `close`, `high`, `low`, `volume`, and `aclose`. The `aclose` variable is the adjusted closing price, that is, a closing price adjusted for stock split and distribution such as dividends.

Inputting data from a text file

When importing data from an external file, Pandas has many functions, such as `read_table()`, `read_fwf()`, `read_hdf()`, and `io()`. There is no way that we could discuss each of them in this chapter. Thus, we focus on a few widely used functions. Assume that our input dataset is the Fama-French monthly factors. To download the time series manually, we go to Prof. French's data library at http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html. Click on **Fama/French Factors**, and download it. Then, unzip the file, delete the annual part, and then name the text file `ff_monthly.txt`. The data items start from the fifth line, and the fourth line is a header; refer to the following lines:

```
This file was created by CMPT_ME_BEME_RETS using the 201209 CRSP
database.
```

```
The 1-month TBill return is from Ibbotson and Associates, Inc.
```

```
[this is a blank line]
```

	Mkt-RF	SMB	HML	RF
192607	2.62	-2.16	-2.92	0.22
192608	2.56	-1.49	4.88	0.25
192609	0.36	-1.38	-0.01	0.23

We could use `read_table()` to retrieve these three factors as shown in the following code:

```
>>>import pandas as pd
>>>x=pd.read_table("c:/temp/ff_monthly.txt",skiprows=4)
```

After issuing `help(read_table)`, we could get more information about this function. The first several lines are shown as follows:

```
>>>import pandas as pd
>>>help(pd.read_table)
Help on function read_table in module pandas.io.parsers:
```

```
read_table(filepath_or_buffer, sep='\t', dialect=None, compression=None,
doublequote=True, escapechar=None, quotechar='"', quoting=0,
skipinitialspace=False, lineterminator=None, header='infer', index_
col=None, names=None, prefix=None, skiprows=None, skipfooter=None,
skip_footer=0, na_values=None, true_values=None, false_values=None,
delimiter=None, converters=None, dtype=None, usecols=None, engine='c',
delim_whitespace=False, as_reccarray=False, na_filter=True, compact_
ints=False, use_unsigned=False, low_memory=True, buffer_lines=None,
warn_bad_lines=True, error_bad_lines=True, keep_default_na=True,
thousands=None, comment=None, decimal='.', parse_dates=False, keep_
date_col=False, dayfirst=False, date_parser=None, memory_map=False,
nrows=None, iterator=False, chunksize=None, verbose=False, encoding=None,
squeeze=False)
```

Read general delimited file into DataFrame

Also supports optionally iterating or breaking of the file
into chunks.

The most important input variables for the `read_table()` function are `skiprows`, `sep`, `index_col`, and `doublequote`. Some of them will be discussed later in this chapter.

Inputting data from an Excel file

Assume that we have an Excel file with just two observations with a file named `test.xlsx` saved under `C:\temp\`. In addition, the name of the spreadsheet that contains the two lines of code shown after the screenshot is called `Sheet1`:

	A	B	C
1	1/1/2013	0.1	0.3
2	1/2/2013	0.2	0.4

```
>>>infile=pd.ExcelFile("c:/temp/test.xlsx")
>>>x=infile.parse('Sheet1',header=None)
>>>x
           0      1      2
0 2013-01-01 00:00:00  0.1  0.3
1 2013-01-02 00:00:00  0.2  0.4
>>>
```

Inputting data from a CSV file

The function we could use to read a CSV file could be `read_csv()`, or `read_table()`. Assume that we have the earlier data downloaded from Yahoo! Finance. The file's name is `ibm.csv`, and it is located at `C:\temp`. Input IBM's daily price data. The input file is `ibm.csv`, which we just downloaded from Yahoo! Finance. To see a few lines, we just type `f[1:2]` as shown in the following code:

```
>>>import pandas as pd
>>>f=pd.read_csv("c:\\temp\\ibm.csv")
>>>f[1:3]
```

	Date	Open	High	Low	Close	Volume	Adj Close
1	2013-07-25	196.30	197.83	195.66	197.22	3014300	197.22
2	2013-07-24	195.95	197.30	195.86	196.61	2957900	196.61

Note that in this program, to retrieve data from an external CSV file, we use the `pd.read_csv("c:\\temp\\ibm.csv")` command, which is equivalent to the `pd.read_csv("c:/temp/ibm.csv")` command.

Retrieving data from a web page

An easy way to retrieve a stock's price data directly from Yahoo! Finance is to use the `pd.read_csv()` function from a web page as shown in the following code:

```
>>>import pandas as pd
>>>x=pd.read_csv("http://chart.yahoo.com/table.csv?s=IBM")
>>>type(x)
<class 'pandas.core.frame.DataFrame'>
>>>
```

The last command in the previous code indicates that `x` is a `DataFrame`. If we just type `x`, we will get more information about this variable as shown in the following code:

```
>>>x
<class 'pandas.core.frame.DataFrame'>
Int64Index: 13072 entries, 0 to 13071
Data columns (total 7 columns):
Date          13072  non-null values
Open          13072  non-null values
High          13072  non-null values
Low           13072  non-null values
Close         13072  non-null values
```

```

Volume      13072  non-null values
Adj Close    13072  non-null values
dtypes: float64(5), int64(1), object(1)
>>>x[0:5]

```

	Date	Open	High	Low	Close	Volume	Adj Close
0	2013-12-04	175.37	177.50	175.16	175.74	5267400	175.74
1	2013-12-03	177.00	178.23	175.64	176.08	5864000	176.08
2	2013-12-02	179.46	179.59	177.12	177.48	4560000	177.48
3	2013-11-29	179.21	180.76	179.00	179.68	2870500	179.68
4	2013-11-27	177.83	180.18	177.82	178.97	4596500	178.97

```

>>>

```

The `read_csv()` function in Pandas is used to retrieve data from an external file. In the preceding example, we need just two variables: date and adjusted closing price. Since we have seven columns, of which date is the first and adjusted price is the last one, their column numbers are 0 and 6. The keyword, `usecols()`, could be used to achieve this:

```

>>>import pandas as pd
>>>url='http://chart.yahoo.com/table.csv?s=IBM'
>>>x=pd.read_csv(url,usecols=[0,6])
>>>x[0:5]

```

	Date	Adj Close
0	2013-12-04	175.74
1	2013-12-03	176.08
2	2013-12-02	177.48
3	2013-11-29	179.68
4	2013-11-27	178.97

```

>>>

```

Inputting data from a MATLAB dataset

First, from <http://canisius.edu/~yany/ibm.mat>, we download MATLAB data. Assume that the downloaded MATLAB dataset is saved under `C:\temp\`. We can use the `loadmat()` function of SciPy to load it as follows:

```

>>>from __future__ import print_function
>>>import scipy.io as sp
>>>matData = sp.loadmat('c:/temp/ibm.mat')

```


Several important functionalities

Here, we introduce several important functionalities that we are going to use in the rest of the chapters. The `Series()` function included in the Pandas module would help us to generate time series. When dealing with time series, the most important variable is date. This is why we explain the date variable in more detail. `Data.Frame` is used intensively in Python and other languages, such as R.

Using `pd.Series()` to generate one-dimensional time series

We could easily use the `pd.Series()` function to generate our time series; refer to the following example:

```
>>>import pandas as pd
>>>x = pd.date_range('1/1/2013', periods=252)
>>>data = pd.Series(randn(len(x)), index=x)
>>>data.head()
2013-01-01    0.776670
2013-01-02    0.128904
2013-01-03   -0.064601
2013-01-04    0.988347
2013-01-05    0.459587
Freq: D, dtype: float64
>>>data.tail()
2013-09-05   -0.167599
2013-09-06    0.530864
2013-09-07    1.378951
2013-09-08   -0.729705
2013-09-09    1.414596
Freq: D, dtype: float64
>>>
```

Using date variables

To better facilitate working with time series data, we introduce the `read_csv()` and `read_table()` functions in Pandas. In particular, users can use `parse_dates` or `date_parse` to designate a specific column as a date-time object. To use the first column as our index, we issue the following commands:

```
>>>import pandas as pd
>>>url='http://chart.yahoo.com/table.csv?s=IBM'
>>>x=pd.read_csv(url,index_col=0,parse_dates=True)
>>>x.head()
```

	Open	High	Low	Close	Volume	Adj Close
Date						
2013-11-29	179.21	180.76	179.00	179.68	2870500	179.68
2013-11-27	177.83	180.18	177.82	178.97	4596500	178.97
2013-11-26	178.67	178.94	177.31	177.31	5756000	177.31
2013-11-25	180.25	180.75	177.82	178.94	7161900	178.94
2013-11-22	183.50	184.99	179.92	181.30	7610200	181.30

Using the DataFrame

The first example generates a column as follows:

```
>>>import pandas as pd
>>>df=pd.DataFrame(randn(8, 1), columns = ['A'], dtype = 'float32')
>>>df
```

	A
0	-0.581377
1	-1.790758
2	-0.418108
3	1.122045
4	-0.402717
5	0.694823
6	0.035632
7	0.919457

```
>>>
```

When running the previous code, we will not get the same results since the `np.random.seed()` function is not used. When we use the `read_csv()` or `read_table()` functions to input data from an external file with a text format, the data type is also `DataFrame` as shown in the following example:

```
>>>import pandas as pd
>>>index = pd.date_range('1/1/2013', periods=8)
>>>df = pd.DataFrame(randn(8, 3), index=index, columns=['A', 'B', 'C'])
>>>df
```

	A	B	C
2013-01-01	-1.185345	-0.422447	-0.610870
2013-01-02	-1.507653	-0.295807	-0.636771
2013-01-03	1.686858	-2.013024	-0.980905
2013-01-04	0.372631	-1.580834	0.515045
2013-01-05	-0.322729	-0.677587	-1.053555
2013-01-06	-0.518918	-0.952527	0.000124
2013-01-07	0.482760	2.049442	1.833976
2013-01-08	0.313321	0.162334	0.662253

Assume that we are only interested in two variables from IBM's historical daily data from Yahoo! Finance: date and adjusted close price. In addition, we plan to use the date variable as our index. We have the following Python program to accomplish these requirements:

```
>>>import pandas as pd
>>>x=pd.read_csv('http://chart.yahoo.com/table.
csv?s=IBM',usecols=[0,6],index_col=0)
>>>type(x)
<class 'pandas.core.frame.DataFrame'>
>>>x.head()
```

Date	Adj Close
2013-11-21	184.13
2013-11-20	185.19
2013-11-19	185.25
2013-11-18	184.47
2013-11-15	183.19

To find more information about the `pd.DataFrame()` function included in Pandas, we can issue `help(pd.DataFrame)` as follows:

```
>>>help(pd.DataFrame)
Help on class DataFrame in module pandas.core.frame:

class DataFrame(pandas.core.generic.NDFrame)
    Two-dimensional size-mutable, potentially heterogeneous tabular
    datastructure with labeled axes (rows and columns). Arithmetic
    operations align on both row and column labels. Can be thought of as a
    dict-like container for Series objects. The primary pandas data structure

    Parameters
    -----
    data : numpy ndarray (structured or homogeneous), dict, or DataFrame
    Dict can contain Series, arrays, constants, or list-like objects
    index : Index or array-like
        Index to use for resulting frame. Will default to np.arange(n) if
        no indexing information part of input data and no index provided
    columns : Index or array-like Will default to np.arange(n) if not
    column labels provided
    dtype : dtype, default None Data type to force, otherwise infer
    copy : boolean, default False
        Copy data from inputs. Only affects DataFrame / 2d ndarray input
```

Return estimation

If we have price data, we have to calculate returns. In addition, sometimes we have to convert daily returns to weekly or monthly, or convert monthly returns to quarterly or annual. Thus, understanding how to estimate returns and their conversion is vital. Assume that we have four prices and we choose the first and last three prices as follows:

```
>>>import numpy as np
>>>p=np.array([1,1.1,0.9,1.05])
```

It is important how these prices are sorted. If the first price happened before the second price, we know that the first return should be $(1.1-1)/1=10\%$. Next, we learn how to retrieve the first $n-1$ and the last $n-1$ records from an n -record array. To list the first $n-1$ prices, we use `p[:-1]`, while for the last three prices we use `p[1:]` as shown in the following code:

```
>>>print(p[:-1])
>>>print(p[1:])
[ 1.    1.1  0.9]
[ 1.1   0.9  1.05]
```

To estimate returns, use the following code:

```
>>>ret=(p[1:]-p[:-1])/p[:-1]
>>>print ret
[ 0.1          -0.18181818  0.16666667]
```

However, if the prices are arranged in the reverse order, for example, the first one is the latest price and the last one is the oldest price, then we have to estimate returns in the following ways:

```
>>>ret=(p[:-1]-p[1:])/p[1:]
>>>print ret
[-0.09090909  0.22222222 -0.14285714]
>>>
```

The following code shows how to download daily price data from Yahoo! Finance and estimate daily returns:

```
>>>from matplotlib.finance import quotes_historical_yahoo
>>>ticker='IBM'
>>>begdate=(2013,1,1)
>>>enddate=(2013,11,9)
>>>x = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
>>>ret=(x.aclose[1:]-x.aclose[:-1])/x.aclose[:-1]
```

The first line uploads a function from `matplotlib.finance`. We define the beginning and ending dates using a tuple data type. The downloaded historical daily price data is assigned to `x`. To verify that our returns are correctly estimated, we can print a few prices to our screens. Then, we could manually verify one or two return values as shown in the following code:

```
>>>x.date[0:3]
array([datetime.date(2013, 1, 2), datetime.date(2013, 1, 3),
       datetime.date(2013, 1, 4)], dtype=object)
>>>x.aclose[0:3]
array([ 192.61,  191.55,  190.3 ])
>>>ret[0:2]
array([-0.00550335, -0.00652571])
>>>(191.55-192.61)/192.61
-0.005503348735787354
>>>
```

Yes, the last result confirms that our first return is correctly estimated.

Converting daily returns to monthly returns

Sometimes, we need to convert daily returns to monthly or annual returns. Here is our procedure. First, we estimate the daily log returns. We then take a summation of all daily log returns within each month to find out the corresponding monthly log returns. The final step is to convert a log monthly return to a monthly percentage return. Assume that we have the price data of $p_0, p_1, p_2, \dots, p_{20}$, where p_0 is the last trading price of the last month, p_1 is the first price of this month, and p_{20} is the last price of this month. Thus, this month's percentage return is given as follows:

$$R_{monthly} = \frac{P_{20} - P_0}{P_0} \quad (2)$$

The monthly log return is defined as follows:

$$\log_return_{monthly} = \log\left(\frac{P_{20}}{P_0}\right) \quad (3)$$

The relationship between monthly percentage and log return is given as follows:

$$R_{monthly} = \exp(\log_return) - 1 \quad (4)$$

The daily log return is defined similarly as follows:

$$\log_return_i^{daily} = \log\left(\frac{p_i}{p_{i-1}}\right) \quad (5)$$

Let's look at the following summation of log returns:

$$\log_return_{monthly} = \log\left(\frac{p_{20}}{p_0}\right) = \sum_{i=1}^{20} \log_return_i^{daily} \quad (6)$$

Based on the previous procedure, the following Python program converts daily returns into monthly returns:

```
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd
ticker='IBM'
begdate=(2013,1,1)
enddate=(2013,11,9)
x = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
logret = log(x.aclose[1:]/x.aclose[:-1])
yyyymm=[]
d0=x.date
for i in range(0,size(logret)):
    yyyymm.append(''.join([d0[i].strftime("%Y"),d0[i].strftime("%m")]))
y=pd.DataFrame(logret,yyyymm,columns=['ret_monthly'])
ret_monthly=y.groupby(y.index).sum()
```

In the preceding program, we download daily price data based on a given ticker and the beginning and ending dates. Since the closing price is adjusted for stock split and potential distribution, we use it to estimate log returns. The procedure to generate daily log returns instead of daily percentage returns is based on equation (6). Then, we generate a variable called `yyyymm`. To show a few of its observations, we have the following output:

```
>>>
>>> yyyymm[0:5]
['201301', '201301', '201301', '201301', '201301']
>>>
>>>
```

The objective of generating such a date variable is to use it for grouping. To generate this variable, we apply the `join()` function; refer to the following example. We use `d0[0]` as an illustration:

```
>>>d0[0]
datetime.date(2013, 1, 2)
>>>d0[0].strftime("%Y")
'2013'
>>>d0[0].strftime("%m")
'01'
>>>'.'.join([d0[0].strftime("%Y"),d0[0].strftime("%m")])
'201301'
>>>
```

After printing the monthly returns, we have the following values:

```
>>>ret_monthly
      ret_monthly
201301      0.043980
201302     -0.006880
201303      0.045571
201304     -0.061913
201305      0.050327
201306     -0.088366
201307      0.023441
```



```
201308    -0.057450
201309     0.013031
201310    -0.039109
201311     0.009602
>>>
```

Converting daily returns to annual returns

Similarly, we could convert daily returns to annual ones with the help of the following code:

```
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd
ticker='IBM'
begdate=(1990,1,1)
enddate=(2012,12,31)
x=quotes_historical_yahoo(ticker,begdate,enddate,asobject=True,adjusted=True)
logret = log(x.aclose[1:]/x.aclose[:-1])
date=[]
d0=x.date
for i in range(0,size(logret)):
    date.append(d0[i].strftime("%Y"))
y=pd.DataFrame(logret,date,columns=['ret_annual'])
ret_annual=exp(y.groupby(y.index).sum())-1
```

The first and last several observations of the dataset are given, hereby using `head()` and `tail()` functions of Pandas as follows:

```
>>>ret_annual.head()
      ret_annual
1990    0.197897
1991   -0.157460
1992   -0.411765
1993    0.187500
```

```

1994      0.300877
>>>ret_annual.tail()
      ret_annual
2008    -0.150799
2009     0.546150
2010     0.134804
2011     0.284612
2012     0.045457

```

Merging datasets by date

Assume that we are interested in estimating the market risk (beta) for IBM using daily data. The following is the program we can use to download IBM's price, market return, and risk-free interest rate since we need them to run a **capital asset pricing model** (CAPM):

```

from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd
ticker='IBM'
begdate=(2013,10,1)
enddate=(2013,11,9)
x = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
k=x.date
date=[]
for i in range(0,size(x)):
    date.append(''.join([k[i].strftime("%Y"),k[i].strftime("%m"),k[i].
strftime("%d")]))
x2=pd.DataFrame(x['aclose'],np.array(date,dtype=int64),columns=[ticker+'_
adjClose'])
ff=load('c:/temp/ffDaily.pickle')
final=pd.merge(x2,ff,left_index=True,right_index=True)

```

A part of the output is given as follows:

```
>>> final.head()
      IBM_adjClose  Mkt_Rf      SMB      HML  Rf
20131001      184.37  0.0091  0.0039 -0.0013  0
20131002      182.97 -0.0010 -0.0036  0.0004  0
20131003      181.88 -0.0087 -0.0017  0.0020  0
20131004      182.12  0.0074  0.0000  0.0010  0
20131007      180.05 -0.0095 -0.0026  0.0006  0
>>> final.tail()
      IBM_adjClose  Mkt_Rf      SMB      HML  Rf
20131025      174.95  0.0033 -0.0035  0.0022  0
20131028      175.44  0.0009 -0.0005 -0.0002  0
20131029      180.16  0.0054 -0.0016 -0.0015  0
20131030      178.21 -0.0061 -0.0072  0.0040  0
20131031      177.28 -0.0034 -0.0008 -0.0036  0
>>>
```

In the preceding output, there are two types of data for the five columns: price and returns. The first column is price while the rest are returns. This does not make any sense. However, we just show how to merge different time series by a date variable. To merge a stock return column with returns, we simply estimate the return of IBM before a merger.

Forming an n-stock portfolio

In the following program, we generate a dataset with three stocks in addition to S&P500:

```
import statsimport numpy as np
import pandas as pd
tickers=['IBM','dell','wmt']
final=pd.read_csv('http://chart.yahoo.com/table.csv?s=^GSPC',usecols=[0,6],index_col=0)
final.columns=['^GSPC']
for ticker in tickers:
    print ticker
    x = pd.read_csv('http://chart.yahoo.com/table.csv?s=ttt'.replace('ttt',ticker),usecols=[0,6],index_col=0)
    x.columns=[ticker]
    final=pd.merge(final,x,left_index=True,right_index=True)
```

To show the first and last few lines, we use the `head()` and `tail()` functions as follows:

```
>>>final.head()
          ^GSPC      IBM  dell  wmt
Date
2013-10-18  1744.50  172.85  13.83  75.71
2013-10-17  1733.15  173.90  13.85  75.78
2013-10-16  1721.54  185.73  13.85  75.60
2013-10-15  1698.06  183.67  13.83  74.37
2013-10-14  1710.14  185.97  13.85  74.68
>>>final.tail()
          ^GSPC      IBM  dell  wmt
Date
1988-08-23  257.09  17.38  0.08  2.83
1988-08-22  256.98  17.36  0.08  2.87
1988-08-19  260.24  17.67  0.09  2.94
1988-08-18  261.03  17.97  0.09  2.98
1988-08-17  260.77  17.97  0.09  2.98
>>>
```

T-test and F-test

In finance, T-test could be viewed as one of the most used statistical hypothesis tests in which the test statistic follows a student's t distribution if the null hypothesis is supported. We know that the mean for a standard normal distribution is zero. In the following program, we generate 1,000 random numbers from a standard distribution. Then, we conduct two tests: test whether the mean is 0.5, and test whether the mean is zero:

```
>>>from scipy import stats
>>>np.random.seed(1235)
>>>x = stats.norm.rvs(size=10000)
>>>print("T-value    P-value (two-tail)")
>>>print(stats.ttest_1samp(x,5.0))
>>>print(stats.ttest_1samp(x,0))
T-value    P-value (two-tail)
```

```
(array(-495.266783341032), 0.0)
(array(-0.26310321925083124), 0.79247644375164772)
>>>
```

For the first test, in which we test whether the time series has a mean of 0.5, we reject the null hypothesis since the T-value is 495.2 and the P-value is 0. For the second test, we accept the null hypothesis since the T-value is close to -0.26 and the P-value is 0.79. In the following program, we test whether the mean daily returns from IBM in 2013 is zero:

```
from scipy import stats
from matplotlib.finance import quotes_historical_yahoo
ticker='ibm'
begdate=(2013,1,1)
enddate=(2013,11,9)
p=quotes_historical_yahoo(ticker,begdate,enddate,asobject=True,
adjusted=True)
ret=(p.aclose[1:] - p.aclose[:-1])/p.aclose[:-1]
print('          Mean          T-value          P-value  ' )
print(round(mean(ret),5), stats.ttest_1samp(ret,0))

          Mean          T-value          P-value
(-0.00024, (array(-0.296271094280657), 0.76730904089713181))
```

From the previous results, we know that the average daily returns for IBM is 0.0024 percent. The T-value is -0.29 while the P-value is 0.77. Thus, the mean is statistically not different from zero.

Tests of equal means and equal variances

Next, we test whether two variances for IBM and DELL in 2013 are equal or not. The function called `sp.stats.bartlett` performs Bartlett's test for equal variances with a null hypothesis that all input samples are from populations with equal variances. The outputs are T-value and P-value:

```
import scipy as sp
from matplotlib.finance import quotes_historical_yahoo
begdate=(2013,1,1)
enddate=(2013,11,9)
def ret_f(ticker,begdate,enddate):
```

```

    p = quotes_historical_yahoo(ticker,begdate, enddate,asobject=True,adjusted=True)

    return((p.open[1:] - p.open[:-1])/p.open[:-1])
y=ret_f('IBM',begdate,enddate)
x=ret_f('DELL',begdate,enddate)
print(sp.stats.bartlett(x,y))
(5.1377132006045105, 0.023411467035559311)

```

With a T-value of 5.13 and a P-value of 2.3 percent, we conclude that these two stocks will have different variances for their daily stock returns in 2013 if we choose a significant level of 5 percent.

Testing the January effect

In this section, we use IBM's data to test the existence of the so-called January effect which states that stock returns in January are statistically different from those in other months. First, we collect the daily price for IBM from Yahoo! Finance. Then, we convert daily returns to monthly ones. After that, we classify all monthly returns into two groups: returns in January versus returns in other months. Finally, we test the equality of group means as shown in the following code:

```

from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import scipy as sp
from datetime import datetime
ticker='IBM'
begdate=(1962,1,1)
enddate=(2013,11,22)
x = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,adjusted=True)
logret = log(x.aclose[1:]/x.aclose[:-1])
date=[]
d0=x.date
for i in range(0,size(logret)):
    t1=''.join([d0[i].strftime("%Y"),d0[i].strftime("%m"),"01"])
    date.append(datetime.strptime(t1,"%Y%m%d"))
y=pd.DataFrame(logret,date,columns=['logret'])
retM=y.groupby(y.index).sum()

```

```
ret_Jan=retM[retM.index.month==1]
ret_others=retM[retM.index.month!=1]
print(sp.stats.bartlett(ret_Jan.values,ret_others.values))
(1.1592293088621082, 0.28162543233634485)
>>>
```

Since the T-value is 1.16 and P-value is 0.28, we conclude that there is no January effect if we use IBM as an example and choose a 5 percent significant level. A word of caution: we should not generalize this result since it is based on just one stock. In terms of the weekday effect, we could apply the same procedure to test its existence.

Many useful applications

In this section, we discuss many issues, such as the 52-week high and low trading strategy, estimating the Roll (1984) spread, Amihud (2002) illiquidity measure, Pastor and Stambaugh (2003) liquidity measure, and CAPM, and running a Fama-French three-factor model, Fama-Macbeth regression, rolling beta, and VaR.

52-week high and low trading strategy

Some investors/researchers argue that we could adopt a 52-week high and low trading strategy by taking a long position if today's price is close to the minimum price achieved in the past 52 weeks and taking an opposite position if today's price is close to its 52-week high. The following Python program presents this 52-week's range and today's position:

```
from matplotlib.finance import quotes_historical_yahoo
from datetime import datetime
from dateutil.relativedelta import relativedelta
ticker='IBM'
enddate=datetime.now()
begdate=enddate-relativedelta(years=1)
p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
x=p[-1]
y=np.array(p.tolist())[:,-1]
high=max(y)
low=min(y)
print("      Today,                Price      High      Low,    % from low ")
print(x[0], x[-1], high, low, round((x[-1]-low)/(high-low)*100,2))
```

The corresponding output is shown as follows:

```
>>> print("      Today,          Price      High  Low, % from low ")
      Today,          Price      High  Low, % from low
>>> print(x[0], x[-1], high, low, round((x[-1]-low)/(high-low)*100,2))
(datetime.date(2014, 2, 28), 185.16999999999999, 211.42, 171.0, 35.06)
>>>
```

According to the 52-week high and low trading strategy, we have more incentive to buy IBM's stock today.

Roll's model to estimate spread (1984)

Liquidity is defined as how quickly we can dispose of our asset without losing its intrinsic value. Usually, we use spread to represent liquidity. However, we need high-frequency data to estimate spread. Later in the chapter, we show how to estimate spread directly by using high-frequency data. To measure spread indirectly based on daily observations, Roll (1984) shows that we can estimate it based on the serial covariance in price changes as follows:

$$S = 2\sqrt{-cov(\Delta P_t, \Delta P_{t-1})} \quad (7A)$$

$$\%spread = \frac{S}{\bar{p}} \quad (7B)$$

Here, P_t is the closing price of a stock on day t , $\bar{p}$ is the average share price in the estimation period. One of the problems of the Roll spread is that for some stocks over certain periods, their covariance of price change is positive. For such cases, users set S as being equal to zero. For example, we could use the following code to estimate the covariance between ΔP_t and ΔP_{t-1} . The following Python code estimates Roll's spread for a given ticker, in this case, DELL, using the latest one year's 252 trading days' daily data from Yahoo! Finance:

```
from matplotlib.finance import quotes_historical_yahoo
ticker='IBM'
begdate=(2013,9,1)
enddate=(2013,11,11)
data= quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
p=data.aclose
d=diff(p)
```



```
cov_=cov(d[:-1],d[1:])
if cov_[0,1]<0:
    print("Roll spread for ", ticker, 'is', round(2*sqrt(-cov_[0,1]),3))
else:
    print("Cov is positive for ",ticker, 'positive', round(cov_[0,1],3))
```

The corresponding output is shown as follows:

```
...
('Roll spread for ', 'IBM', 'is', 1.145)
>>>
```

Thus, during that period, Roll's spread for IBM is 1.145. The major assumption for Roll's model is that the covariance between ΔP_t and ΔP_{t-1} is negative. When its value is positive, Roll's model would fail. In a real world, it is true for many cases. Usually, practitioners adopt two approaches: when the spread is negative, we just ignore those cases or use other methods to estimate spread. The second approach is to add a negative sign in front of a positive covariance.

Amihud's model for illiquidity (2002)

According to Amihud (2002), liquidity reflects the impact of order flow on price. His illiquidity measure is defined as follows:

$$illiq_t = \frac{|R_t|}{P_t * V_t} \quad (8)$$

Here, R_t is the daily return at day t , P_t is closing price at t , and V_t is the daily dollar trading volume at t . Since the illiquidity is the reciprocal of liquidity, the lower the illiquidity value, the higher the liquidity of the underlying security. First, let us look at an item-by-item division:

```
>>>x=np.array([1,2,3],dtype='float')
>>>y=np.array([2,2,4],dtype='float')
>>>np.divide(x,y)
array([ 0.5 ,  1.   ,  0.75])
>>>
```

In the following code, we estimate Amihud's illiquidity for IBM based on trading data in October 2013. The value is 1.165×10^{-11} . It seems that this value is quite small. Actually, the absolute value is not important; the relative value matters. If we estimate the illiquidity for DELL over the same period, we would find a value of 0.638×10^{-11} . Since 1.165 is greater than 0.638, we conclude that IBM is less liquid than DELL. This correlation is represented in the following code:

```
import numpy as np
import statsmodels.api as sm
from matplotlib.finance import quotes_historical_yahoo
begdate=(2013,10,1)
enddate=(2013,10,30)
ticker='IBM' #WMT
data= quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
p=np.array(data.aclose)
dollar_vol=np.array(data.volume*p)
ret=np.array((p[1:] - p[:-1])/p[1:])
illiq=mean(np.divide(abs(ret),dollar_vol[1:]))
print("Aminud illiq=", illiq)
('Aminud illiq=', 1.1650681670001537e-11)
```

Pastor and Stambaugh (2003) liquidity measure

Based on the methodology and empirical evidence in Campbell, Grossman, and Wang (1993), Pastor and Stambaugh (2003) designed the following model to measure individual stock's liquidity and the market liquidity:

$$y_t = \alpha + \beta_1 x_{1,t-1} + \beta_2 x_{2,t-1} + \epsilon_t \quad (9)$$

Here, y_t is the excess stock return, $R_t - R_{f,t}$, on day t , R_t is the return for the stock, $R_{f,t}$ is the risk-free rate, $x_{1,t}$ is the market return; $x_{2,t}$ is the signed dollar trading volume ($x_{2,t} = \text{sign}(R_t - R_{f,t}) * p_t * \text{volume}_t$), p_t is the stock price, and volume_t is the trading volume. The regression is run based on daily data for each month. In other words, for each month, we get one β_2 that is defined as the liquidity measure for individual stock.

The following code estimates the liquidity for IBM. First, we download the IBM and S&P500 daily price data, estimate their daily returns, and merge them as follows:

```
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd
import statsmodels.api as sm

ticker='IBM'
begdate=(2013,1,1)
enddate=(2013,1,31)

data = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)

ret = (data.aclose[1:]-data.aclose[:-1])/data.aclose[:-1]
dollar_vol=np.array(data.aclose[1:])*np.array(data.volume[1:])
date=[]
d0=data.date
for i in range(0,size(ret)):
    date.append(''.join([d0[i].strftime("%Y"),d0[i].strftime("%m"),d0[i].
strftime("%d")]))
tt=pd.DataFrame(ret,np.array(date,dtype=int64),columns=['ret'])
tt2=pd.DataFrame(dollar_vol,np.array(date,dtype=int64),columns=['dollar_vol'])
ff=load('c:/temp/ffDaily.pickle')
tt3=pd.merge(tt,tt2,left_index=True,right_index=True)
final=pd.merge(tt3,ff,left_index=True,right_index=True)
y=final.ret[1:]-final.Rf[1:]
x1=final.Mkt_Rf[:-1]
x2=sign(np.array(final.ret[:-1]-final.Rf[:-1]))*np.array(final.dollar_vol[:-1])
x3=[x1,x2]
n=size(x3)
x=np.reshape(x3,[n/2,2])
x=sm.add_constant(x)
results=sm.OLS(y,x).fit()
print results.params
```

In the previous program, y is IBM's excess return at time $t+1$, $x1$ is the market excess return at time t , and $x2$ is the signed dollar trading volume at time t . The coefficient before $x2$ is Pastor and Stambaugh's liquidity measure. The corresponding output is given as follows:

```
const    2.699831e-03
x1       -1.297664e-13
x2       5.837434e-12
dtype: float64
>>>
```

Assume that we are interested in estimating the market risk (beta) for IBM using daily data downloaded from Yahoo! Finance. The beta is defined by the following linear regression:

$$R_{i,t} = R_f + \beta_i (R_{mkt,t} - R_{f,t}) + \epsilon_t, \quad (10)$$

Here, $R_{i,t}$ is the stock return for stock i , R_f is the risk-free rate, $R_{mkt,t}$ is the market return, and β_i is the beta for stock i . Since the impact of the risk-free rate is quite small on the beta estimation, we could use the following formula for an approximation:

$$R_{i,t} = \alpha + \beta_i R_{mkt,t} + \epsilon_t \quad (11)$$

The following Python program is used to download the IBM and S&P500 daily price data and estimate IBM's beta in 2013:

```
import numpy as np
import numpy as np
import statsmodels.api as sm
from matplotlib.finance import quotes_historical_yahoo
def ret_f(ticker,begdate, enddate):
    p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
    return((p.aclose[1:] - p.aclose[:-1])/p.aclose[:-1])
begdate=(2013,1,1)
enddate=(2013,11,9)
```

```
y=ret_f('IBM',begdate,enddate)
x=ret_f('^GSPC',begdate,enddate)
x=sm.add_constant(x)
model=sm.OLS(y,x)
results=model.fit()
print results.summary()
```

In the following program, we use a module called `matplotlib`, which is discussed in the previous chapter. In particular, the function called `quote_historical_yahoo()` is used to retrieve the daily price data from Yahoo! Finance for IBM and S&P500 (with a ticker symbol `^GSPC`). For the return estimate, we use the `adjust close`, that is, `aclose`. For the formula to estimate returns, we have `p.aclose[:-1]` and `p.aclose[1:]`; refer to the following code:

```
>>>x=np.array([1,2,3,4,10])
>>>x[1:]
array([ 2,  3,  4, 10])
>>>x[:-1]
array([1, 2, 3, 4])
>>>
```

The output is shown as follows:

OLS Regression Results						
=====						
Dep. Variable:	y	R-squared:	0.190			
Model:	OLS	Adj. R-squared:	0.187			
Method:	Least Squares	F-statistic:	50.32			
Date:	Tue, 03 Dec 2013	Prob (F-statistic):	1.88e-11			
Time:	13:19:56	Log-Likelihood:	672.76			
No. Observations:	216	AIC:	-1342.			
Df Residuals:	214	BIC:	-1335.			
Df Model:	1					
=====						
	coef	std err	t	P> t	[95.0% Conf. Int.]	

const	-0.0009	0.001	-1.237	0.217	-0.002	0.001
x1	0.7412	0.104	7.094	0.000	0.535	0.947
=====						
Omnibus:	202.221	Durbin-Watson:	1.819			
Prob(Omnibus):	0.000	Jarque-Bera (JB):	7283.884			
Skew:	-3.378	Prob(JB):	0.00			
Kurtosis:	30.635	Cond. No.	142.			
=====						

From the output, we know that beta for IBM is 0.74, which means that if the market risk premium increases by 1 percent, then IBM's risk premium would increase by 0.74 percent. In total, we have 216 observations used in 2013. The adjusted R^2 is 18.7 percent. In addition, careful readers would find that we could get more information, such as Durbin-Watson and Jarque-Bera, among others.

Before discussing how to run a Fama-French three-factor model, we show how to save the Fama-French as a dataset with a special format. Any `Pandas` object has a saving method, which uses Python's `cPickle` module to save data structures to a designated name under a directory as shown in the following code:

```
>>>import pandas as pd
>>>import numpy as np
>>>np.random.seed(1234)
>>>a = pd.DataFrame(randn(6,5))
>>>a.to_pickle('c:/temp/a.pickle')
>>>k=load("c:/temp/a.pickle")
```

In the preceding program, `np.random.seed(1234)` is not needed if we plan to generate any set of random numbers with 6 rows and 5 columns. Its usage will guarantee that we can generate the same set of random numbers irrespective of the number of times we run the preceding code. The values of the following output would be the same if anyone runs the preceding code. In addition to this, the extension of the output file need not necessarily be `.pickle`, that is, any extension is fine (even no extension is fine):

```
>>>print(k)
>>>
```

	0	1	2	3	4
0	0.471435	-1.190976	1.432707	-0.312652	-0.720589
1	0.887163	0.859588	-0.636524	0.015696	-2.242685
2	1.150036	0.991946	0.953324	-2.021255	-0.334077
3	0.002118	0.405453	0.289092	1.321158	-1.546906
4	-0.202646	-0.655969	0.193421	0.553439	1.318152
5	-0.469305	0.675554	-1.817027	-0.183109	1.058969

```
>>>
```

Fama-French three-factor model

The Fama-French three-factor model could be viewed as a natural extension of CAPM, which is actually a single factor model. The IBM return is defined as follows:

$$R_{IBM} = R_f + \beta_m (R_m - R_f) + \beta_{SMB} * SMB + \beta_{HML} * HML + \varepsilon_t, \quad (12)$$

Here, R_{IBM} is the IBM return, R_f is the risk-free return, R_m is the market return, SMB is the portfolio return of small stocks minus returns of big stocks, and HML is the portfolio returns for high book-to-market value minus returns of low book-to-market value stocks. The following program retrieves the Fama-French monthly factors and generates a dataset with the pickle format:

```
>>>import pandas as pd
>>>file=open("c:/temp/ff_monthly.txt","r")
>>>data=file.readlines()
>>>f=[]
>>>index=[]
>>>for i in range(4,size(data)):
    t=data[i].split()
    index.append(int(t[0]))
    for j in range(1,5):
        k=float(t[j])
        f.append(k/100)
>>>n=len(f)
>>>f1=np.reshape(f, [n/4,4])
>>>ff=pd.DataFrame(f1,index=index,columns=['Mkt_Rf','SMB','HML','Rf'])
>>>ff.to_pickle("c:/temp/ffMonthly.pickle")
>>>ff.head()
      Mkt_Rf      SMB      HML      Rf
192607  0.0265 -0.0239 -0.0257  0.0022
192608  0.0259 -0.0127  0.0458  0.0025
192609  0.0037 -0.0125 -0.0009  0.0023
192610 -0.0345 -0.0002  0.0102  0.0032
192611  0.0243 -0.0024 -0.0063  0.0031
>>>ff.tail()
      Mkt_Rf      SMB      HML      Rf
201306 -0.0121  0.0123 -0.0045  0
```

```

201307  0.0565  0.0185  0.0079  0
201308 -0.0269  0.0028 -0.0246  0
201309  0.0376  0.0285 -0.0152  0
201310  0.0417 -0.0152  0.0139  0
>>>

```

Next, we show how to run a Fama-French three-factor regression using five-year monthly data, downloaded from Yahoo! Finance for IBM. The dataset for the Fama-French monthly dataset in the Pandas' pickle format can be downloaded from <http://www.canisius.edu/~yany/ffMonthly.pickle>:

```

from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd
import statsmodels.api as sm
ticker='IBM'
begdate=(2008,10,1)
enddate=(2013,11,30)
p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
logret = log(p.aclose[1:]/p.aclose[:-1])
date=[]
d0=p.date
for i in range(0,size(logret)):
date.append(''.join([d0[i].strftime("%Y"),d0[i].strftime("%m")]))

t=pd.DataFrame(logret,np.array(date,dtype=int64),columns=['ret'])
ret=exp(t.groupby(t.index).sum())-1
ff=load('c:/temp/ffMonthly.pickle')
final=pd.merge(ret,ff,left_index=True,right_index=True)
y=final.ret
x=final[['Mkt_Rf','SMB','HML']]
x=sm.add_constant(x)
results=sm.OLS(y,x).fit()
print results.params

```


In the preceding program, we use a few modules. The beginning date is October 1, 2008, and the ending date is November 9, 2013. After retrieving the daily price data, we estimate the daily return and then convert them to monthly ones. We upload the Fama-French monthly three-factors time series and the Pandas' pickle format. In the preceding program, the usage of `np.array(date, dtype=int64)` is to make both indices have the same data types. The corresponding output is shown as follows:

```
const    2.699831e-03
x1       -1.297664e-13
x2        5.837434e-12
dtype: float64
>>>
```

Fama-MacBeth regression

First, let's look at the OLS regression by using the `pd.ols` function as follows:

```
from datetime import datetime
import numpy as np
import pandas as pd
n = 252
np.random.seed(12345)
begdate=datetime(2013, 1, 2)
dateRange = pd.date_range(begdate, periods=n)
x0= pd.DataFrame(np.random.randn(n, 1), columns=['ret'], index=dateRange)
y0=pd.Series(np.random.randn(n), index=dateRange)
print pd.ols(y=y0, x=x0)
```

For the Fama-MacBeth regression, we have the following code:

```
from datetime import datetime
import numpy as np
import pandas as pd
n = 252
np.random.seed(12345)
begdate=datetime(2013, 1, 2)
dateRange = pd.date_range(begdate, periods=n)
def makeDataFrame():
    data=pd.DataFrame(np.random.randn(n,7), columns=['A', 'B', 'C', 'D', 'E', 'F', 'G'],
```

```

        index=dateRange)
    return data
data = {
    'A': makeDataFrame(),
    'B': makeDataFrame(),
    'C': makeDataFrame()
}
Y = makeDataFrame()
print(pd.fama_macbeth(y=Y,x=data))

```

Estimating rolling beta

In the following implementation of the `pd.ols` function, the window parameter, such as `window=252`, is for moving or rolling the window:

```

import numpy as np
import statsmodels.api as sm
from matplotlib.finance import quotes_historical_yahoo
def ret_f(ticker,begdate, enddate):
    p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
    return((p.aclose[1:] - p.aclose[0:-1])/p.aclose[:-1])
begdate=(1962,1,1)
enddate=(2013,11,9)
y0=pd.Series(ret_f('IBM',begdate,enddate))
x0=pd.Series(ret_f('^GSPC',begdate,enddate))
model = pd.ols(y=y0, x=x0, window=252)

```

We could view these beta values and the graph with the following code:

```

>>>model.beta.head()
           x  intercept
251  1.608007  -0.000650
252  1.610066  -0.000652
253  1.608572  -0.000706
254  1.609975  -0.000736
255  1.611035  -0.000673
>>>model.beta.tail()

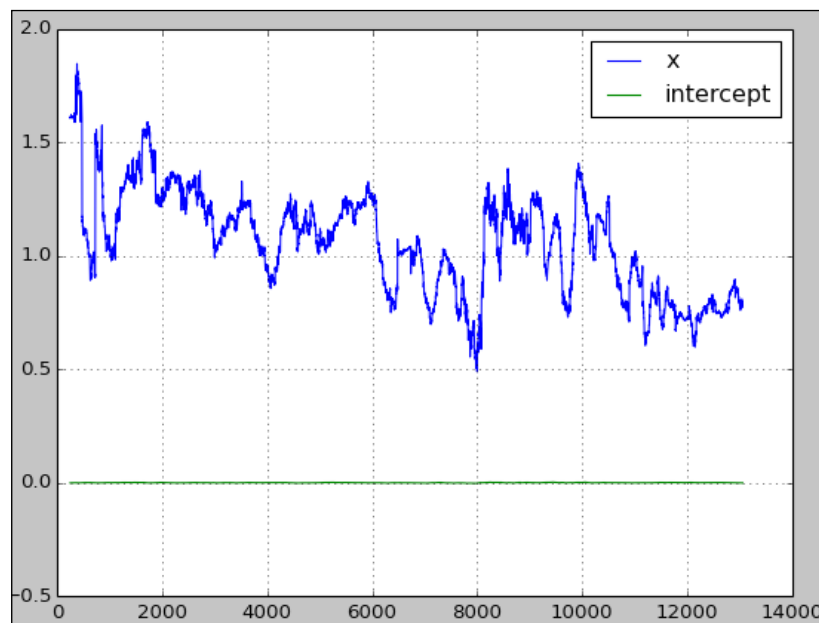
```

```
      x  intercept
13049  0.784624  -0.000856
13050  0.787177  -0.000911
13051  0.790030  -0.000870
13052  0.780330  -0.000814
13053  0.775992  -0.000867
>>>
```

To show the graph of the moving beta, we use the `plot()` function as follows:

```
>>>model.beta.plot()
```

The corresponding graph is shown as follows:



Usually, we care more about annual betas, instead of the previous betas for overlapping time periods. The following program estimates annual betas, which are another type of rolling betas:

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
from matplotlib.finance import quotes_historical_yahoo
```

```

def ret_f(ticker,begdate, enddate):
    p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
    return((p.aclose[1:] - p.aclose[:-1])/p.aclose[:-1])
begdate=(1962,1,1)
enddate=(2013,11,9)
y0=pd.Series(ret_f('IBM',begdate,enddate))
x0=pd.Series(ret_f('^GSPC',begdate,enddate))
d=quotes_historical_yahoo('^GSPC', begdate, enddate,asobject=True,
adjusted=True).date[0:-1]
lag_year=d[0].strftime("%Y")
y1=[]
x1=[]
beta=[]
index0=[]
for i in range(1,len(d)):
    year=d[i].strftime("%Y")
    if(year==lag_year):
        x1.append(x0[i])
        y1.append(y0[i])
    else:
        model=pd.ols(y=pd.Series(y1),x=pd.Series(x1))
        print(lag_year, round(model.beta[0],4))
        beta.append(model.beta[0])
        index0.append(lag_year)
        x1=[]
        y1=[]
        lag_year=year

```

The first several years' betas are given as follows:

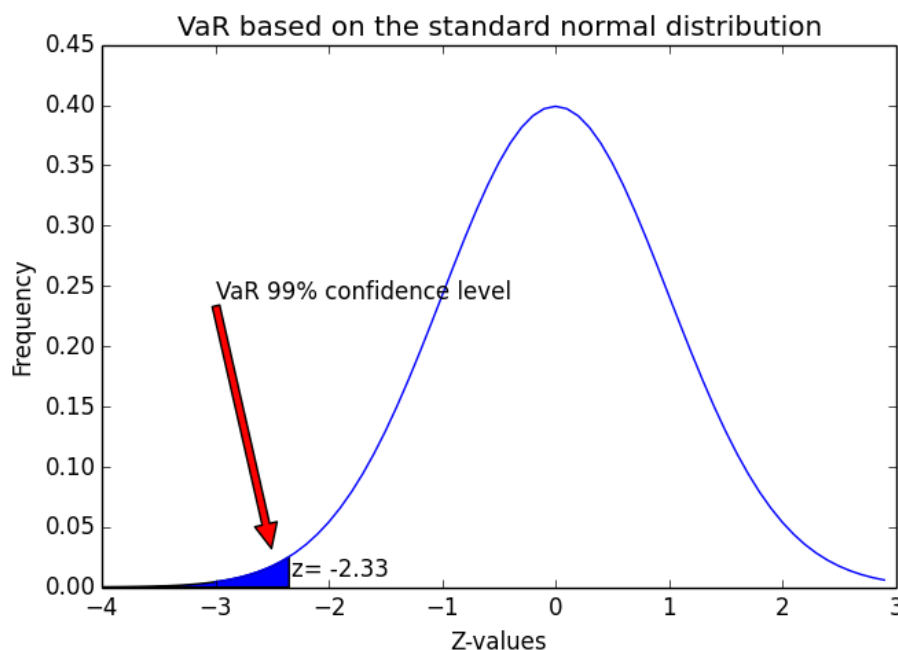
```

('1962', 1.6075)
('1963', 1.0976)
('1964', 1.4896)
('1965', 1.0463)
('1966', 1.2961)
('1967', 1.3819)
('1968', 1.5372)
('1969', 1.2412)

```

Understanding VaR

To evaluate the risk of a firm, a security, or a portfolio, various measures can be used, such as standard deviation, variance, beta, or Sharpe ratio. However, most CEOs prefer one simple number. In this case, one of the commonly used measures is VaR, which is defined as the maximum loss with a confidence level over a predetermined period. The following graph illustrates the concept of VaR based on a standard normal distribution:



Here are a few examples. I have 200 shares of DELL stocks. Today's value is \$2,942. The maximum loss tomorrow is \$239 with a 99 percent confidence level. Our mutual fund has a value of \$10 million today. The maximum loss over the next three months is 0.5 million with 90 percent confidence. The value of our bank is \$150 million. The VaR of our bank is \$10 million with 99 percent confidence over the next six months. The most commonly used parameters for VaR are 1 percent and 5 percent probabilities (99 percent and 95 percent confidence levels), and one-day and two-week horizons. Based on the assumption of normality, we have the following general form:

$$VaR_{period} = position * (\mu_{period} - z * \sigma_{period}) \quad (13)$$

Here, *position* is the current market value of our portfolio, μ_{period} is the expected period return, z is the cutoff point depending on a confidence level, and σ is the volatility. For a normal distribution, $z=2.33$ for a 99 percent confident level, and $z=1.64$ for a 95 percent confident level. When the time period is short, such as one day, we could ignore the impact of μ_{period} . Thus, we have the following simplest form:

$$VaR = p * z * \sigma \quad (14)$$

The following code shows the VaR for holding 50 shares of Wal-Mart stocks over a 10-day period:

```
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd
from scipy.stats import norm

n_shares=50                # input 1
confidence_level=0.99      # input 2
n_days=10                  # input 3
z=norm.ppf(confidence_level)
ticker='WMT'
begdate=(2012,1,1)
enddate=(2012,12,31)
x=quotes_historical_yahoo(ticker,begdate,enddate,asobject=True,adjusted=True)
ret = (x.aclose[1:]-x.aclose[:-1])/x.aclose[:-1]
position=n_shares*x.close[0]
VaR=position*z*std(ret)*sqrt(n_days)
print("Holding=",position, "VaR=", round(VaR,4), "in ", n_days, "Days")
('Holding=', 2890.0, 'VaR=', 218.2253, 'in ', 10, 'Days')
```

Today, the value of our holding is \$2,890. Our maximum loss is \$218.23 in the next 10 days with a confidence of 99 percent.

Constructing an efficient frontier

In finance, constructing an efficient frontier is always a challenging job. This is especially true with real-world data. In this section, we discuss the estimation of a variance-covariance matrix and its optimization, finding an optimal portfolio, and constructing an efficient frontier with stock data downloaded from Yahoo! Finance.

Estimating a variance-covariance matrix

When a return matrix is given, we could estimate its variance-covariance matrix. For a given set of weights, we could further estimate the portfolio variance. The formulae to estimate the variance and standard deviation for returns from a single stock are given as follows:

$$\bar{R} = \frac{\sum_{i=1}^n R_i}{n} \quad (15)$$

$$\left\{ \begin{array}{l} \sigma^2 = \frac{\sum_{i=1}^n (R_i - \bar{R})^2}{n-1} \\ \sigma = \sqrt{\sigma^2} \end{array} \right. \quad (16)$$

Here, R_i is the stock return for period i , $\bar{R}$ is their mean, and n is the number of the observations. For an n -stock portfolio, we have the following formulae:

$$R_{port} = \sum_{i=1}^n w_i R_i \quad (17)$$

The variance of a two-stock portfolio is given as follows:

$$\sigma_{port}^2 = w_1^2 \sigma_1^2 + w_2^2 \sigma_2^2 + 2w_1 w_2 \sigma_{1,2} = w_1^2 \sigma_1^2 + w_1^2 \sigma_1^2 + 2w_1 (1 - w_1) \rho \sigma_1 \sigma_2 \quad (18)$$

Here, $\sigma_{1,2}$ is the covariance between stocks 1 and 2, $\rho_{1,2}$ is the correlation coefficient between stocks 1 and 2. The covariance is defined as follows:

$$\sigma_{1,2} = \frac{\sum_{i=1}^n (R_{1,i} - \bar{R}_1)(R_{2,i} - \bar{R}_2)}{n-1} \quad (19)$$

The variance of an n -stock portfolio is given as follows:

$$\sigma_{port}^2 = \sum_{i=1}^n \sum_{j=1}^n w_i w_j \sigma_{i,j} \quad \text{where, } \sigma_{i,i} = \sigma_i^2 \quad (20)$$

Assume that our return matrix is n by m , that is, n period with m stocks:

$$R = \begin{pmatrix} R_{1,1} & R_{1,2} & \dots & R_{1,m} \\ R_{2,1} & R_{2,2} & \dots & R_{2,m} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ R_{n,1} & R_{n,2} & \dots & R_{n,m} \end{pmatrix} \begin{pmatrix} R_{1,1} & R_{1,2} & \dots & R_{1,m} \\ R_{2,1} & R_{2,2} & \dots & R_{2,m} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ R_{n,1} & R_{n,2} & \dots & R_{n,m} \end{pmatrix} \quad (21)$$

$$w = (w_1, w_2, w_3, \dots, w_m) \quad (22)$$

For a matrix representation, our portfolio's expected return is given as follows:

$$E(R_{port}) = w * E(R) \quad (23)$$

Its corresponding portfolio variance will be:

$$\Sigma = \begin{pmatrix} \sigma_{1,1} & \dots & \sigma_{1,m} \\ \sigma_{2,1} & \dots & \sigma_{2,m} \\ \dots & \dots & \dots \\ \dots & \dots & \dots \\ \sigma_{n,1} & \dots & \sigma_{n,m} \end{pmatrix} \begin{pmatrix} \sigma_{1,1} & \dots & \sigma_{1,m} \\ \sigma_{2,1} & \dots & \sigma_{2,m} \\ \dots & \dots & \dots \\ \dots & \dots & \dots \\ \sigma_{n,1} & \dots & \sigma_{n,m} \end{pmatrix} \quad (24)$$

$$\sigma_{port}^2 = w * \Sigma * w' \quad (25)$$

Of course, a two-stock portfolio is just a special case of an n -stock portfolio. Again, if the values of the return matrix and the weight vector are given, we can estimate their variance-covariance matrix and portfolio variance as follows:

```
>>>import numpy as np
>>>ret=matrix(np.
rray([[0.1,0.2],[0.10,0.1071],[-
0.02,0.25],[0.012,0.028],[0.06,0.262],[0.14,0.115]]))
>>>print("return matrix", ret)
>>>covar_=ret.T*ret
```



```
>>>weight=matrix(np.array([0.4,0.6]))
>>>print ("weight vecot",weight)
>>>print (weight*covar_*weight.T)
```

The corresponding two outputs, for return matrix and portfolio variance, are given as follows:

```
>>> print("return matrix", ret)
('return matrix', matrix([[ 0.1   ,  0.2   ],
        [ 0.1   ,  0.1071],
        [-0.02  ,  0.25   ],
        [ 0.012 ,  0.028  ],
        [ 0.06  ,  0.262  ],
        [ 0.14  ,  0.115  ]]))
>>> print(weight*covar_*weight.T)
[[ 0.10555915]]
>>>
```

Optimization – minimization

In the following example, we minimize our objective function of y :

$$y = 2 + ax^2 \quad (26)$$

Obviously, we know that when x is 0, y is minimized. The Python code for minimization is as follows:

```
>>>from scipy.optimize import minimize
>>>def y_f(x):
    return (3+2*x**2)
>>>x0=100
>>>res = minimize(y_f,x0,method='nelder-mead',options={'xtol':1e-
8,'disp': True})
>>>print(res.x)
Optimization terminated successfully.
    Current function value: 3.000000
    Iterations: 37
    Function evaluations: 74
[ 0.]
>>>
```

The output shows that the function value is 3, and it is achieved by assigning x as 0.

Constructing an optimal portfolio

In finance, we are dealing with the trade-off between risk and return. One of the widely used criteria is the Sharpe ratio, which is defined as follows:

$$Sharpe = \frac{E(R) - R_f}{\sigma_p} \quad (27)$$

The following program would maximize the Sharpe ratio by changing the weights of the stock in the portfolio. We have several steps in the program: the input area is very simple, just several tickers in addition to the beginning and ending dates. Then, we define four functions: converting daily returns into annual ones, estimate a portfolio variance, estimate the Sharpe ratio, and estimate the n^{th} weight when $n-1$ weights are given:

```
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd
import scipy as sp
from scipy.optimize import fmin

# Step 1: input area
ticker=('IBM','WMT','C')    # tickers
begdate=(1990,1,1)          # beginning date
enddate=(2012,12,31)        # ending date
rf=0.0003                    # annual risk-free rate
```

In the second part of the program, we define a few functions: download data from Yahoo! Finance, estimate daily returns and convert them into annual ones, estimate portfolio variance, and estimate Sharpe ratio as shown in the following program:

```
# Step 2: define a few functions
# function 1:
def ret_annual(ticker,begdate,enddate):
    x=quotes_historical_yahoo(ticker,begdate,enddate,asobject=True,adjusted=True)
    logret = log(x.aclose[1:]/x.aclose[:-1])
    date=[]
```

```
d0=x.date
for i in range(0,size(logret)):
    date.append(d0[i].strftime("%Y"))
y=pd.DataFrame(logret,date,columns=[ticker])
return exp(y.groupby(y.index).sum())-1
# function 2: estimate portfolio variance
def portfolio_var(R,w):
    cor = sp.corrcoef(R.T)
    std_dev=sp.std(R,axis=0)
    var = 0.0
    for i in xrange(n):
        for j in xrange(n):
            var += w[i]*w[j]*std_dev[i]*std_dev[j]*cor[i, j]
    return var
# function 3: estimate Sharpe ratio
def sharpe(R,w):
    var = portfolio_var(R,w)
    mean_return=mean(R,axis=0)
    ret = sp.array(mean_return)
    return (sp.dot(w,ret) - rf)/sqrt(var)
# function 4: for given n-1 weights, return a negative sharpe ratio
def negative_sharpe_n_minus_1_stock(w):
    w2=sp.append(w,1-sum(w))
    return -sharpe(R,w2)          # using a return matrix here!!!!!!
```

Our major function would start from Step 3 as shown in the following code:

```
# Step 3: generate a return matrix (annual return)
n=len(ticker)          # number of stocks
x2=ret_annual(ticker[0],begdate,enddate)
for i in range(1,n):
    x_=ret_annual(ticker[i],begdate,enddate)
    x2=pd.merge(x2,x_,left_index=True,right_index=True)
# using scipy array format
R = sp.array(x2)
print('Efficient portfolio (mean-variance) :ticker used')
print(ticker)
```

```

print('Sharpe ratio for an equal-weighted portfolio')
equal_w=sp.ones(n, dtype=float) * 1.0 /n
print(equal_w)
print(sharpe(R,equal_w))
# for n stocks, we could only choose n-1 weights
w0= sp.ones(n-1, dtype=float) * 1.0 /n
w1 = fmin(negative_sharpe_n_minus_1_stock,w0)
final_w = sp.append(w1, 1 - sum(w1))
final_sharpe = sharpe(R,final_w)
print ('Optimal weights are ')
print (final_w)
print ('final Sharpe ratio is ')
print(final_sharpe)

```

From the following output, we know that if we use a naïve equal-weighted strategy, the Sharpe ratio is 0.63. However, the Sharpe ratio for our optimal portfolio is 0.67:

```

Efficient portfolio (mean-variance) :ticker used
('IBM', 'WMT', 'C')
Sharpe ratio for an equal-weighted portfolio
[ 0.33333333  0.33333333  0.33333333]
0.634645504708
Optimization terminated successfully.
      Current function value: -0.669702
      Iterations: 30
      Function evaluations: 58
Optimal weights are
[ 0.49713116  0.31047116  0.19239769]
final Sharpe ratio is
0.669701971388
>>>

```

Constructing an efficient frontier with n stocks

Constructing an efficient frontier is always one of the most difficult tasks for finance instructors since the task involves matrix manipulation and a constrained optimization procedure. One efficient frontier could vividly explain the Markowitz Portfolio theory. The following Python program uses five stocks to construct an efficient frontier:

```

from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd

```

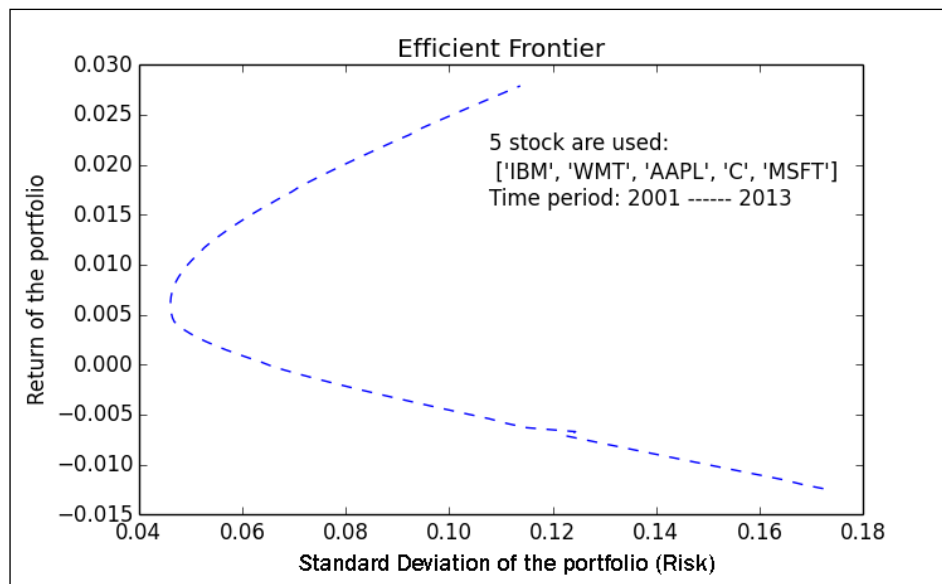
```
from numpy.linalg import inv, pinv
# Step 1: input area
begYear,endYear = 2001,2013
stocks=['IBM','WMT','AAPL','C','MSFT']
# Step 2: define a few functions
#         function 1
def ret_monthly(ticker):
    x = quotes_historical_yahoo(ticker, (begYear,1,1), (endYear,12,31), asobj
    ject=True,adjusted=True)
    logret=log(x.aclose[1:]/x.aclose[:-1])
    date=[]
    d0=x.date
    for i in range(0,size(logret)):
        date.append(''.join([d0[i].strftime("%Y"),d0[i].strftime("%m")]))
    y=pd.DataFrame(logret,date,columns=[ticker])
    return y.groupby(y.index).sum()
# function 2: objective function
def objFunction(W, R, target_ret):
    stock_mean=np.mean(R,axis=0)
    port_mean=np.dot(W,stock_mean)          # portfolio mean
    cov=np.cov(R.T)                          # var-cov matrix
    port_var=np.dot(np.dot(W,cov),W.T)       # portfolio variance
    penalty = 2000*abs(port_mean-target_ret)# penalty 4 deviation
    return np.sqrt(port_var) + penalty       # objective function
# Step 3: Generate a return matrix R
R0=ret_monthly(stocks[0])                   # starting from 1st stock
n_stock=len(stocks)                        # number of stocks
for i in xrange(1,n_stock):                # then merge with other stocks
    x=ret_monthly(stocks[i])
    R0=pd.merge(R0,x,left_index=True,right_index=True)
R=np.array(R0)
# Step 4: estimate optimal portfolio for a given return
out_mean,out_std,out_weight=[],[],[]
stockMean=np.mean(R,axis=0)
for r in np.linspace(np.min(stockMean), np.max(stockMean), num=100):
```

```

W = ones([n_stock])/n_stock      # starting from equal weights
b_ = [(0,1) for i in range(n_stock)] # bounds, here no short
c_ = ({'type':'eq', 'fun': lambda W: sum(W)-1. })#constraint
result=sp.optimize.minimize(objFunction,W,(R,r),method='SLSQP',constraints=c_, bounds=b_)
if not result.success:            # handle error
    raise BaseException(result.message)
out_mean.append(round(r,4))        # 4 decimal places
std_=round(np.std(np.sum(R*result.x,axis=1)),6)
out_std.append(std_)
out_weight.append(result.x)
# Step 4: plot the efficient frontier
title('Efficient Frontier')
xlabel('Standard Deviation of the portfolio (Risk)')
ylabel('Return of the portfolio')
figtext(0.5,0.75,str(n_stock)+' stock are used: ')
figtext(0.5,0.7,' '+str(stocks))
figtext(0.5,0.65,'Time period: '+str(begYear)+' ----- '+str(endYear))
plot(out_std,out_mean,'--')

```

The output graph is presented as follows:



Understanding the interpolation technique

Interpolation is a technique used quite frequently in finance. In the following example, we have to find NaN between 2 and 6. The `pd.interpolate()` function, for a linear interpolation, is used to fill in the two missing values:

```
>>>import pandas as pd
>>>import numpy as np
>>>x=pd.Series([1,2,np.nan,np.nan,6])
>>>x.interpolate()
0    1.000000
1    2.000000
2    3.333333
3    4.666667
4    6.000000
```

If the two known points are represented by the coordinates (x_0, y_0) and (x_1, y_1) , the linear interpolation is the straight line between these two points. For a value x in the interval of (x_0, x_1) , the value y along the straight line is given by the following formula:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0} \quad (28)$$

Solving this equation for y , which is the unknown value at x , gives the following result:

$$y = y_0 + \frac{(x - x_0)(y_1 - y_0)}{x_1 - x_0} \quad (29)$$

From the Yahoo! Finance bond page, we can get the following information:

Maturity	Yield	Yesterday	Last Week	Last Month
3 Month	0.05	0.05	0.04	0.03
6 Month	0.08	0.07	0.07	0.06
2 Year	0.29	0.29	0.31	0.33
3 Year	0.57	0.54	0.59	0.61
5 Year	1.34	1.32	1.41	1.39
10 Year	2.7	2.66	2.75	2.66
30 Year	3.8	3.78	3.85	3.72

Based on the tabular data, we have the following code:

```
>>>import numpy as np
>>>x=pd.Series([0.29,0.57,np.nan,1.34,np.nan,np.nan,np.nan,np.nan,2.7])
>>>y=x.interpolate()
>>>print y
0    0.290
1    0.570
2    0.955
3    1.340
4    1.612
5    1.884
6    2.156
7    2.428
8    2.700
dtype: float64
>>>
```

Outputting data to external files

In this section, we discuss several ways to save our data, such as saving data or estimating results to a text file, a binary file, and so on.

Outputting data to a text file

The following code will download IBM's daily price historical data and save it to a text file:

```
>>>from matplotlib.finance import quotes_historical_yahoo
>>>import re
>>>ticker='dell'
>>>outfile=open("c:/temp/dell.txt","w")
>>>begdate=(2013,1,1)
>>>enddate=(2013,11,9)
>>>p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
>>>x2= re.sub('[\(\)\{\}\.\<>a-zA-Z]','', x)
>>>outfile.write(x2)
>>>outfile.close()
```


Saving our data to a binary file

The following program first generates a simple array that has just three values. We save them to a binary file named `tmp.bin` at `C:\temp\`:

```
>>>import array
>>>import numpy as np
>>>outfile = "c:/temp/tmp.bin"
>>>fileobj = open(outfile, mode='wb')
>>>outvalues = array.array('f')
>>>data=np.array([1,2,3])
>>>outvalues.fromlist(data.tolist())
>>>outvalues.tofile(fileobj)
>>>fileobj.close()
```

Reading data from a binary file

Assume that we have generated a binary file called `C:\temp\tmp.bin` from the previous discussion. The file has just three numbers 1, 2, and 3. The following Python code is used to read them:

```
>>>import array
>>>infile=open("c:/temp/tmp.bin", "rb")
>>>s=infile.read()           # read all bytes into a string
>>>d=array.array("f", s)     # "f" for float
>>>print(d)
>>>infile.close()
```

Python for high-frequency data

High-frequency data is referred to second-by-second or millisecond-by-millisecond transaction and quotation data. The New York Stock Exchange's **TAQ (Trade and Quotation)** database is a typical example (<http://www.nyxdata.com/data-products/daily-taq>). The following program can be used to retrieve high-frequency data from Google Finance:

```
>>>import re, string
>>>import pandas as pd
>>>ticker='AAPL'           # input a ticker
>>>f1="c:/temp/ttt.txt"    # ttt will be replace with above sticker
```

```
>>>f2=f1.replace("ttt",ticker)
>>>outfile=open(f2,"w")
>>>path="http://www.google.com/finance/getprices?q=ttt&i=300&p=10d&f=d,o,h,l,c,v"
>>>path2=path.replace("ttt",ticker)
>>>df=pd.read_csv(path2,skiprows=8,header=None)
>>>df.to_csv(outfile,header=False,index=False)
>>>outfile.close()
```

In the preceding program, we have two input variables: `ticker` and `path`. After we choose `path` with an embedded variable called `ttt`, we replace it with our `ticker` using the `string.replace()` function. The first and last five lines are shown as follows using the `head()` and `tail()` functions:

```
>>>df.head()
   0      1      2      3      4      5
0  1  519.55  520.20  517.05  517.23  256716
1  2  519.20  520.40  518.84  519.59  202711
2  3  518.71  519.29  518.00  519.18  144928
3  4  519.11  519.60  518.08  518.76  108554
4  5  519.31  519.80  518.67  519.09  104715
>>>df.tail()
      0      1      2      3      4      5
748  898  525.450  525.500  524.990  525.140  113120
749  899  525.660  525.670  525.170  525.440   68422
750  900  525.460  525.680  525.370  525.660   10639
751  901  525.548  525.557  525.200  525.370      0
752  902  525.420  525.580  525.265  525.545      0
>>>
```

The related web page for the intra-day high-frequency data from Google is located at <http://www.google.com/finance/getprices?q=AAPL&i=300&p=10d&f=d,o,h,l,c,v> and its header (first 10) lines are given as follows:

```
EXCHANGE%3DNASDAQ
MARKET_OPEN_MINUTE=570
MARKET_CLOSE_MINUTE=960
INTERVAL=300
COLUMNS=DATE,CLOSE,HIGH,LOW,OPEN
DATA=
```

```
TIMEZONE_OFFSET=-300
a1383575400,521.2,521.35,521.07,521.1
1,522.48,522.58,519.75,521.37
2,519.44,522.89,518.81,522.49
3,520.36,520.98,519.1901,519.49
```

The objective of the following program is to add a timestamp:

```
import pandas as pd, numpy as np, datetime
ticker='AAPL'
path='http://www.google.com/finance/getprices?q=ttt&i=60&p=5d&f=d,o,h,l,c,v'
x=np.array(pd.read_csv(path.replace('ttt',ticker),skiprows=7,header=None))
date=[]
for i in range(0,len(x)):
    if x2[i][0][0]=='a':
        t= datetime.datetime.fromtimestamp(int(x2[i][0].replace('a','')))
        print ticker, t, x[i][1:]
        date.append(t)
    else:
        date.append(t+datetime.timedelta(minutes =int(x[i][0])))
final=pd.DataFrame(x,index=date)
final.columns=['a','Open','High','Low','Close','Vol']
del final['a']
final.to_csv('c:/temp/abc.csv'.replace('abc',ticker))
```

After running the program, we can observe the following output:

```
>>> runfile('C:/k/43750S_08_49_high_freq_my11_best.py', wdir=r'C:/k')
AAPL 2014-02-25 09:30:00 [529.25 529.565 529.08 529.42 63319L]
AAPL 2014-02-26 09:30:00 [524.0 524.0 523.61 523.61 55958L]
AAPL 2014-02-27 09:30:00 [517.35 517.35 517.14 517.14 95704L]
AAPL 2014-02-28 09:30:00 [528.97 529.08 528.53 529.08 161320L]
AAPL 2014-03-03 09:30:00 [523.41 523.47 523.0 523.42 89110L]
>>>
```

To view the first and last several lines, we could use the `head()` and `tail()` functions as follows:

```
>>>final.head()
```

	Open	High	Low	Close	Vol
2013-11-18 09:30:00	524.87	525.2402	524.762	524.99	80590
2013-11-18 09:31:00	525.08	525.5	524.76	524.82	79311
2013-11-18 09:32:00	525.75	525.8	525.01	525.03	43164
2013-11-18 09:33:00	526.445	526.58	525.65	525.75	81967
2013-11-18 09:34:00	526.48	526.5899	526.05	526.5899	40671

```
>>>final.tail()
```

	Open	High	Low	Close	Vol
2013-11-22 15:57:00	519.53	519.56	519.39	519.39	35530
2013-11-22 15:58:00	519.43	519.56	519.4	519.53	36581
2013-11-22 15:59:00	519.52	519.54	519.41	519.43	50983
2013-11-22 16:00:00	519.8	519.85	519.49	519.52	482044
2013-11-22 16:01:00	519.8	519.8	519.8	519.8	0

Since the TAQ database is quite expensive, most of the potential readers could not access the data. Fortunately, we have a database called **TORQ (Trade, Order, Report, and Quotation)**. Thanks to Prof. Hasbrouck, the database could be downloaded from <http://people.stern.nyu.edu/jhasbrou/Research/WorkingPaperIndex.htm>. From the same web page, we could download the TORQ manual as well. Based on Prof. Hasbrouck's binary datasets, we generate a few corresponding datasets in the pickle format of Pandas. The **Consolidated Trade (CT)** dataset can be downloaded from <http://canisius.edu/~yany/TORQct.pickle>. After saving this dataset under `C:\temp`, we could issue the following two lines of Python code to retrieve it:

```
>>>import pandas as pd
>>>ct=load('c:/temp/TORQct.pickle')
```

To view the first and last couple of lines, we use the `head()` and `tail()` functions as follows:

```
>>>ct.head()
```

	date	time	price	siz	g127	tseq	cond	ex
symbol								
AC	19901101	10:39:06	13	100	0	1587		N
AC	19901101	10:39:36	13	100	0	0		M

```

AC      19901101  10:39:38      13  100      0      0      M
AC      19901101  10:39:41      13  100      0      0      M
AC      19901101  10:41:38      13  300      0  1591      N
>>>ct.tail()
      date      time  price    siz  gl27    tseq cond ex
symbol
ZNT      19910131  11:03:31  12.375   1000      0  237884      N
ZNT      19910131  12:47:21  12.500   6800      0  237887      N
ZNT      19910131  13:16:59  12.500  10000      0  237889      N
ZNT      19910131  14:51:52  12.500    100      0  237891      N
ZNT      19910131  14:52:27  12.500   3600      0      0      Z  T
>>>

```

Since the ticker is used as an index, we could list all unique index values to find out the names of stocks contained in the dataset as follows:

```

>>>import numpy as np
>>>unique(np.array(ct.index))
array(['AC', 'ACN', 'ACS', 'ADU', 'AL', 'ALL', 'ALX', 'AMD', 'AMN', 'AMO',
      'AR', 'ARX', 'ATE', 'AYD', 'BA', 'BG', 'BMC', 'BRT', 'BZF', 'CAL',
      'CL', 'CLE', 'CLF', 'CMH', 'CMI', 'CMY', 'COA', 'CP', 'CPC', 'CPY',
      'CU', 'CUC', 'CUE', 'CYM', 'CYR', 'DBD', 'DCN', 'DI', 'DLT', 'DP',
      'DSI', 'EFG', 'EHP', 'EKO', 'EMC', 'FBO', 'FDX', 'FFB', 'FLP',
      'FMI', 'FNM', 'FOE', 'FPC', 'FPL', 'GBE', 'GE', 'GFB', 'GLX', 'GMH',
      'GPI', 'GRH', 'HAN', 'HAT', 'HE', 'HF', 'HFI', 'HTR', 'IBM', 'ICM',
      'IEI', 'IPT', 'IS', 'ITG', 'KfV', 'KR', 'KWD', 'LOG', 'LPX', 'LUK',
      'MBK', 'MC', 'MCC', 'MCN', 'MDP', 'MNY', 'MO', 'MON', 'MRT', 'MTR',
      'MX', 'NI', 'NIC', 'NNP', 'NSI', 'NSO', 'NSP', 'NT', 'OCQ', 'OEH',
      'PCO', 'PEO', 'PH', 'PIM', 'PIR', 'PLP', 'PMI', 'POM', 'PPL', 'PRI',
      'RDA', 'REC', 'RPS', 'SAH', 'SJI', 'SLB', 'SLT', 'SNT', 'SPF', 'SWY',
      'T', 'TCI', 'TEK', 'TUG', 'TXI', 'UAM', 'UEP', 'UMG', 'URS',
      'USH', 'UTD', 'UWR', 'VCC', 'VRC', 'W', 'WAE', 'WBN', 'WCS', 'WDG', 'WHX',
      'WIN', 'XON', 'Y', 'ZIF', 'ZNT'], dtype=object)
>>>

```

Spread estimated based on high-frequency data

Based on the **Consolidated Quote (CQ)** dataset supplied by Prof. Hasbrouck, we generate a dataset with the pickle format of Pandas, that can be downloaded from <http://canisius.edu/TORQcq.pickle>. Assume that the following data is located under C:\temp:

```
>>>import pandas as pd
>>>cq=load("c:/temp/TORQcq.pickle")
>>>cq.head()
```

	date	time	bid	ofr	bidsiz	ofrsiz	mode	qseq
symbol								
AC	19901101	9:30:44	12.875	13.125	32	5	10	50
AC	19901101	9:30:47	12.750	13.250	1	1	12	0
AC	19901101	9:30:51	12.750	13.250	1	1	12	0
AC	19901101	9:30:52	12.750	13.250	1	1	12	0
AC	19901101	10:40:13	12.750	13.125	2	2	12	0

```
>>>cq.tail()
```

	date	time	bid	ofr	bidsiz	ofrsiz	mode	qseq
symbol								
ZNT	19910131	13:31:06	12.375	12.875	1	1	12	0
ZNT	19910131	13:31:06	12.375	12.875	1	1	12	0
ZNT	19910131	16:08:44	12.500	12.750	1	1	3	69
ZNT	19910131	16:08:49	12.375	12.875	1	1	12	0
ZNT	19910131	16:16:54	12.375	12.875	1	1	3	0

Again, we could use the `unique()` function to find out all tickers. Assume that we are interested in a stock with an MO ticker as shown in the following code:

```
>>>x=cq[cq.index=='MO']
>>>x.head()
```

	date	time	bid	ofr	bidsiz	ofrsiz	mode	qseq
symbol								
MO	19901101	9:30:33	47.000	47.125	100	4	10	50
MO	19901101	9:30:35	46.750	47.375	1	1	12	0
MO	19901101	9:30:38	46.875	47.750	1	1	12	0
MO	19901101	9:30:40	46.875	47.250	1	1	12	0
MO	19901101	9:30:47	47.000	47.125	100	3	12	51

It is a good idea to check a few observations. From the first line of the following output, we know that spread should be 0.125 (47.125-47.000):

```
>>>x.head().ofr-x.head().bid
symbol
MO      0.125
MO      0.625
MO      0.875
MO      0.375
MO      0.125
dtype: float64
>>>
```

To find the mean spread and the mean relative spread, we have the following code. The complete program is given as follows:

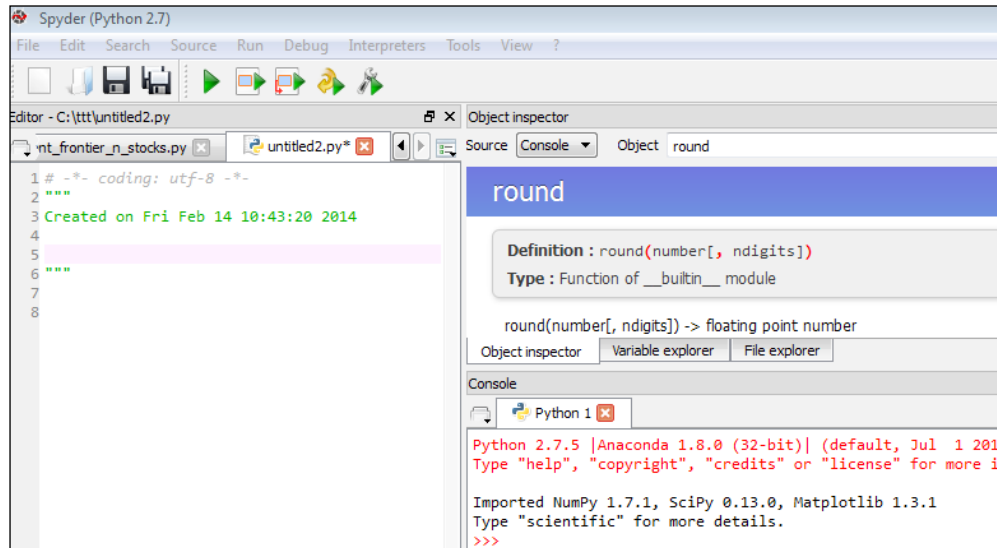
```
import pandas as pd
cq=load('c:/temp/TORQcq.pickle')
x=cq[cq.index=='MO']
spread=mean(x.ofr-x.bid)
rel_spread=mean(2*(x.ofr-x.bid)/(x.ofr+x.bid))
print round(spread,5)
print round(rel_spread,5)
0.39671
0.00788
```

In the preceding example, we didn't process or clean the data. Usually, we have to process data by adding various filters, such as delete quotes with negative spread, bidsiz is zero, or ofrsiz is zero, before we estimate spread and do other estimates.

More on using Spyder

Since Spyder is a wonderful editor, it deserves more space to explain its usage. The related web page for Spyder is <http://pythonhosted.org/spyder/>. According to its importance, we go through the most used features. To see several programs we are just recently working on is a very good feature:

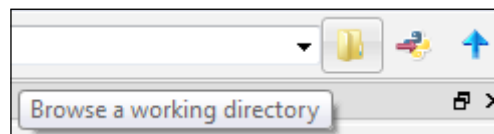
1. Navigate to **File | Open Recent**. We will see a list of files we recently worked on. Just click on the program you want to work on, and it will be loaded as shown in the following screenshot:



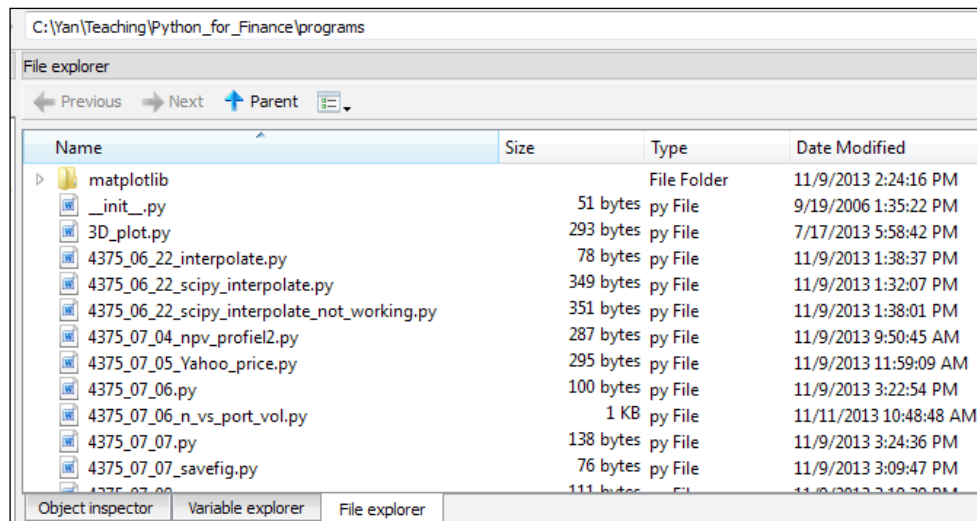
2. Another feature is to run several lines of program instead of the whole program. Select a few lines, click the second green icon just under **Run**. This feature makes our programming and debugging task a little bit easier as shown in the following screenshot:



3. The panel (window) called **File explorer** helps us to see programs under a certain directory. First, we click on the open icon on the top-right of the screen as shown in the following screenshot:



- Then, choose the directory that contains all programs; see the following screenshot:



Occasionally, the **File explorer** panel is not there. You can simply click on **x** on top of the window to make it disappear. To make the **File explorer** window available, click on **View | Windows and tool bars**, and check **File explorer**.

A useful dataset

With limited research funding, many teaching schools would not have a CRSP subscription. For them, we have generated a dataset that contains more than 200 stocks, 15 different country indices, **Consumer Price Index (CPI)**, the US national debt, the prime rate, the risk-free rate, **Small minus Big (SMB)**, **High minus Low (HML)**, Russell indices, and gold prices. The frequency of the dataset is monthly. Since the name of each time series is used as an index, we have only two columns: date and value. The value column contains two types of data: price (level) and return. For stocks, CPI, debt-level, gold price, and Russell indices, their values are the price (level), while for prime rate, risk-free rate, SMB, and HML, the second column under value stands for return. The prime reason to have two types of data is that we want to make such a dataset as reliable as possible since any user could verify any number himself/herself. The dataset could be downloaded from <http://canisius.edu/~yany/yanMonthly.Pickle>. To load this data, we have just one line of the following Python code. Here, we assume that the dataset is saved under `C:\temp`:

```
>>>df=load("c:/temp/yanMonthly.Pickle")
>>>t=unique(np.array(df.index))
```

The corresponding output is displayed as follows:

```
>>> t
array(['000001.SS', 'A', 'AA', 'AAPL', 'BC', 'BCF', 'C', 'CNC', 'COH',
      'CPI', 'DELL', 'GE', 'GOLDPRICE', 'GV', 'GVT', 'HI', 'HML', 'HPS',
      'HY', 'IBM', 'ID', 'IL', 'IN', 'INF', 'ING', 'INY', 'IO', 'ISL',
      'IT', 'J', 'JKD', 'JKE', 'JPC', 'KB', 'KCC', 'KFT', 'KIE', 'KO',
      'KOF', 'LBY', 'LCC', 'LCM', 'LF', 'LG', 'LM', 'M', 'MA', 'MAA',
      'MD', 'MFL', 'MM', 'MPV', 'MY', 'Mkt_Rf', 'NEV', 'NIO', 'NP', 'NU',
      'NYF', 'OI', 'OPK', 'PAF', 'PFO', 'PSJ', 'PZZA', 'Q', 'RH', 'RLV',
      'Rf', 'Russ3000E_D', 'Russ3000E_X', 'S', 'SBR', 'SCD', 'SEF', 'SI',
      'SKK', 'SMB', 'STC', 'T', 'TA', 'TBAC', 'TEN', 'TK', 'TLT', 'TOK',
      'TR', 'TZE', 'UHS', 'UIS', 'URZ', 'US_DEBT', 'US_GDP2009dollar',
      'US_GDP2013dollar', 'V', 'VC', 'VG', 'VGI', 'VO', 'VV', 'WG',
      'WIFI', 'WMT', 'WR', 'XLI', 'XON', 'Y', 'YANG', 'Z', '^AORD',
      '^BSESN', '^CCSI', '^CSE', '^FCHI', '^FTSE', '^GSPC', '^GSPTSE',
      '^HSI', '^IBEX', '^ISEQ', '^JKSE', '^KLSE', '^KS11', '^MXX',
      '^NZ50', '^OMX', '^STI', '^STOXX50E', '^TWII'], dtype=object)
>>> len(t)
129
>>>
```

From the preceding output, we know that we have a total of 129 time series. To select one individual time series, we use the index. For example, if we are interested in the CPI time series, we can retrieve it from the dataset with the following code:

```
>>>x=df[df.index=='CPI']
>>>x.head()
      DATE  VALUE
NAME
CPI  19130101    9.8
CPI  19130201    9.8
CPI  19130301    9.8
CPI  19130401    9.8
CPI  19130501    9.7
>>>x.tail()
      DATE  VALUE
NAME
CPI  20130401  232.531
CPI  20130501  232.945
CPI  20130601  233.504
CPI  20130701  233.596
CPI  20130801  233.877
>>>
```

Summary

In this chapter, many concepts and issues associated with statistics are discussed in detail. Topics include how to download historical prices from Yahoo! Finance; estimate returns, total risk, market risk, correlation among stocks, and correlation among different country's markets; form various types of portfolios; estimate a portfolio variance-covariance matrix; construct an efficient portfolio, and an efficient frontier; and estimate the Roll (1984) spread, Amihud's (2002) illiquidity, and Pastor and Stambaugh's (2003) liquidity.

Although in *Chapter 4, 13 Lines of Python Code to Price a Call Option*, we discuss how to use 13 lines to price a call option based on the Black-Scholes-Merton model even without understanding its underlying theory and logic. In the next chapter, we will explain the option theory and its related applications in more detail.

Exercise

1. What is the usage of the module called `Pandas`?
2. What is the usage of the module called `statsmodels`?
3. How can you install `Pandas` and `statsmodels`?
4. Which module contains the function called `rolling_kurt`? How can you use the function?
5. Based on daily data downloaded from Yahoo! Finance, find whether IBM's daily returns follows a normal distribution.
6. Based on daily returns in 2012, are the mean returns for IBM and DELL the same? [Hint: you can use Yahoo! Finance as your source of data].
7. How can you replicate the Jagadeech and Tidman (1993) momentum strategy using Python and CRSP data? [Assume that your school has CRSP subscription].
8. How many events happened in 2012 for IBM based on its daily returns?
9. For the following stock tickers, IBM, DELL, WMT, ^GSPC, C, A, AA, MOFT, estimate their variance-covariance and correlation matrices based on the last five-year monthly returns data, for example, from 2008-2012. Which two stocks are strongly correlated?
10. Write a Python program to estimate rolling beta on a yearly basis. Use it to show the annual beta for IBM from 1962 to 2013.

11. Assume that we just downloaded the prime rate from the Federal Banks' data library from <http://www.federalreserve.gov/releases/h15/data.htm>. We downloaded the time series for Financial 1-month business day. A few lines of the file are given as follows. Write a Python program to retrieve it and use the first column as the index:

```
Series Description  30-Day AA Financial Commercial Paper Interest Rate
Unit: Percent
Multiplier: 1
Currency: NA
Unique Identifier:  H15/H15/RIFSPFFAAD30_N.B
Time Period  RIFSPFFAAD30_N.B
1/2/1997  5.35
1/3/1997  5.34
```

12. Which political party could manage the stock market better? According to the web page at <http://www.enchantedlearning.com/history/us/pres/list.shtml>, we can find to which party a president belongs. Thus, we can generate the following table. The PARTY and RANGE variables are from the web page. YEAR2 is the second number of RANGE minus one, except the last row:

PARTY	RANGE	YEAR1	YEAR2
Republican	1923-1929	1923	1928
Republican	1929-1933	1929	1932
Democrat	1933-1945	1933	1944
Democrat	1945-1953	1945	1952
Republican	1953-1961	1953	1960
Democrat	1961-1963	1961	1962
Democrat	1963-1969	1963	1968
Republican	1969-1974	1969	1973
Republican	1974-1977	1974	1976
Democrat	1977-1981	1977	1980
Republican	1981-1989	1981	1988
Republican	1989-1993	1989	1992
Democrat	1993-2001	1993	2000
Republican	2001-2009	2001	2008
Democrat	2009-2012	2009	2012

1. Download excess market return and risk-free from Prof. French data library at http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html.
2. Estimate market returns (excess market return plus risk-free rate).
3. Classify those returns into two groups: under Republican and Democratic.
4. Test the null hypothesis: two group means are equal:

$$\bar{R}_{Democratic} = \bar{R}_{Republican}$$

Note: 1: How do we download and estimate market returns?

1. Go to the web page http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html.
2. Click on **Fama-French Factor**, and download their monthly factors named **F-F_Research_Data_Factors.zip**.
3. Unzip the zip file and estimate market monthly returns. For example, for July 1926, market return = 2.65/100+0.22/100:

This file was created by CMPT_ME_BEME_RETS using the 201212 CRSP database.

The 1-month T-Bill return is from Ibbotson and Associates, Inc.

	Mkt-RF	SMB	HML	RF
192607	2.65	-2.16	-2.92	0.22
192608	2.58	-1.49	4.88	0.25
192609	0.37	-1.38	-0.01	0.23
192610	-3.46	0.04	0.71	0.32
192611	2.43	-0.24	-0.31	0.31
192612	2.75	-0.01	-0.10	0.28
192701	-0.16	-0.30	4.79	0.25
192702	4.22	-0.24	3.35	0.26
192703	0.38	-1.87	-2.58	0.30
192704	0.41	0.29	0.95	0.25
192705	5.36	1.53	5.07	0.30

13. From Prof. French's data library at http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html, download the monthly and daily Fama-French factors, where SMB is for Small minus Big, and HML is for High minus Low. Assume that you are holding an SMB portfolio. Answer the following three questions:

1) What is the total return from January 1, 1989 to December 31, 2012 using daily data?

2) What is the total return from January 1, 1989, to December 31, 2012, using monthly data?

3) Are they the same? If they are different, why?

14. The following table presents the relationship between rating, default risk (spread), and time. Write a Python program to interpolate the missing spreads, such as S from year 11 to 29:

Rating	1 yr	2 yr	3 yr	5 yr	7 yr	10 yr	30 yr
Aaa/AAA	14	16	27	40	56	68	90
Aa1/AA+	22	30	31	48	64	77	99
Aa2/AA	24	37	39	54	67	80	103
Aa3/AA-	25	39	40	58	71	81	109
A1/A+	43	48	52	65	79	93	117
A2/A	46	51	54	67	81	95	121
A3/A-	50	54	57	72	84	98	124
Baa1/BBB+	62	72	80	92	121	141	170
Baa2/BBB	65	80	88	97	128	151	177
Baa3/BBB-	72	85	90	102	134	159	183
Ba1/BB+	185	195	205	215	235	255	275
Ba2/BB	195	205	215	225	245	265	285
Ba3/BB-	205	215	225	235	255	275	295
B1/B+	265	275	285	315	355	395	445
B2/B	275	285	295	325	365	405	455
B3/B-	285	295	305	335	375	415	465
Caa/CCC+	450	460	470	495	505	515	545

The table is located at <http://www.bondsonline.com>. The values in the table are expressed in basis points, that are equivalent to 100th of one percent. For example, 40 is equivalent to 40*0.0001.

15. First, download three daily and monthly factors, Market, SMB, and HML from Prof. French's data library at http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html. Write a Python program to process them. Choose a time period, such as from January 1, 2000, to December 31, 2013, to estimate the total returns for the SMB portfolio by using both daily and monthly factors. Are they the same? What is the total difference? What is the average annual difference? Comment on your findings.

9

The Black-Scholes-Merton Option Model

In modern finance, the option theory and its applications play an important role. Many trading strategies, corporate incentive plans, and hedging strategies include various types of options. In *Chapter 6, Introduction to NumPy and SciPy*, we showed that you can write a five-line Python program to price a call option based on the Black-Scholes-Merton option model even without understanding its underlying theory and logic. In this chapter, we will explain the option theory and its related applications in more detail.

In particular, we will cover the following topics:

- Payoff and profit/loss functions and their graphical representations of call and put
- European versus American options
- Normal distribution, standard normal distribution, and cumulative normal distribution
- The Black-Scholes-Merton option model with/without dividend
- Various trading strategies and their visual presentations, such as covered call, straddle, butterfly, and calendar spread
- Delta, gamma, and other Greeks
- The put-call parity and its graphical representation
- Graphical representation for a one-step and a two-step binomial tree model
- Using the binomial tree method to price both European and American options
- Hedging strategies

Payoff and profit/loss functions for the call and put options

An option gives its buyer the right to buy (call option) or sell (put option) something in the future to the option seller at a predetermined price (exercise price). For example, if we buy a European call option to acquire a stock for X dollars, such as \$30, at the end of three months, our payoff on maturity day will be the one calculated using the following formula:

$$\text{payoff}(\text{call}) = \text{Max}(S_T - X, 0) \quad (1)$$

Here, S_T is the stock price at the maturity date (T), and the exercise price is X (X is equal to 30 in this case). Assume that three months later the stock price will be \$25. We would not exercise our call option to pay \$30 in exchange for the stock, since we could buy the same stock with \$25 in the open market. On the other hand, if the stock price is \$40, we will exercise our right to reap a payoff of \$10, that is, buy the stock at \$30 and sell it at \$40. The following program presents the payoff function for a call:

```
>>>def payoff_call(sT,x):  
    return (sT-x+abs(sT-x))/2
```

Applying the payoff function is straightforward, as shown in the following code:

```
>>>payoff_call(25,30)  
0  
>>>payoff_call(40,30)  
10
```

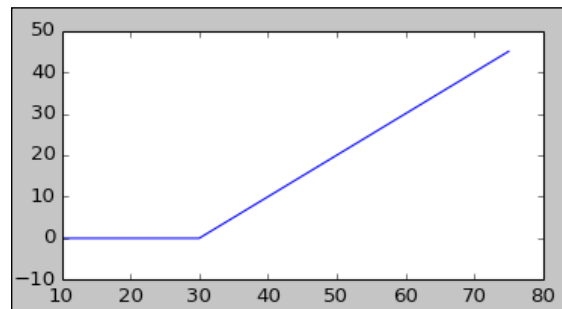
The first input variable, the stock price at the maturity T , could be an array as well, as shown in the following code:

```
>>>import numpy as np  
>>>x=20  
>>>sT=np.arange(10,50,10)  
>>>sT  
array([10, 20, 30, 40])  
>>>payoff_call(s,x)  
array([ 0.,  0., 10., 20.]  
>>>
```

To create a graphical representation, we have the following commands:

```
>>>import numpy as np
>>>s = np.arange(10,80,5)
>>>x=30
>>>payoff=(abs(s-x)+s-x)/2
>>>ylim(-10,50)
>>>plot(s,payoff)
```

The graph is shown in the following screenshot:



The payoff for a call option seller is the opposite of its buyer. It is important to remember that this is a zero-sum game: you win, I lose. For example, an investor sold three call options with an exercise price of \$10. When the stock price is \$15 on the maturity, the option buyer's payoff is \$15, while the total loss to the option writer is \$15 as well. If the call premium (option price) is c , the profit/loss function for a call option buyer is the difference between his/her payoff and his/her initial investment (c). Obviously, the timing of cash flows of paying an option premium upfront and its payoff at maturity date is different. Here, we ignore the time value of money since maturities are usually quite short.

For a call option buyer, the profit is calculated using the following formula:

$$\text{Buyer Profit / loss}(\text{call}) = \text{Max}(S_T - X, 0) - c \quad (2)$$

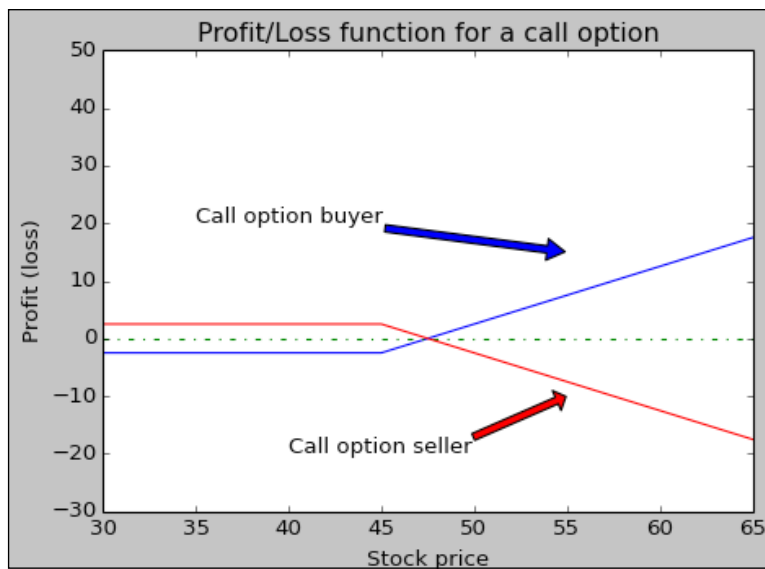
For a call option seller, the profit is calculated by using the following formula:

$$\text{Seller Profit / loss}(\text{call}) = c - \text{Max}(S_T - X, 0) \quad (3)$$

A graph showing the profit/loss functions for the call option buyer and seller is generated using the following code:

```
s = arange(30,70,5)
x=45;call=2.5
profit=(abs(s-x)+s-x)/2 -call
y2=zeros(len(s))
ylim(-30,50)
plot(s,profit)
plot(s,y2,'-.')
plot(s,-profit)
title("Profit/Loss function")
xlabel('Stock price')
ylabel('Profit (loss)')
plt.annotate('Call option buyer', xy=(55,15), xytext=(35,20),
            arrowprops=dict(facecolor='blue',shrink=0.01),)
plt.annotate('Call option seller', xy=(55,-10), xytext=(40,-20),
            arrowprops=dict(facecolor='red',shrink=0.01),)
show()
```

The graphical representation is shown in the following screenshot:



A put option gives its buyer the right to sell a security (commodity) to the put option buyer in the future at a predetermined price, X . The following is its payoff function:

$$\text{Payoff}(\text{put}) = \text{Max}(X - S_T, 0) \quad (4)$$

Here, S_T is the stock price at maturity and X is the exercise price (strike price). For a put option buyer, the profit/loss function is as follows:

$$\text{Buyer Profit / loss}(\text{call}) = \text{Max}(X - S_T, 0) - p \quad (5)$$

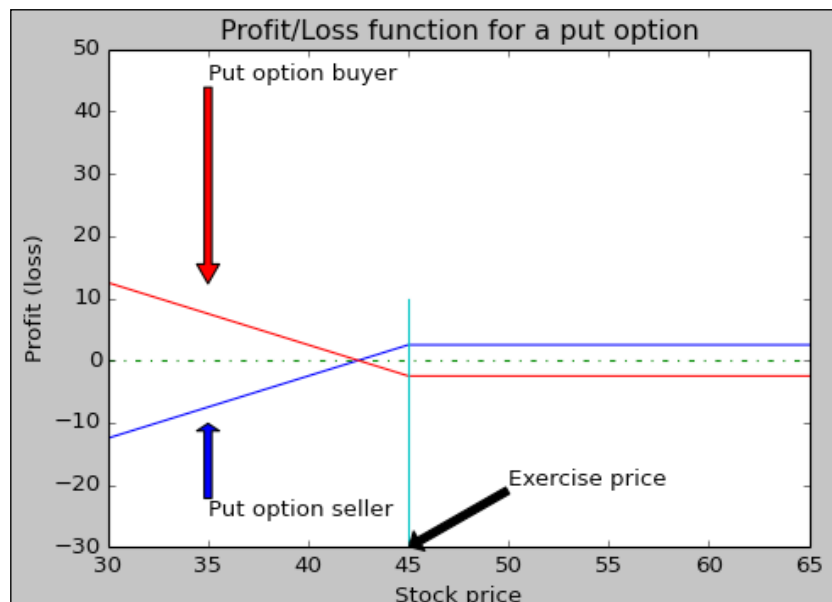
The profit/loss function for a put option seller is just the opposite, as follows:

$$\text{Seller Profit / loss}(\text{call}) = p - \text{Max}(X - S_T, 0) \quad (6)$$

The related program and graph for the profit and loss functions for a put option buyer and a seller are as follows:

```
s = arange(30,70,5)
x=45;p=2
y=c-(abs(x-s)+x-s)/2
y2=zeros(len(s))
x3=[x, x]
y3=[-30,10]
ylim(-30,50)
plot(s,y)
plot(s,y2,'-.')
plot(s,-y)
plot(x3,y3)
title("Profit/Loss function for a put option")
xlabel('Stock price')
ylabel('Profit (loss)')
plt.annotate('Put option buyer', xy=(35,12), xytext=(35,45),
             arrowprops=dict(facecolor='red',shrink=0.01),)
plt.annotate('Put option seller', xy=(35,-10), xytext=(35,-25),
             arrowprops=dict(facecolor='blue',shrink=0.01),)
plt.annotate('Exercise price', xy=(45,-30), xytext=(50,-20),
             arrowprops=dict(facecolor='black',shrink=0.01),)
show()
```

The graph is shown in the following image:



European versus American options

A European option can be exercised only on the maturity date, while an American option can be exercised any time before or on its maturity date. Since an American option could be held until it matures, its price (option premium) should be higher than or equal to its European counterparty.

$$\begin{cases} C_{American} \geq C_{European} \\ P_{American} \geq P_{European} \end{cases} \quad (7)$$

An important difference is that for a European option, we have a closed-form solution, that is, the Black-Scholes-Merton option model. However, we don't have a closed-form solution for an American option. Fortunately, we have several ways to price an American option. Later in the chapter, we will show how to use the binomial tree method, also called the CRR method, to price an American option.

Cash flows, types of options, a right, and an obligation

We know that for each business contract, we have two sides, a buyer and a seller. This is true for an option contract as well. A call buyer will pay upfront (cash output) to acquire a right. Since this is a zero-sum game, a call option seller would enjoy an upfront cash inflow and assumes an obligation. The following table presents those positions (buyer or seller), directions of the initial cash flows (inflow or outflow), the option buyer's rights (buy or sell), and the option seller's obligations (that is, to satisfy the option seller's demand):

	Buyer (long position)	Seller (short position)	European options	American options
Call	A right to buy a security (commodity) at a prefixed price	An obligation to sell a security (commodity) at a prefixed price	Are exercised on the maturity date only	Could be exercised any time before or on the maturity date
Put	A right to sell a security with a prefixed price	An obligation to buy		
Cash Flow	Upfront cash outflow	Upfront cash inflow		

The preceding table displays long/short, call/put, European/American options and directions of initial cash flows.

Normal distribution, standard normal distribution, and cumulative standard normal distribution

In finance, normal distribution plays a central role. This is especially true for option theory. The major reason is that it is commonly assumed that the stock prices follow a log normal distribution while the stock returns follow a normal distribution. The density of a normal distribution is defined as follows:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (8)$$

Here, μ is the mean and σ is the standard deviation.

By setting μ as 0 and σ as 1, the preceding general normal distribution density function collapses to the following standard normal distribution:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \quad (9)$$

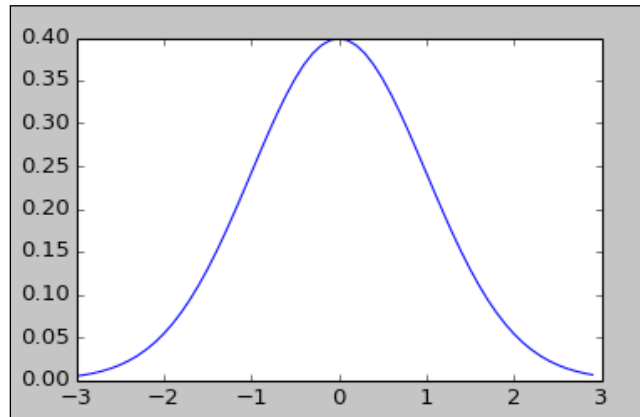
The following code generates a graph for the standard normal distribution. The SciPy's `stats.norm.pdf()` function is used for the standard normal distribution. The default setting is with a zero mean and unit standard deviation, that is, the standard normal density function:

```
>>>from scipy import exp,sqrt,stats
>>>stats.norm.pdf(0)
0.3989422804014327
>>>1/sqrt(2*pi)                                     # verify manually
0.3989422804014327
>>>stats.norm.pdf(0,0.1,0.05)
1.0798193302637611
>>>1/sqrt(2*pi*0.05**2)*exp(-(0.1)**2/0.05**2/2) # verify manually
1.0798193302637611
>>>
```

To draw a standard normal distribution, we have the following program:

```
>>>from scipy import exp,sqrt,stats
>>>x = arange(-3,3,0.1)
>>>y=stats.norm.pdf(x)
>>>plot(x,y)
```

The graph is shown in the following screenshot:



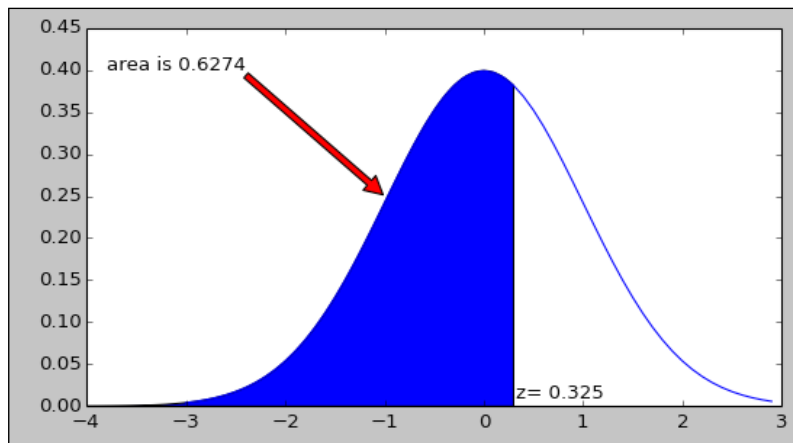
The cumulative standard normal distribution is the area under the standard normal density function. In the following program, we will randomly choose a value of 0.325 (the z value). The shaded area on the left-hand side of the z value and under the standard normal distribution will be the value for a cumulative normal distribution:

```
import numpy as np
from scipy import exp,sqrt,stats
from matplotlib import pyplot as plt
z=0.325                      # user can change this number
def f(t):
    return stats.norm.pdf(t)
ylim(0,0.45)
x = np.arange(-3,3,0.1)
y1=f(x)
plt.plot(x,y1)
x2= np.arange(-4,z,1/40.)
sum=0
```



```
delta=0.05
s=np.arange(-10,z,delta)
for i in s:
    sum+=f(i)*delta
plt.annotate('area is '+str(round(sum,4)),xy=(-1,0.25),xytext=(-3.8,0.4),
            arrowprops=dict(facecolor='red',shrink=0.01))
plt.annotate('z= '+str(z),xy=(z,0.01))
plt.fill_between(x2,f(x2))
```

The graphical representation for the preceding code is as follows:

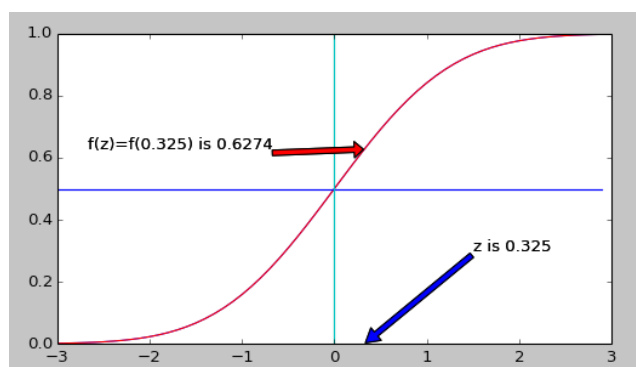


The `stats.norm.cdf()` function is the cumulative standard normal distribution and is as follows:

```
from scipy import exp,sqrt,stats
from matplotlib import pyplot as plt
z=0.325
def f(x):
    return stats.norm.cdf(x)
x = arange(-3,3,0.1)
y1=f(x)
y2=ones(len(x))*0.5
x3=[0,0]
y3=[0,1]
plt.plot(x,y1)
plt.plot(x, y2, 'b-')
```

```
plt.plot(x3,y3)
plt.annotate('f(z)=f('+str(z)+') is '+str(np.round(f(z),4)),xy=(z,f(z)),
            xytext=(z-3,f(z)), arrowprops=dict(facecolor='red',shrink=0.01))
plt.annotate('z is '+str(z),xy=(z,0),xytext=(1.5,0.3),
            arrowprops=dict(facecolor='blue',shrink=0.01))
```

The following is the corresponding graph of the preceding code. Obviously, since the normal distribution is symmetric, we could expect the cumulative standard normal distribution to be 0.5 at zero, as shown in the following screenshot:



The Black-Scholes-Merton option model on non-dividend paying stocks

The Black-Scholes-Merton option model is a closed-form solution to price a European option on a stock that does not pay any dividends before its maturity date. If we use S_0 for the price today, X for the exercise price, r for the continuously compounded risk-free rate, T for the maturity in years, and σ for the volatility of the stock, the closed-form formulae for a European call (c) and put (p) will be as follows:

$$\left\{ \begin{array}{l} d_1 = \frac{\ln\left(\frac{S_0}{X}\right) + \left(\gamma + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}} \\ d_2 = \frac{\ln\left(\frac{S_0}{X}\right) + \left(\gamma - \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T} \\ c = S_0 N(d_1) - Xe^{-\gamma T} N(d_2) \\ p = Xe^{-\gamma T} N(-d_2) - S_0 N(-d_1) \end{array} \right. \quad (10)$$

Here, $N()$ is the cumulative standard normal distribution. The following Python code snippet represents the preceding formulae to evaluate a European call:

```
from scipy import log,exp,sqrt,stats
def bs_call(S,X,T,r,sigma):
    d1=(log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    d2 = d1-sigma*sqrt(T)
    return S*stats.norm.cdf(d1)-X*exp(-r*T)*stats.norm.cdf(d2)
```

In the preceding program, the `stats.norm.cdf()` function is the cumulative normal distribution, that is, $N()$ in the Black-Scholes-Merton option model. The current stock price is \$40, the strike price is \$42, the time to maturity is six months, the risk-free rate is 1.5 percent compounded continuously, and the volatility of the underlying stock is 20 percent (compounded continuously). Based on the preceding code, the European call is worth \$1.56, as shown in the following code:

```
>>>c=bs_call(40,42,0.5,0.015,0.2)
>>>round(c,2)
1.56
```

The p4f module for options

In *Chapter 3, Using Python as a Financial Calculator*, we recommended the combining of many small Python programs as one program. In this chapter, we adopted the same strategy to combine all the programs in a big file `p4f.py`. For instance, the preceding Python program, that is, the `bs_call()` function is included. Such a collection of programs offers several benefits. First, when we use the `bs_call()` function, we don't have to type those five lines. To save space, we will only show a few functions included in `p4f.py`. For brevity, we will remove all the comments included for each function. Those comments are designed to help future users when issuing the `help()` function, such as `help(bs_call())`.

```
def bs_call(S,X,T,rf,sigma):
    from scipy import log,exp,sqrt,stats
    d1=(log(S/X)+(rf+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    d2 = d1-sigma*sqrt(T)
    return S*stats.norm.cdf(d1)-X*exp(-rf*T)*stats.norm.cdf(d2)
```

The following program uses a binomial model to price a call option:

```
def binomial_grid(n):
    import networkx as nx
    import matplotlib.pyplot as plt
```

```

G=nx.Graph()
for i in range(0,n+1):
    for j in range(1,i+2):
        if i<n:
            G.add_edge((i,j),(i+1,j))
            G.add_edge((i,j),(i+1,j+1))
posG={}    #dictionary with nodes position
for node in G.nodes():
    posG[node]=(node[0],n+2+node[0]-2*node[1])
nx.draw(G,pos=posG)

def delta_call(S,X,T,rf,sigma):
    from scipy import log,exp,sqrt,stats
    d1=(log(S/X)+(rf+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    return(stats.norm.cdf(d1))

def delta_put(S,X,T,rf,sigma):
    from scipy import log,exp,sqrt,stats
    d1=(log(S/X)+(rf+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    return(stats.norm.cdf(d1)-1)

```

To implement the Black-Scholes-Merton call option model, we simply use the following code:

```

>>>import p4f
>>>c=p4f.bs_call(40,42,0.5,0.015,0.2)
>>>round(c,2)
1.56

```

The second advantage is to save space and make our programming simpler. Later in the chapter, this point will become more clearer when we use the `binomial_grid()` function. From now on, when a function is discussed for the first time, we will offer complete code. However, when the program is used again and the program is quite complex, we would call it indirectly via `p4f`. To find our working directory, use the following code:

```

>>>import os
>>>print os.getcwd()

```

European options with known dividends

Assume that we have a known dividend d distributed at time T_1 , $T_1 < T$, where T is our maturity date. We can modify the original Black-Scholes-Merton option model by replacing S_0 with S , where:

$$S = S_0 - e^{-rT_1} d \quad (11)$$

$$d_1 = \frac{\ln\left(\frac{S}{X}\right) + \left(\gamma + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}} \quad (12)$$

$$d_2 = \frac{\ln\left(\frac{S}{X}\right) + \left(\gamma - \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T} \quad (13)$$

$$c = S * N(d_1) - X * e^{-rT} N(d_2) \quad (14)$$

$$p = X * e^{-rT} N(-d_2) - S * N(-d_1) \quad (15)$$

In the previously discussed example, if we have a known dividend of \$1.5 delivered in one month, what is the price of the call?. The price is calculated as follows:

```
>>>import p4f
>>>s0=40
>>>d=1.5
>>>r=0.015
>>>T=6/12
>>>s=s0-exp(-r*T*d)
>>>x=42
>>>sigma=0.2
>>>round(p4f.bs_call(s,x,T,r,sigma),2)
1.18
```

The first line of the program imports the `p4f` module, which contains the call option model. The result shows that the price of the call is \$1.18, which is lower than the previous value (\$1.56). It is understandable since the price of the underlying stock would drop roughly by \$1.5 in one month. Because of this, the chance that we could exercise our call option will be less, that is, less likely to go beyond \$42. The preceding argument is true for multiple known dividends distributed before T , that is, $S_t = S_0 - \sum e^{-rt_i} d_i$.

Various trading strategies

In the following table, we will summarize several commonly used trading strategies involving various types of options:

Names	Description	Direction of initial cash flow	Expectation of future price movement
Bull spread with calls	Buy a call (x_1) sell a call (x_2) [$x_1 < x_2$]	Outflow	Rise
Bull spread with puts	Buy a put (x_1), sell a put (x_2) [$x_1 < x_2$]	Inflow	Rise
Bear spread with puts	Buy a put (x_2), sell a put (x_1) [$x_1 < x_2$]	Outflow	Fall
Bear spread with calls	Buy a call (x_2), sell a call (x_1) [$x_1 < x_2$]	Inflow	Fall
Straddle	Buy a call and sell a put with the same x value	Outflow	Rise or fall
Strip	Buy two puts and a call (with the same x value)	Outflow	prob (fall) > prob (rise)
Strap	Buy two calls and one put (with the same x value)	Outflow	prob (rise) > prob (fall)
Strangle	Buy a call (x_2) and buy a put (x_1) [$x_1 < x_2$]	Outflow	rise or fall
Butterfly with calls	Buy two calls (x_1, x_3) and sell two calls (x_2) $x_2 = (x_1 + x_3) / 2$	Outflow	stay around x_2
Butterfly with puts	Buy two puts (x_1, x_3) and sell two puts (x_2) $x_2 = (x_1 + x_3) / 2$		stay around x_2
Calendar spread	Sell a call (T_1) and buy a call (T_2) with the same strike price and $T_1 < T_2$	Outflow	

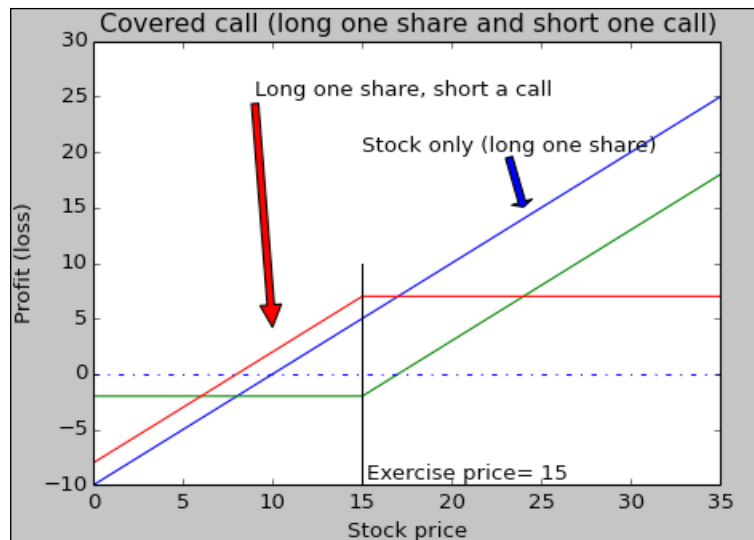
Covered call – long a stock and short a call

Assume that we purchase 100 shares of stock A, with a price of \$10 each. Thus, the total cost is \$1,000. If at the same time, we write a call contract, one contract is worth 100 shares, at a price of \$20. Thus, our total cost will be reduced by \$20. Assume further that the exercise price is \$12. The graphical representation of our profit and loss function is as follows:

```
import matplotlib as plt
import numpy as np
sT = arange(0,40,5)
k=15;s0=10;c=2
y0=zeros(len(sT))
y1=sT-s0                                # stock only
y2=(abs(sT-k)+sT-k)/2-c                 # long a call
y3=y1-y2                                # covered-call
ylim(-10,30)
plot(sT,y1)
plot(sT,y2)
plot(sT,y3,'red')
plot(sT,y0,'b-.')
plot([k,k],[-10,10],'black')
title('Covered call (long one share and short one call)')
xlabel('Stock price')
ylabel('Profit (loss)')
plt.annotate('Stock only (long one share)', xy=(24,15),xytext=(15,20),
            arrowprops=dict(facecolor='blue',shrink=0.01),)
plt.annotate('Long one share, short a call', xy=(10,4), xytext=(9,25),
            arrowprops=dict(facecolor='red',shrink=0.01),)
plt.annotate('Exercise price= '+str(k), xy=(k+0.2,-10+0.5))

show()
```

The related graph showing the positions of a **Stock only** call and a covered call is given in the following screenshot. Obviously, when the stock price is below \$17 (15 + 2), the covered call is better than a long share.



Straddle – buy a call and a put with the same exercise prices

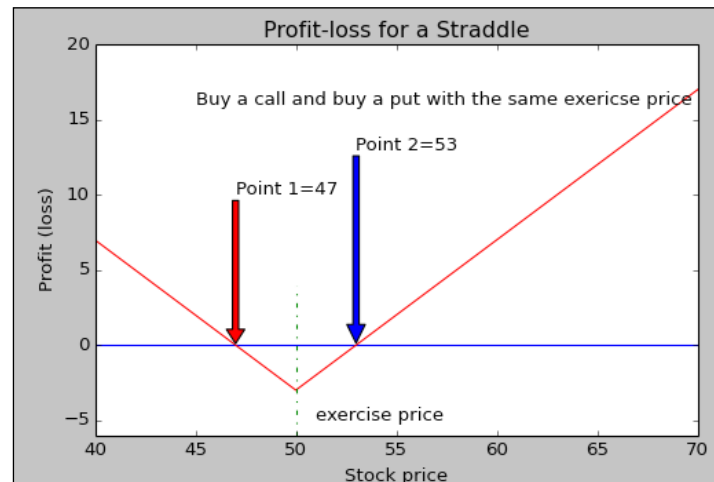
Let's look at a very simple scenario. A firm faces an uncertain event next month. The issue is that we are not sure about its direction, that is, whether it is a good event or a bad one. To take advantage of such an opportunity, we could buy a call and a put with the same exercise prices. This means that we will benefit either ways: the stock moves up or down. Assume further that the exercise price is \$30. The payoff of such a strategy is given in the following code:

```
import matplotlib.pyplot as plt
sT = arange(30,80,5)
x=50; c=2; p=1
straddle=(abs(sT-x)+sT-x)/2-c + (abs(x-sT)+x-sT)/2-p
y0=zeros(len(sT))
ylim(-6,20)
xlim(40,70)
plot(sT,y0)
plot(sT,straddle,'r')
plot([x,x],[-6,4],'g-.')
title("Profit-loss for a Straddle")
xlabel('Stock price')
ylabel('Profit (loss)')
plt.annotate('Point 1='+str(x-c-p), xy=(x-p-c,0), xytext=(x-p-c,10),
```



```
        arrowprops=dict(facecolor='red',shrink=0.01),)
plt.annotate('Point 2='+str(x+c+p), xy=(x+p+c,0), xytext=(x+p+c,13),
            arrowprops=dict(facecolor='blue',shrink=0.01),)
plt.annotate('exercise price', xy=(x+1,-5))
plt.annotate('Buy a call and buy a put with the same exercise price',xy=(45,16))
```

The preceding code gives us the following graph:



The preceding graph shows that whichever way the stock goes, we would get the profit. When could we lose? Obviously, when the stock does not change much, that is, our expectation fails to materialize.

A calendar spread

A calendar spread involves two call options with the same exercise price but different maturities: T_1 and T_2 (where $T_1 < T_2$). We sell a call with a shorter maturity (T_1) and buy a call with a longer maturity (T_2). Since the call option price is positively correlated with the maturity, we have initial cash outflow. Our expectation is that when the first option matures at T_1 , the price of the underlying stock is close to our exercise price. The code and graph for this are as follows:

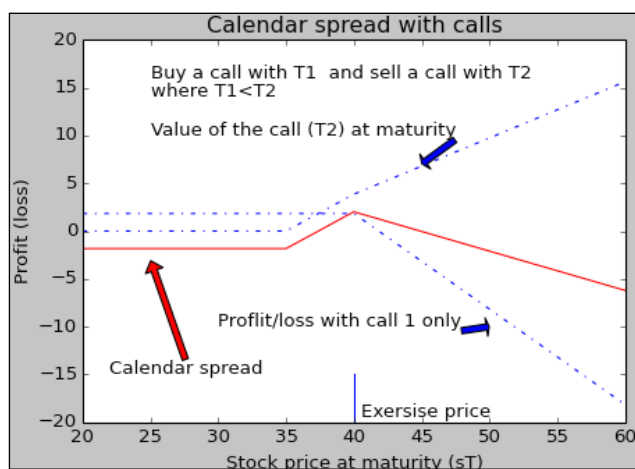
```
import p4f
sT = arange(20,70,5)
s=40;x=40;T1=0.5;T2=1;sigma=0.3;r=0.05
payoff=(abs(sT-x)+sT-x)/2
call_01=p4f.bs_call(s,x,T1,r,sigma) # short
call_02=p4f.bs_call(s,x,T2,r,sigma) # long
```

```

profit_01=payoff-call_01
call_03=p4f.bs_call(sT,x,(T2-T1),r,sigma)
calendar_spread=call_03-payoff+call_01 -call_02
y0=zeros(len(sT))
ylim(-20,20)
xlim(20,60)
plot(sT,call_03,'b-.')
plot(sT,call_02-call_01-payoff,'b-.')
plot(sT,calendar_spread,'r')
plot([x,x],[-20,-15])
title("Calendar spread with calls")
xlabel('Stock price at maturity (sT)')
ylabel('Profit (loss)')
plt.annotate('Buy a call with T1 and sell a call with T2', xy=(25,16))
plt.annotate('where T1<T2', xy=(25,14))
plt.annotate('Calendar spread', xy=(25,-3), xytext=(22,-15),
            arrowprops=dict(facecolor='red',shrink=0.01),)
plt.annotate('Value of the call (T2) at maturity', xy=(45,7),
            xytext=(25,10),
            arrowprops=dict(facecolor='blue',shrink=0.01),)
plt.annotate('Profit/loss with call 1 only', xy=(50,-10),
            xytext=(30,-10),
            arrowprops=dict(facecolor='blue',shrink=0.01),)
plt.annotate('Exercise price', xy=(x+0.5,-20+0.5))
show()

```

The preceding code gives the following graph:



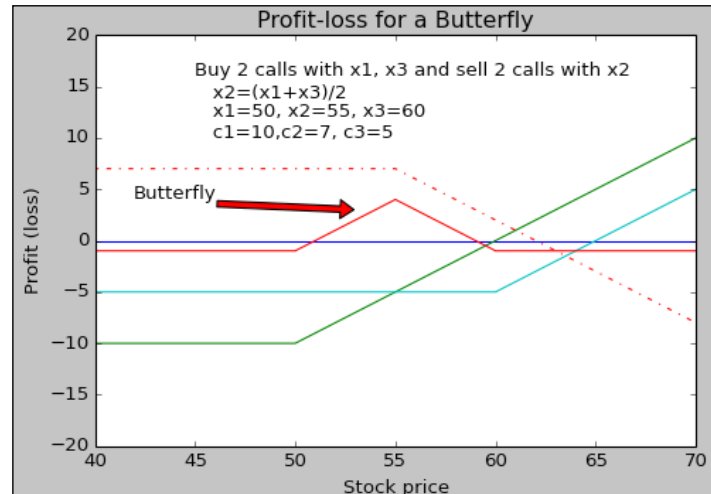
Butterfly with calls

While buying two calls with the exercise prices x_1 and x_3 and selling two calls with the exercise price x_2 , where $x_2 = (x_1 + x_3)/2$, with the same maturity for the same stock, we call it butterfly. Its profit/loss function is as follows:

```
sT = arange(30,80,5)
x1=50;    c1=10
x2=55;    c2=7
x3=60;    c3=5

y1=(abs(sT-x1)+sT-x1)/2-c1
y2=(abs(sT-x2)+sT-x2)/2-c2
y3=(abs(sT-x3)+sT-x3)/2-c3
butter_fly=y1+y3-2*y2
y0=zeros(len(sT))
ylim(-20,20)
xlim(40,70)
plot(sT,y0)
plot(sT,y1)
plot(sT,-y2,'-.')
plot(sT,y3)
plot(sT,butter_fly,'r')
title("Profit-loss for a Butterfly")
xlabel('Stock price')
ylabel('Profit (loss)')
plt.annotate('Butterfly', xy=(53,3), xytext=(42,4),
             arrowprops=dict(facecolor='red',shrink=0.01),)
plt.annotate('Buy 2 calls with x1, x3 and sell 2 calls with x2',
             xy=(45,16))
plt.annotate('    x2=(x1+x3)/2', xy=(45,14))
plt.annotate('    x1=50, x2=55, x3=60',xy=(45,12))
plt.annotate('    c1=10,c2=7, c3=5', xy=(45,10))
show()
```

The related graph is shown in the following image:



Relationship between input values and option values

When the volatility of an underlying stock increases, both its call and put values increase. The logic is that when a stock becomes more volatile, we have a better chance to observe extreme values, that is, we have a better chance to exercise our option. The following Python program shows this relationship:

```
import numpy as np
import p4f as pf
s0=30;T0=0.5;sigma0=0.2;r0=0.05;x0=30
sigma=np.arange(0.05,0.8,0.05)
T=np.arange(0.5,2.0,0.5)
call_0=pf.bs_call(s0,x0,T0,r0,sigma0)
call_sigma=pf.bs_call(s0,x0,T0,r0,sigma)
call_T=pf.bs_call(s0,x0,T,r0,sigma0)
plot(sigma,call_sigma,'b')
plot(T,call_T)
```

Greek letters for options

In the option theory, several Greek letters, usually called Greeks, are used to represent the sensitivity of the price of derivatives such as options to bring a change in the underlying security. Collectively, those Greek letters are also called the risk sensitivities, risk measures, or hedge parameters.

Delta (Δ) is defined as the derivative of the option to its underlying security price. The delta of a call is defined as follows:

$$\Delta = \frac{\partial C}{\partial S} \quad (16)$$

We could design a hedge based on the delta value. The delta of a European call on a non-dividend-paying stock is defined as follows:

$$\Delta_{call} = N(d_1) \quad (17)$$

For example, if we write one call, we could buy delta number of shares of stocks so that a small change in the stock price is offset by the change in the short call. The definition of the `delta_call()` function is quite simple. Since it is included in the `p4f.py` file, we can call it easily, as shown in the following code:

```
>>>from p4f import *
>>>round(delta_call(40,40,1,0.1,0.2),4)
0.7257
```

The delta for a European put on a non-dividend-paying stock is defined as follows:

$$\Delta_{put} = N(d_1) - 1 \quad (18)$$

The definition of the `delta_put` function is as follows:

```
>>>from p4f import *
>>>round(delta_put(40,40,1,0.1,0.2),4)
-0.2743
```

Gamma is the rate of change of delta with respect to price. It can be defined as follows:

$$\Gamma = \frac{\partial \Delta}{\partial S} \quad (19)$$

To implement an effective delta hedge, we have to update our stockholding constantly since delta depends on the price of the underlying security. Thus, if gamma is small, we don't have to change our hedge too frequently. For a European call (or put), its gamma value is given as follows:

$$\Gamma = \frac{N'(d_1)}{S_0 \sigma \sqrt{T}} \quad (20)$$

Here, $N'(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$.

The put-call parity and its graphical representation

Let's look at a call with an exercise price of \$20, a maturity of three months, and a risk-free rate of 5 percent. The present value of this future \$20 price is calculated in the following code:

```
>>>x=20*exp(-0.05*3/12)
>>>round(x,2)
19.75
>>>
```

In three months, what will be the wealth of our portfolio, which consists of a call on the same stock and \$19.75 cash today? If the stock price is below \$20, we don't exercise the call and keep the cash. If the stock price is above \$20, we use our cash of \$20 to exercise our call option to own the stock. Thus, our portfolio value will be the maximum of those two values, that is, the stock price in three months or \$20, $\max(s, 20)$.

On the other hand, how about a portfolio with a stock and a put option with an exercise price of \$20? If the stock price falls below \$20, we exercise the put option and get \$20. If the stock price is above \$20, we simply keep the stock. Thus, our portfolio value will be the maximum of those two values, that is, the stock price in three months or \$20, $\max(s, 20)$.

Thus, for both the portfolios, we have the same terminal wealth of $\max(s, 20)$. Based on the no-arbitrage principle, the present values of those two portfolios should be equal. We call this put-call parity, and we define it as follows:

$$C + Xe^{-r_f T} = P + S_0 \quad (21)$$

When the stock has known dividend payments before its maturity date, we have the following equality:

$$C + PV(D) + Xe^{-r_f T} = P + S_0 \quad (22)$$

Here, D is the present value of all dividends before their maturity date (T). The following Python program offers a graphical representation of the put-call parity:

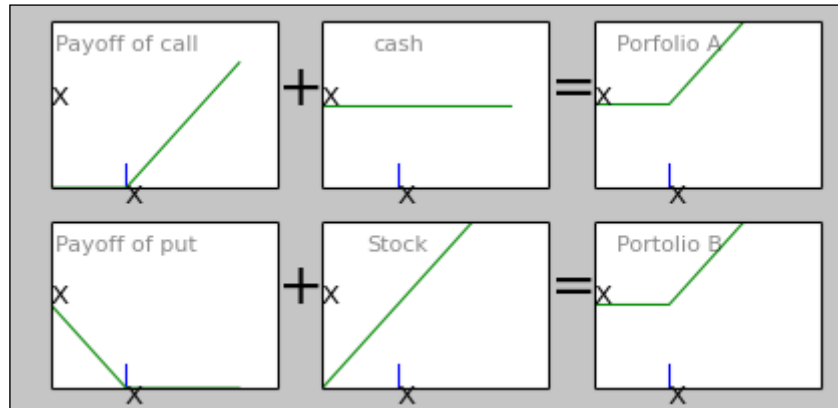
```
import pylab as pl
import numpy as np
x=10
sT=np.arange(0,30,5)
payoff_call=(abs(sT-x)+sT-x)/2
payoff_put=(abs(x-sT)+x-sT)/2
cash=np.zeros(len(sT))+x
def graph(text,text2=''):
    pl.xticks(())
    pl.yticks(())
    pl.xlim(0,30)
    pl.ylim(0,20)
    pl.plot([x,x],[0,3])
    pl.text(x,-2,"X");
    pl.text(0,x,"X")
    pl.text(x,x*1.7, text, ha='center', va='center',size=10, alpha=.5)
    pl.text(-5,10,text2,size=25)
pl.figure(figsize=(6, 4))
pl.subplot(2, 3, 1); graph('Payoff of call'); pl.plot(sT,payoff_call)
pl.subplot(2, 3, 2); graph('cash','+'); pl.plot(sT,cash)
pl.subplot(2, 3, 3); graph('Portfolio A ','='); pl.plot(sT,cash+payoff_
call)
```

```

pl.subplot(2, 3, 4); graph('Payoff of put '); pl.plot(sT,payoff_put)
pl.subplot(2, 3, 5); graph('Stock','+'); pl.plot(sT,sT)
pl.subplot(2, 3, 6); graph('Portfolio B','='); pl.plot(sT,sT+payoff_put)
pl.show()

```

The following is the output image:



Binomial tree (the CRR method) and its graphical representation

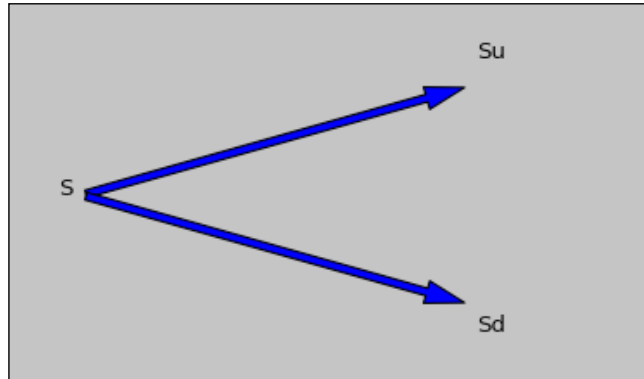
The binomial tree method was proposed by *Cox, Ross, and Rubinstein* in 1979. Because of this, it is also called the CRR method. Based on the CRR method, we have the following two-step approach. First, we draw a tree, such as the following one-step tree. If we assume that our current stock value is S , there are two outcomes $S*u$ and $S*d$, where $u > 1$ and $d < 1$, as shown in the following code:

```

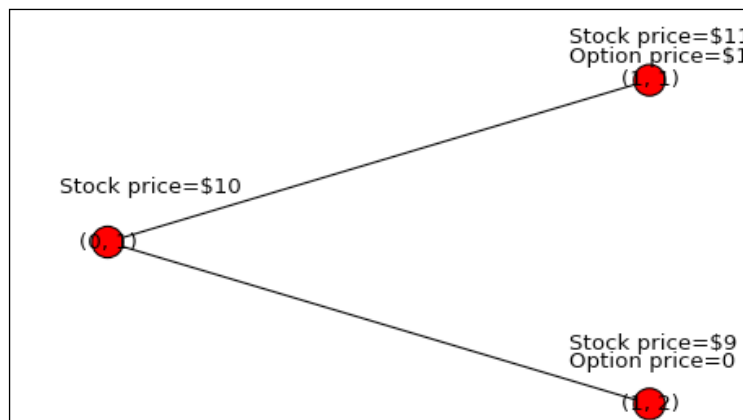
import matplotlib.pyplot as plt
xlim(0,1)
plt.figtext(0.18,0.5,'S')
plt.figtext(0.6,0.5+0.25,'Su')
plt.figtext(0.6,0.5-0.25,'Sd')
plt.annotate(' ',xy=(0.6,0.5+0.25), xytext=(0.1,0.5), arrowprops=dict(face
color='b',shrink=0.01))
plt.annotate(' ',xy=(0.6,0.5-0.25), xytext=(0.1,0.5), arrowprops=dict(face
color='b',shrink=0.01))
plt.axis('off')

```


The following is its corresponding graph:



Obviously, the simplest tree is a one-step tree. Assume that today's price is \$10, the exercise price is \$11, and a call option would mature in six months. In addition, assume that we know that the price would have two outcomes, moving up ($u = 1.15$) or moving down ($d = 0.9$). In other words, the final values are either \$11 or \$9. Based on such information, we have the following graph showing the prices for such a one-step binomial tree:



The following is the code to generate the preceding graph. This code is based on the code available at <http://litvak.eu/pyfi/>:

```
import networkx as nx
import matplotlib.pyplot as plt
plt.figtext(0.08,0.6,"Stock price=$20")
plt.figtext(0.75,0.91,"Stock price=$22")
plt.figtext(0.75,0.87,"Option price=$1")
```

```

plt.figtext(0.75,0.28,"Stock price=$18")
plt.figtext(0.75,0.24,"Option price=0")
n=1
def binomial_grid(n):
    G=nx.Graph()
    for i in range(0,n+1):
        for j in range(1,i+2):
            if i<n:
                G.add_edge((i,j),(i+1,j))
                G.add_edge((i,j),(i+1,j+1))
    posG={}
    for node in G.nodes():
        posG[node] = (node[0],n+2+node[0]-2*node[1])
    nx.draw(G,pos=posG)
binomial_grid(n)

```

In the preceding program, we generated the `binomial_grid()` function, since we will call this function many times later in the chapter. Since we knew beforehand that we would have two outcomes, we could choose an appropriate combination of stock and call options to get our final outcome with certainty, that is, the same terminal values. Assume that we choose an appropriate delta number of shares of the underlying security and one call to have the same terminal value at the end of one period, that is, $\Delta * 11.5 - 1 = 9\Delta + 0$ and thus, $\Delta = \frac{1}{11.5-9} = 0.4$. This means that if we long 0.4 shares and short one call option, our final wealth will be the same, $0.4 * 11.5 - 1 = 3.6$ when the stock moves up or $0.4 * 9 = 3.6$ when the stock moves down. Assume further that if the continuously compounded risk-free value is 0.12 percent, the value of today's portfolio will be equivalent to the discounted future certain value 4.5, $(0.4 * 10 - c = pv(3.6))$ that is, $c = 0.4 * 10 - e^{-0.012 * 0.5} * 3.6 = 0.42$. If we use Python, we will have the following result:

```

>>>round(0.4*10-exp(-0.012*0.5)*3.6,2)
0.42
>>>

```

For a two-step binomial tree, we have the following code:

```

import p4f
plt.figtext(0.08,0.6,"Stock price=$20")
plt.figtext(0.08,0.56,"call =7.43")
plt.figtext(0.33,0.76,"Stock price=$67.49")

```

```
plt.figtext(0.33,0.70,"Option price=0.93")
plt.figtext(0.33,0.27,"Stock price=$37.40")
plt.figtext(0.33,0.23,"Option price=14.96")
plt.figtext(0.75,0.91,"Stock price=$91.11")
plt.figtext(0.75,0.87,"Option price=0")
plt.figtext(0.75,0.6,"Stock price=$50")
plt.figtext(0.75,0.57,"Option price=2")
plt.figtext(0.75,0.28,"Stock price=$27.44")
plt.figtext(0.75,0.24,"Option price=24.56")
n=2
p4f.binomial_grid(n)
```

Based on the CRR method, we have the following procedure:

1. Draw an n-step tree.
2. At the end of the nth step, estimate terminal prices.
3. Calculate the option value at each node based on the terminal price, exercise, call, or put.
4. Discount it back one step, that is, from n to n-1, according to the risk-neutral probability.
5. Repeat the previous step until we find the final value at step 0.
The formulae for calculating u , d , and p are as follows:

$$u = e^{\sigma\sqrt{\Delta t}} \quad (23)$$

$$d = \frac{1}{u} = e^{-\sigma\sqrt{\Delta t}} \quad (24)$$

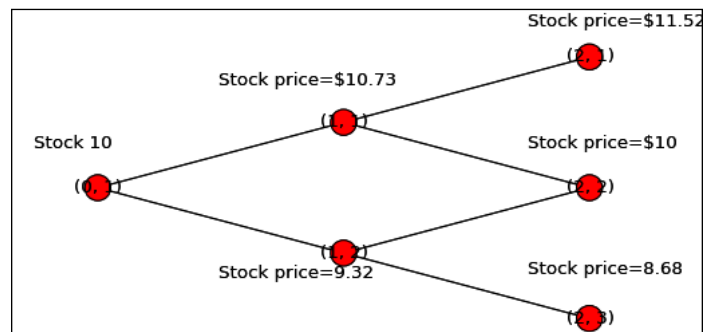
$$a = e^{(r-q)\Delta t} \quad (25)$$

$$p = \frac{a-d}{u-d} \quad (26)$$

$$v_i = pv_{i+1}^u + (1-p)v_{i+1}^d \quad (27)$$

Here, u is the upward movement, d is the downward movement, σ is the volatility of the underlying security, and r is the risk-free rate and Δt is the step, that is, $\Delta t = \frac{T}{n}$. Here, T is the maturity in years, n is the number of steps, q is the dividend yield,ⁿ and p is the risk-neutral probability of an upward movement. The `binomial_grid()` function is based on the functions shown under the one-step binomial tree graphical representation. Again, as we mentioned before that this function is included in the grand master file `p4fy.py`. The output graph is shown here. One obvious result is that the preceding Python program is very simple and straight. Let us use a two-step binomial tree to explain the whole process. Assume that the current stock price is \$10, the exercise price is \$10, the maturity is three months, the number of steps is two, the risk-free rate is 2 percent, and the volatility of the underlying security is 0.2. The following Python code would generate a two-step tree:

```
from math import sqrt,exp
s=10;r=0.02;sigma=0.2;T=3./12;x=10
n=2;deltaT=T/n;q=0
u=exp(sigma*sqrt(deltaT));d=1/u
a=exp((r-q)*deltaT)
p=(a-d)/(u-d)
su=round(s*u,2);suu=round(s*u*u,2)
sd=round(s*d,2);sdd=round(s*d*d,2)
sud=s
plt.figtext(0.08,0.6,'Stock '+str(s))
plt.figtext(0.33,0.76,"Stock price="+str(su))
plt.figtext(0.33,0.27,'Stock price='+str(sd))
plt.figtext(0.75,0.91,'Stock price="+str(suu))
plt.figtext(0.75,0.6,'Stock price="+str(sud))
plt.figtext(0.75,0.28,"Stock price="+str(sdd))
p4f.binomial_grid(n)
```



Now, we will use the risk-neutral probability to discount each value one step backward. The corresponding code and graph are given as follows:

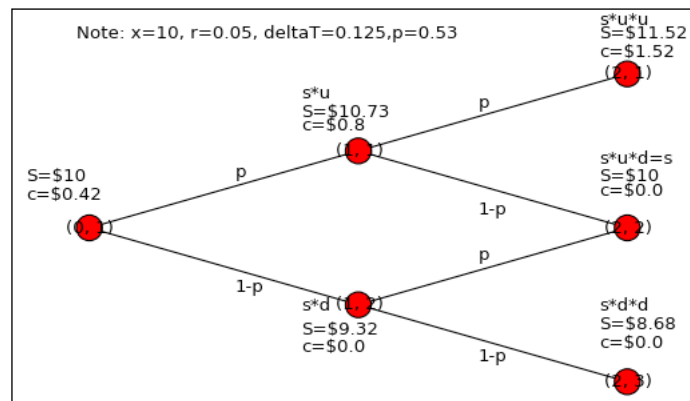
```
import p4f
s=10;x=10;r=0.05;sigma=0.2;T=3./12.;n=2;q=0 # q is dividend yield
deltaT=T/n # step
u=exp(sigma*sqrt(deltaT))
d=1/u
a=exp((r-q)*deltaT)
p=(a-d)/(u-d)
s_dollar='S=$';c_dollar='c=$'
p2=round(p,2)
plt.figtext(0.15,0.91,'Note: x='+str(x)+', r='+str(r)+', deltaT='+str(deltaT)+' ,p='+str(p2))
plt.figtext(0.35,0.61,'p')
plt.figtext(0.65,0.76,'p')
plt.figtext(0.65,0.43,'p')
plt.figtext(0.35,0.36,'1-p')
plt.figtext(0.65,0.53,'1-p')
plt.figtext(0.65,0.21,'1-p')
# at level 2
su=round(s*u,2);suu=round(s*u*u,2)
sd=round(s*d,2);sdd=round(s*d*d,2)
sud=s
c_suu=round(max(suu-x,0),2)
c_s=round(max(s-x,0),2)
c_sdd=round(max(sdd-x,0),2)
plt.figtext(0.8,0.94,'s*u*u')
plt.figtext(0.8,0.91,s_dollar+str(suu))
plt.figtext(0.8,0.87,c_dollar+str(c_suu))
plt.figtext(0.8,0.6,s_dollar+str(sud))
plt.figtext(0.8,0.64,'s*u*d=s')
plt.figtext(0.8,0.57,c_dollar+str(c_s))
plt.figtext(0.8,0.32,'s*d*d')
plt.figtext(0.8,0.28,s_dollar+str(sdd))
plt.figtext(0.8,0.24,c_dollar+str(c_sdd))
# at level 1
```

```

c_01=round((p*c_suu+(1-p)*c_s)*exp(-r*deltaT),2)
c_02=round((p*c_s+(1-p)*c_sdd)*exp(-r*deltaT),2)

plt.figtext(0.43,0.78,'s*u')
plt.figtext(0.43,0.74,s_dollar+str(su))
plt.figtext(0.43,0.71,c_dollar+str(c_01))
plt.figtext(0.43,0.32,'s*d')
plt.figtext(0.43,0.27,s_dollar+str(sd))
plt.figtext(0.43,0.23,c_dollar+str(c_02))
# at level 0 (today)
c_00=round(p*exp(-r*deltaT)*c_01+(1-p)*exp(-r*deltaT)*c_02,2)
plt.figtext(0.09,0.6,s_dollar+str(s))
plt.figtext(0.09,0.56,c_dollar+str(c_00))
p4f.binomial_grid(n)

```



Here, we explain a few values shown in the preceding graph. At the highest node ($s*u*u$), since the terminal stock price is 11.52 and the exercise price is 10, the call value is 1.52 ($11.52 - 10$). Similarly, on the $s*u*d=s$ node, the call value is 0 since $10 - 10 = 0$. For the call value 0.8, we have the following verification:

```

>>>p
0.5266253390068362
>>>deltaT
0.125
>>>v=(p*1.52+(1-p)*0)*exp(-r*deltaT)
>>>round(v,2)
0.80
>>>

```

The binomial tree method for European options

The following code is for the binomial tree method to price a European option:

```
from math import exp,sqrt
def binomialCall(s,x,T,r,sigma,n=100):
    deltaT = T /n
    u = exp(sigma * sqrt(deltaT))
    d = 1.0 / u
    a = exp(r * deltaT)
    p = (a - d) / (u - d)
    v = [[0.0 for j in xrange(i + 1)] for i in xrange(n + 1)]
    for j in xrange(i+1):
        v[n][j] = max(s * u**j * d**(n - j) - x, 0.0)
    for i in xrange(n-1, -1, -1):
        for j in xrange(i + 1):
            v[i][j]=exp(-r*deltaT)*(p*v[i+1][j+1]+(1.0-p)*v[i+1][j])
    return v[0][0]
```

To apply the function, we give it a set of input values. For the comparison, the result based on the Black-Scholes-Merton option model is also shown in the following code:

```
>>>binomialCall(40,42,0.5,0.1,0.2,1000)
2.1055845631835846
>>>bs_call(40,42,0.5,0.1,0.2)
2.2777803294555348
>>>
```

The binomial tree method for American options

Unlike the Black-Scholes-Merton option model, which could only be applied to European options, the binomial tree (CRR) method could be used to price American options. The only difference is that we have to consider the early exercise. The following is the code to price an American option using the binomial tree method:

```
from math import exp,sqrt
def binomialCallAmerican(s,x,T,r,sigma,n=100):
```

```

deltaT = T / n
u = exp(sigma * sqrt(deltaT))
d = 1.0 / u
a = exp(r * deltaT)
p = (a - d) / (u - d)
v = [[0.0 for j in xrange(i + 1)] for i in xrange(n + 1)]
for j in xrange(i+1):
    v[n][j] = max(s * u**j * d**(n - j) - x, 0.0)
for i in xrange(n-1, -1, -1):
    for j in xrange(i + 1):
        v1=exp(-r*deltaT)*(p*v[i+1][j+1]+(1.0-p)*v[i+1][j])
        v2=max(x-s,0)          # early exercise
        v[i][j]=max(v1,v2)
return v[0][0]

```

The key difference between pricing an American call option and a European call option is its early exercise opportunity. In the preceding program, the last few lines reflect this. For each node, we estimate two values, that is, v_1 is for the discounted value, and v_2 is the payoff from an early exercise. We will choose a higher value, that is, $\max(v_1, v_2)$. If using the same set of values to apply this binomial tree method to price an American call, we have the following value. It is understandable that the final result is higher than its European call counterpart:

```

>>>call=binomialCallAmerican(40,42,0.5,0.1,0.2,1000)
>>>round(call,2)
3.41
>>>

```

Hedging strategies

After selling a European call, we could hold Δ shares of the same stock to hedge our position. This is named delta hedge. Since delta (Δ) is a function of the underlying stock (S), to maintain an effective hedge, we have to rebalance our holding constantly. This is called dynamic hedging. The delta of a portfolio is the weighted deltas of individual securities in the portfolio. Note that when we short a security, its weight will be negative.

$$\Delta_{port} = \sum_{i=1}^n w_i \Delta_i \quad (28)$$

Assume that a US importer will pay 10 million pounds in three months. He or she is concerned with a potential depreciation of the US dollar against the UK pound. There are several ways to hedge such a risk: buy pounds now, enter a futures contract to buy 10 million pounds in three months with a fixed exchange rate, or buy call options with a fixed exchange rate as its exercise price. The first choice is costly since the importer does not need UK pounds today. Entering a future contract is risky as well, since an appreciation of the US dollar would cost the importer extra money. On the other hand, entering a call option will guarantee a maximum exchange rate today. At the same time, if the pound depreciates, the importer will reap the benefits. Such activities are called hedging since we take the opposite position of our risks.

For the currency options, we have the following equations:

$$d_1 = \frac{\ln\left(\frac{S_0}{x}\right) + \left(r_d - r_f + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}} \quad (29)$$

$$d_2 = \frac{\ln\left(\frac{S_0}{x}\right) + \left(r_d - r_f + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T} \quad (30)$$

$$c = S_0 * N(d_1) - X * e^{-rt} N(d_2) \quad (31)$$

$$p = X * e^{-rt} N(-d_2) - S_0 * N(-d_1) \quad (32)$$

Here, S_0 is the exchange rate of the US dollar per foreign currency, r_d is the domestic risk-free rate, and r_f is the foreign country's risk-free rate.

Summary

In this chapter, we discussed the Black-Scholes-Merton option model in detail. In particular, we covered the payoff and profit/loss functions and their graphical representations of call and put options; various trading strategies and their visual presentations, such as covered call, straddle, butterfly, calendar spread, normal distribution, standard normal distribution, and cumulative normal distribution; delta, gamma and other Greeks; the put-call parity; European versus American options; and the binomial tree method to price options and hedging.

In the next chapter, *Python Loops and Implied Volatility*, first we will discuss several types of Python loops. Then, we will explain how to find the implied volatility for a call or put option. In addition, we will explain how to download real-world option data from several public available sources. Using that data, we will estimate implied volatility, volatility skewness, and their applications.

Exercises

1. What is the difference between an American call and a European call?
2. What is the unit of rf in the Black-Scholes-Merton option model?
3. If we are given the annual rate of 3.4 percent, compounded semi-annually, what will the value of rf be that we should use for the Black-Scholes-Merton option model ?
4. How do we use options to hedge?
5. How do we treat predetermined cash dividends to price a European call?
6. Why is an American call worth more than a European call?
7. Assume you are a mutual fund manager and your portfolio's β is strongly correlated with the market. You are worried about the short-term fall in the market. What you could do to protect your portfolio?
8. The current price of stock A is \$38.5 and the strike prices for a call and a put options are both \$37. If the continuously compounded risk-free rate is 3.2 percent, maturity is six months, and the volatility of stock A is 0.25, what are the prices for a European call and put?
9. Use the put-call parity to verify the above solutions.
10. When the strike prices for call and put in (9.11) are different, can we apply the put-call parity?
11. For a set of input values, such as $s = 40$, $x = 40$, $t = 3 / 12 = 0.25$, $r = 0.05$, and $\sigma = 0.20$, using the Black-Scholes-Merton option model, we can estimate the value of the call. Now keeping all parameters constant, except s (the current price of a stock), show the relationship (a graph would be better) between calls and S .
12. Here is my portfolio: the more longer an underlying stock, the more longer a call option. Write a Python program showing the payoff function of this portfolio. Assume that the current stock price is \$40 and the strike price of the European call is \$45.

13. Bull spread with puts: buy a put on a stock with K_1 and sell a put with a strike price of K_2 ($K_1 < K_2$). Since $K_1 < K_2$, the put purchased is less valuable than the put sold, the Bull spread with puts involves upfront cash inflow. Write a Python program for payoff and profit/loss functions, and draw a graph for this.

9.14 Bear spread with puts: investors expect that the stock price is going to fall. Buy a put with K_2 and sell a put with K_1 (where $K_1 < K_2$). Since $K_1 < K_2$ due to which the put purchased is more valuable than the put sold, the bear spread with puts involves initial cash outflow. Write a Python program for payoff, profit/loss functions, and draw a graph for this.

15. Butterfly spread: buy two calls with K_1 and K_3 , and sell two calls with K_2 ($K_2 = 0.5(K_1 + K_3)$).

a) Show that this strategy involves an initial investment. In other words, prove that $C_1 + C_3 \geq 2C_2$. You form a portfolio of long C_1 , C_3 , and short $2 * C_2$.

b) Write a Python program to show its profit function.

16. You have the following portfolio: long 100 shares, short 77 calls on the same stocks, and long 88 puts on the same stocks. Assume that the current stock price is \$40, the strike price for the call is \$45 and the strike price of the put is \$38.

a) Write the payoff function for your portfolio.

b) What is the profit/loss function? Assume that the call and put premiums are \$3 and \$4 respectively.

c) Write a Python program for the preceding two tasks.

17. If we buy two puts and one call with the same exercise price, the strategy is called Strips. Write a Python program to show its profit/loss function.

18. If we buy one put and two calls with the same exercise price, the strategy is called "Strap". See the following graph. a) What is the expectation of such a strategy? b) Write a program to show its profit/loss graph.

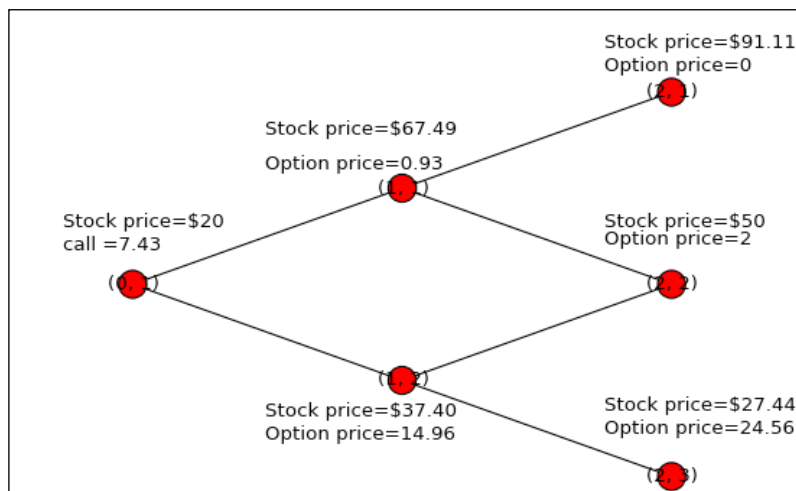
19. Write a program to draw a graph showing the relation of delta (Δ) on a European call on non-dividend stock with its underlying stock price (x axis).

20. The current stock price is \$30, the exercise price is \$30, the risk-free interest rate is 6 percent per annum, compounding semi-annually, the volatility is 25 percent per annum, the time to maturity is four months, and the underlying stock will pay \$1 dividends at end of one month and five months, respectively. What are the prices of a European call and a European put?

21. In this chapter, we have the following code to present the one-step binomial tree:

```
import p4f
plt.figtext(0.08,0.6,"Stock price=$20")
plt.figtext(0.08,0.56,"call =7.43")
plt.figtext(0.33,0.76,"Stock price=$67.49")
plt.figtext(0.33,0.70,"Option price=0.93")
plt.figtext(0.33,0.27,"Stock price=$37.40")
plt.figtext(0.33,0.23,"Option price=14.96")
plt.figtext(0.75,0.91,"Stock price=$91.11")
plt.figtext(0.75,0.87,"Option price=0")
plt.figtext(0.75,0.6,"Stock price=$50")
plt.figtext(0.75,0.57,"Option price=2")
plt.figtext(0.75,0.28,"Stock price=$27.44")
plt.figtext(0.75,0.24,"Option price=24.56")
n=2
p4f.binomial_grid(n)
```

The following is its related graph:



Simplify the preceding program to make it look like the following one:

```
import p4f
plt.figtext("Stock price=$20")
plt.figtext("call =7.43")
```

```
plt.figtext("Stock price=$67.49")
plt.figtext("Option price=0.93")
plt.figtext("Stock price=$37.40")
plt.figtext("Option price=14.96")
plt.figtext("Stock price=$91.11")
plt.figtext("Option price=0")
plt.figtext("Stock price=$50")
plt.figtext("Option price=2")
plt.figtext("Stock price=$27.44")
plt.figtext("Option price=24.56")
n=2
p4f.binomial_grid(n)
```

22. Write a Python program for the graphical representation of a three-step binomial tree.

10

Python Loops and Implied Volatility

In this chapter, we will study two topics: loops and implied volatility based on the European options (Black-Scholes-Merton option model) and American options. For the first topic, we have the `for` loop and `while` loop, the two most used loops. After presenting the definition of the implied volatility and explaining the logic behind it, we discuss three ways for its estimation: based on a `for` loop, on a `while` loop, and on a binary search. A binary search is the most efficient way to find a solution in such cases. However, the precondition to apply a binary search is that the objective function is monotone increasing or decreasing function of our target estimate. Fortunately, this is true since the value of an option price is an increasing function of the volatility.

In particular, we will cover the following topics:

- What is an implied volatility?
- Logic behind the estimation of an implied volatility
- Understanding the `for` loop, `while` loop, and their applications
- Nested (multiple) loops
- The estimation of multiple IRRs
- The mechanism of a binary search
- The estimation of an implied volatility based on an American call
- The `enumerate()` function
- Retrieving option data from Yahoo! Finance and from **Chicago Board Options Exchange (CBOE)**
- A graphical presentation of put-call ratios

Definition of an implied volatility

From the previous chapter, we know that for a set of input variables — S (the present stock price), X (the exercise price), T (the maturity date in years), r (the continuously compounded risk-free rate), and σ (the volatility of the stock, that is, the annualized standard deviation of its returns) — we could estimate the price of a call option based on the Black-Scholes-Merton option model. Recall that to price a European call option, we have the following Python code of five lines:

```
from scipy import log,exp,sqrt,stats
def bs_call(S,X,T,r,sigma):
    d1=(log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    d2 = d1-sigma*sqrt(T)
    return S*stats.norm.cdf(d1)-X*exp(-r*T)*stats.norm.cdf(d2)
```

After entering a set of five values, we can estimate the call price as follows:

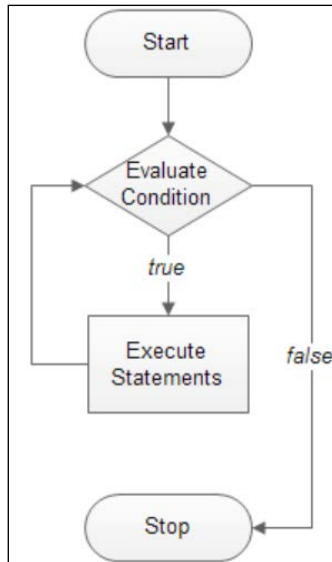
```
>>>bs_call(40,40,0.5,0.05,0.25)
3.3040017284767735
```

On the other hand, if we know S , X , T , r , and c , how can we estimate σ ? Here, σ is our implied volatility. In other words, if we are given a set values such as $S=40$, $X=40$, $T=0.5$, $r=0.05$, and $c=3.30$, we should find out the value of σ , and it should be equal to 0.25 . In this chapter, we will learn how to estimate the implied volatility.

Actually, the underlying logic to figure out the implied volatility is very simple: trial and error. Let's use the previous example as an illustration. We have five values — $S=40$, $X=40$, $T=0.5$, $r=0.05$, and $c=3.30$. The basic design is that after inputting 100 different sigmas, plus the first four input values shown earlier, we have 100 call prices. The implied volatility is the sigma that achieves the smallest absolute difference between the estimated call price and 3.30. Of course, we could increase the number of trials to achieve a higher precision, that is, more decimal places. Alternatively, we could adopt another conversion criterion: we stop when the absolute difference between our estimated call price and the given call value is less than a critical value, such as 1 cent, that is, $|c-3.30| < 0.01$. Since it is not a good idea to randomly pick up 100 or 1,000 different sigmas, we systematically choose those values, that is, use a loop by selecting those sigmas systematically. Next, we will discuss two types of loops: a `for` loop and a `while` loop.

Understanding a for loop

A `for` loop is one of the most used loops in many computer languages. The following flow diagram demonstrates how a loop works. Usually, we start with an initial value. Then, we test a condition. If the condition is false, the program stops. Otherwise, we execute a set of commands:



The simplest example is given as follows:

```
>>>for i in range(1,5):  
    print i
```

Running these two lines will print 1, 2, 3, and 4. We have to be careful with the `range()` function since the last number, 5, will not be printed in Python. Thus, if we intend to print from 1 to n , we have to use the following code:

```
>>>n=10  
>>>for i in range(1,n+1):  
    print i
```

In the previous two examples, the default incremental value is 1. If we intend to use an incremental value other than 1, we have to specify it as follows:

```
>>>for i in xrange(1,10,3):  
    print i
```


The output values will be 1, 4, and 7. Along the same lines, if we want to print 5 to 1, that is, in descending order, the incremental value should be -1:

```
>>>for j in xrange(5,1,-1):
    print j
```

Estimating the implied volatility by using a for loop

First, we should generate a Python program to estimate the call price based on the Black-Scholes-Merton option model as shown in the following code:

```
from scipy import log,exp,sqrt,stats
def bs_call(S,X,T,r,sigma):
    """Objective: estimate call for stock with one known dividend
        S: current stock price
        T : maturity date in years
        r : risk-free rate
        sigma: volatility
    """
    d1=(log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    d2 = d1-sigma*sqrt(T)
    return S*stats.norm.cdf(d1)-X*exp(-r*T)*stats.norm.cdf(d2)
S=40
K=40
T=0.5
r=0.05
c=3.30
for i in range(200):
    sigma=0.005*(i+1)
    diff=c-bs_call(S,K,T,r,sigma)
    if abs(diff)<=0.01:
        print(i,sigma, diff)
```

To make our lives a little bit easier, we could include the `bs_call()` function in a general or master program such as `p4f.py`. Then, our code would be simpler and easier to understand, as shown in the following code snippet:

```
import p4f
S=40
K=40
T=0.5
r=0.05
c=3.30
for i in range(200):
    sigma=0.005*(i+1)
    diff=c-p4f.bs_call(S,K,T,r,sigma)
    if abs(diff)<=0.01:
        print(i,sigma, diff)
```

In the preceding program, we used the same set of input values as the example shown earlier. Thus, our expected implied volatility is 0.25. The logic of this program is that we use the trial-and-error method to feed our Black-Scholes-Merton option model with many different sigmas (volatilities). For a given sigma (volatility), when the difference between our estimated call price and the given call price is less than 0.01, we stop. That sigma (volatility) will be our implied volatility. The output from the earlier program is shown as follows:

```
(49, 0.25, -0.0040060797372882817)
>>>
```

The first number, 49, is the value of the variable `i`, and 0.25 is the implied volatility. The last value is the difference between our estimated call value and the given call price of \$3.30.

Implied volatility function based on a European call

Ultimately, we could write a function to estimate the implied volatility based on a European call. To save space, we remove all comments and examples from the program as shown:

```
def implied_vol_call(S,X,T,r,c):
    from scipy import log,exp,sqrt,stats
    for i in range(200):
```

```
sigma=0.005*(i+1)
d1=(log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
d2 = d1-sigma*sqrt(T)
diff=c-(S*stats.norm.cdf(d1)-X*exp(-r*T)*stats.norm.cdf(d2))
if abs(diff)<=0.01:
    return i,sigma, diff
```

With a set of input values, we could apply the previous program easily as follows:

```
>>>implied_vol_call(40,40,0.5,0.05,3.3)
(49, 0.25, -0.0040060797372882817)
>>>
```

Implied volatility based on a put option model

Similarly, we could estimate an implied volatility based on a European put option model. In the following program, we design a function named `implied_vol_put_min()`. There are several differences between this function and the previous one. First, the current function depends on a put option instead of a call. Thus, the last input value is a put premium instead of a call premium. Second, the conversion criterion is that an estimated price and the given put price have the smallest difference. In the previous function, the conversion criterion is when the absolute difference is less than 0.01. In a sense, the current program will guarantee an implied volatility while the previous program does not guarantee an output:

```
def implied_vol_put_min(S,X,T,r,p):
    from scipy import log,exp,sqrt,stats
    implied_vol=1.0
    min_value=100.0
    for i in xrange(1,10000):
        sigma=0.0001*(i+1)
        d1=(log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
        d2 = d1-sigma*sqrt(T)
        put=X*exp(-r*T)*stats.norm.cdf(-d2)-S*stats.norm.cdf(-d1)
        abs_diff=abs(put-p)
        if abs_diff<min_value:
            min_value=abs_diff
            implied_vol=sigma
            k=i
```

```

        put_out=put
    print 'k, implied_vol, put, abs_diff'
return k,implied_vol, put_out,min_value

```

Let's use a set of input values to estimate the implied volatility. After that, we will explain the logic behind the previous program. Assume, $S=40$, $X=40$, $T=12$ months, $r=0.1$, and the put price is \$1.50, as shown in the following code:

```

>>>implied_vol_put_min(40,40,1.,0.1,1.501)
k, implied_vol, put, abs_diff
(1999, 0.2, 1.5013673553027349, 0.00036735530273501737)
>>>

```

The implied volatility is 20 percent. The logic is that we assign a big value, such as 100, to a variable called `min_value`. For the first sigma with a value of 0.0002, we have an almost zero put value. Thus, the absolute difference is 1.50, which is smaller than 100. Because of this, our `min_value` variable will be replaced with the value 1.50. We continue this way until we go through the loop. For the recorded minimum value, its corresponding sigma will be our implied volatility.

We could optimize the previous program by defining some intermediate values. For example, in the previous program, we estimate $\ln(S/X)$ 10,000 times. Actually, we define a new variable such as `log_S_over_X`, estimate its value just once, and use it 10,000 times. This is true for $\sigma^2/2$, and $\sigma\sqrt{T}$.

The enumerate() function

When generating a **Net Present Value (NPV)** function, we need to estimate the present value of all future and present cash flows as shown in the following formula:

$$NPV = \sum_{i=0}^n \frac{cashflow_i}{(1+R)^i} \quad (1)$$

In a sense, for each cash flow, we need two values: i and cash flow at i . For these cases, we could apply the `enumerate()` function as shown in the following NPV function:

```

def npv_f(rate, cashflows):
    total = 0.0
    for i, cashflow in enumerate(cashflows):
        total += cashflow / (1 + rate)**i
    return total

```

The `enumerate()` function would generate a pair of indices, starting from 0, and its corresponding value. With a set of input values for the discount rate and a cash flow array, we could apply the NPV function as follows:

```
>>>c=[-100.0, 60.0, 60.0, 60.0]
>>>r=0.1
>>>npv=npv_f(r,c)
>>>round(npv,2)
49.21
>>>
```

Estimation of IRR via a for loop

In the first two chapters, we learned that we could apply the **Internal Rate of Return (IRR)** rule to evaluate our project with a set of forecasted current and future cash flows. Based on a `for` loop, we could calculate the IRR of our project. The two related functions, `NPV()` and `IRR_f()`, are shown as follows:

```
def npv_f(rate, cashflows):
    total = 0.0
    for i, cashflow in enumerate(cashflows):
        total += cashflow / (1 + rate)**i
    return total
```

Here, the key is finding out what kinds of values the intermediate variables `i` and `cashflow` would take. From the previous section, we know that `i` will take values from 0 to the number of cash flows and that `cashflow` would take all values from the variable called `cashflows`. The `total+=x` statement is equivalent to `total=total+x`. One issue is that if we enter -1 as our rate, the function would not work. We could add an `if` command to prevent this from happening (refer to the succeeding solution for the `IRR()` function). The second potential issue is that when the second input variable contains NaN, the `npv_f()` function would fail. For these cases, we could use the `isnan()` function contained in the `NumPy` module as shown in the following code:

```
def IRR_f(cashflows, iterations=100):
    if len(cashflows)==0:
        print 'number of cash flows is zero'
        return -99
    rate = 1.0
```

```

investment = cashflows[0]
for i in range(1, iterations+1):
    rate *= (1 - npv_f(rate, cashflows) / investment)
return rate

```

The underlying assumption for the previous code is that the first investment is our initial investment, while all future cash flows are cash inflows. This means that the NPV and the discount rate have a reverse relationship. For a given discount rate, if its corresponding NPV is positive, we are supposed to increase the discount rate in order to find a zero NPV – thus, the current rate times a value greater than one. Notice that the investment is a cash outflow. Thus, it has a negative sign. The value of $(1 - \text{npv_f}(\text{rate}, \text{cashflows}) / \text{investment})$ will be greater than one. On the other hand, if the estimated NPV is negative, we should reduce our discount rate, that is, the current rate times a value lesser than one. Assume that we have the following cash flows, what are the corresponding IRRs?

```

>>>cashflows=[-100,50,60,20,50]
>>>x=IRR_f(cashflows)
>>>round(x,3)
0.304
>>>

```

Estimation of multiple IRRs

In the earlier example, the direction of cash flows changes just once. Thus, there exists one IRR. However, when the directions of cash flows change more than once, we might have multiple IRRs. Assume that we have the following cash flows: `cashflows=[55,-50,-50,-50,100]`. Since the directions of cash flows change twice, we expect two IRRs. If we apply the previous IRR function, we could locate just one cash flow as shown in the following code:

```

>>>cashflows=[55,-50,-50,-50,100]
>>>round(IRR_f(cashflows),3)
0.337
>>>

```

Here, we apply the same logic by using many different discount rates to find out which two rates make the **NPV (Net Present Value)** become zero. The Python program to estimate multiple IRRs is shown as follows:

```

def IRRs_f(cash_flows):
    n=1000

```

```
r=range(1,n)
epsilon=abs(mean(cash_flows)*0.01)
irr=[-99.00]
j=1
npv=[]
for i in r: npv.append(0)
lag_sign=sign(npv_f(float(r[0]*1.0/n*1.0),cash_flows))
for i in range(1,n-1):
    interest=float(r[i]*1.0/n*1.0)
    npv[i]=npv_f(interest,cash_flows)
    s=sign(npv[i])
    if s*lag_sign<0:
        lag_sign=s
        if j==1:
            irr=[interest]
            j=2
        else:
            irr.append(interest)

return irr
```

We could call the function easily as follows:

```
>>>cashflows=[55,-50,-50,-50,100]
>>>IRRs_f(cashflows)
[0.072, 0.337]
>>>
```

Understanding a while loop

In the following program, the first line assigns an initial value to *i*. The second line defines a condition for? when the `while` loop should stop. The last one increases the value of *i* by 1. The `i+=1` statement is equivalent to `i=i+1`. Similarly, `t**=2` should be interpreted as `t=t**2`:

```
i=1
while(i<=4):
    print(i)
    i+=1
```

The key for a `while` loop is that an exit condition should be satisfied at least once. Otherwise, we would enter an infinitive loop. For example, if we run the following scripts, we would enter an infinitive loop. When this happens, we can use `Ctrl + C` to stop it:

```
i=1
while(i!=2.1):
    print(i)
    i+=1
```

In the previous program, we compare two real numbers for equality. It is not a good idea to use the equals sign for two real/float/double numbers. The next example is related to the famous Fibonacci series: the summation of the previous two numbers is the current one:

Fibonacci series = 1,1,2,3,5,8,13,.....

The Python code for computing the Fibonacci series is given as follows:

```
def fib(n):
    """Print a Fibonacci series up to n.
    """
    a, b = 0, 1
    while a < n:
        print a,
        a, b = b, a+b
```

When `n` is 1,000, we get the following results:

```
>>>fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>>
```

Using keyboard commands to stop an infinitive loop

Sometimes, because of carelessness or other reasons, we might end up with an infinitive loop (refer to the following program). Our original intention is to print just four numbers ranging from one to four. However, since we forgot to add 1 to the variable `i` after each printing, the exit condition will never be satisfied, that is, it leads to an infinitive loop. For such cases, we could use `Ctrl + C` or `Ctrl + Enter` to stop such an infinitive loop:


```
i=1
While i<5:
    Print i
>>>
```

If these commands do not work, then use *Ctrl + Alt + Del* to launch the **Task Manager**, choose **Python**, and then click on **End Task**.

Estimating implied volatility by using a while loop

This time, we use the Black-Scholes-Merton put option model and a `while` loop to estimate the implied volatility. First, we present the put option model as follows:

```
def bs_put(S,X,T,rf,sigma):
    from scipy import log,exp,sqrt,stats
    d1=(log(S/X)+(rf+sigma*sigma/2.)*T)/(sigma*sqrt(T))
    d2 = d1-sigma*sqrt(T)
    return X*exp(-rf*T)*stats.norm.cdf(-d2)-S*stats.norm.cdf(-d1)
```

To apply the function, we input a set of values for `S`, `X`, `T`, `rf`, and `sigma` as follows:

```
>>>put=bs_put(40,40,0.5,0.05,0.2)
>>>round(put,2)
1.77
>>>
```

The following program uses both a `while` loop and the put option to estimate the implied volatility. Here, we assume that the previous European put option function is included in the `p4y.py` master program (module):

```
import p4f
S=40;K=40;T=0.5;r=0.05;sigma=0.2;p=1.77
diff=1;i=1
while abs(diff)>0.01:
    sigma=0.005*(i+1)
    diff=p-p4f.bs_put(S,K,T,r,sigma)
    i+=1
print('i, implied-vol, diff')
print(i,sigma, diff)
```

From the following output, we know that the implied volatility is 0.2, the same as we estimated using the Black-Scholes-Merton call option model. Again, we could verify this using 0.2 as our input value for the volatility to confirm whether we have the same put price:

```
i, implied-vol, diff
(40, 0.2, 0.0021120877944480476)
>>>
```

In the following program, we use `break` to exit an infinite loop. The condition of one equals one is always true. The only hope to stop the loop is based on the `break` clause. Another advantage of such a conversion criterion is that we don't have to consider what an appropriate difference level is. Sometimes, choosing an appropriate scale is not easy since option prices vary:

```
import p4f
S=40;K=40;T=0.5;r=0.05;sigma=0.2;p=1.77
diff=1;i=1
sigma_old=0.005
sign_1=sign(p-bs_put(S,K,T,r,sigma_old))
while(1):
    sigma=0.0001*(i+1)
    sign_2=sign(p-p4f.bs_put(S,K,T,r,sigma))
    i+=1
    if sign_1*sign_2<0:
        break
    else:
        sigma_old=sigma
print('i, implied-vol, diff')
print(i,(sigma_old+sigma)/2, diff)
```

The `sign()` function returns 1 if the input is greater than 1. It returns -1 if the input is less than zero. A sample implementation of the `sign()` function is given as follows:

```
>>>sign(-2)
-1
>>>sign(2)
1
>>>sign(0)
0
```

Nested (multiple) for loops

For a two-dimensional matrix, you need two loops with variables *i* and *j* shown as follows:

```
n1=2
n2=3
for x in xrange(1, n1+1):
    for y in xrange(1, n2+1):
        print '%d * %d = %d' % (x, y, x*y)
```

We can use two `while` loops or the combination of a `for` loop and a `while` loop to accomplish the same task.

Estimating implied volatility by using an American call

Since almost all exchange listed stock options are American options, we show the following program to estimate an implied volatility based on an American call option:

```
from math import exp,sqrt
def binomialCallAmerican(s,x,T,r,sigma,n=100):
    deltaT = T /n
    u = exp(sigma * sqrt(deltaT))
    d = 1.0 / u
    a = exp(r * deltaT)
    p = (a - d) / (u - d)
    v = [[0.0 for j in xrange(i + 1)] for i in xrange(n + 1)]
    for j in xrange(i+1):
        v[n][j] = max(s * u**j * d**(n - j) - x, 0.0)
    for i in xrange(n-1, -1, -1):
        for j in xrange(i + 1):
            v1=exp(-r*deltaT)*(p*v[i+1][j+1]+(1.0-p)*v[i+1][j])
            v2=max(s-x,0)
            v[i][j]=max(v1,v2)
    return v[0][0]
```

The previous Python program is used to estimate an American call option based on the binomial-tree method, or CRR method. Based on the input values, we first calculate u , d , and p , where u represents the up movement, d represents the down movement, and p is the risk-neutral probability. The first loop estimates the option values at the end of the tree for all nodes. With the second set of double loops, we move backward from the last step to time zero. The variable $v1$ is the discounted two-call option from the previous step while $v2$ is the early exercise premium since it is an American option:

```
def implied_vol_American_call(s,x,T,r,c):
    implied_vol=1.0
    min_value=1000
    for i in range(1000):
        sigma=0.001*(i+1)
        c2=binomialCallAmerican(s,x,T,r,sigma)
        abs_diff=abs(c2-c)
        if abs_diff<min_value:
            min_value=abs_diff
            implied_vol=sigma
            k=i
    return implied_vol
```

To test the program, we could estimate an American call by inputting a set of values, including σ , and then estimate the implied volatility as follows:

```
>>>binomialCallAmerican(150,150,2./12.,0.003,0.2)
4.908836114170818
>>>implied_vol_American_call(150,150,2./12.,0.003,4.91)
0.2
>>>
```

Measuring efficiency by time spent in finishing a program

The following program measures how much time, in seconds, is required to finish a program. The function used is `time.clock()`:

```
import time
start = time.clock()
n=10000000
```

```
for i in range(1,n):
    k=i+i+2
diff= (time.clock() - start)
print round(diff,2)
```

The total time we need to finish the previous meaningless loop is about 1.59 seconds.

The mechanism of a binary search

To estimate the implied volatility, the logic underlying the earlier methods is to run the Black-Scholes-Merton option model a hundred times and choose the `sigma` value that achieves the smallest difference between the estimated option price and the observed price. Although the logic is easy to understand, such an approach is not efficient since we need to call the Black-Scholes-Merton option model a few hundred times. To estimate a few implied volatilities, such an approach would not pose any problems. However, under two scenarios, such an approach is problematic. First, if we need higher precision, such as `sigma=0.25333`, or we have to estimate several million implied volatilities, we need to optimize our approach. Let's look at a simple example.

Assume that we randomly pick up a value between one and 5,000. How many steps do we need to match this value if we sequentially run a loop from one to 5,000? A binomial search is the $\log(n)$ worst-case scenario when linear search is the n worst-case scenario. Thus, to search a value from one to 5,000, a linear search would need 5,000 steps (average 2,050) in a worst-case scenario, while a binary search would need 12 steps (average 6) in a worst-case scenario. The following Python program performs a binary search:

```
def binary_search(x, target, my_min=1, my_max=None):
    if my_max is None:
        my_max = len(x) - 1
    while my_min <= my_max:
        mid = (my_min + my_max) // 2
        midval = x[mid]
        if midval < target:
            my_min = my_mid + 1
        elif midval > target:
            my_max = mid - 1
        else:
            return mid
    raise ValueError
```

The next program generates a list of unique words from the Bible. Then, we conduct a binary search to find the location for a given word. First, we have to download the Bible in text format:

1. Go to <http://printkjv.ifbweb.com/>.
2. Download the zip file that includes the text file.

Unzip the downloaded zip file, and we will see a text file called `AV1611Bible.txt`. Assuming that the text file is saved under `C:\temp\`:

```
from string import maketrans
import pandas as pd
word_freq = {}
word_list = open("c:/temp/AV1611Bible.txt", "r").read().split()
for word in word_list:
    word = word.translate(maketrans("", ""), '!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~0123456789')
    if word.startswith('-'): word = word.replace('-', '')
    if len(word): word_freq[word] = word_freq.get(word, 0) + 1
keys = sorted(word_freq.keys())
x=pd.DataFrame(keys)
x.to_pickle('c:/temp/uniqueWords.pickle')
```

This time, we compare words instead of values, the program for which is given as follows:

```
def binaryText(x, target, my_min=1, my_max=None):
    if my_max is None:
        my_max = len(x) - 1
    while my_min <= my_max:
        mid = (my_min + my_max)//2
        midval = x.iloc[mid]
        if midval.values < target:
            my_min = mid + 1
        elif midval.values > target:
            my_max = mid - 1
        else:
            return mid
    raise ValueError
```

In the previous program, `x.iloc[mid]` gives us the value since `x` is in a `Data.Frame` format:

```
>>>x.iloc[600]
      0
600  Baasha
>>>binaryText(x, 'Baasha', 1)
600
>>>
```

If users have issues in downloading the Bible discussed earlier, they could download a file in a Pandas' pickle format from <http://canisius.edu/~yany/uniqueWords.pickle>. Assuming that such a dataset is saved under `C:\temp\`, the following code can be used to perform the binary search:

```
>>>x=load("c:/temp/uniqueWords.pickle")
>>>x.iloc[610]
      0
610  Bahurim
>>>binaryText(x, 'Bahurim', 1)
610
>>>
```

Sequential versus random access

If we have daily stock data, we could have them saved in different patterns. One way is to save them as stock ID, date, high, low, opening price, closing price, and trading volume. We could sort our stock ID and save them one after another. We have two ways to write a Python program to access IBM data: sequential access and random access. For sequential access, we read one line and check its stock ID to see if it matches our ticker. If not, we go to the next line, until we find our data. Such a sequential search is not efficient, especially when our dataset is huge, such as several gigabits. It is a good idea to generate an index file, such as IBM, 1,000, 2,000. Based on this information, we know that IBM's data is located from line 1,000 to line 2,000. Thus, to retrieve IBM's data, we could jump to line 1,000 immediately without having to go through the first 999 lines. This is called random access.

Looping through an array/DataFrame

The following program shows how to print all values in an array:

```
import numpy as np
x = np.arange(10).reshape(2,5)
for y in np.nditer(x):
    print y
```

For another example of going through all tickers, we download a dataset called `yanMonthly.pickle` from <http://canisius.edu/~yany/yanMonthly.pickle>. Assume again that the downloaded dataset is saved under `C:\temp\`. We could use the following program to retrieve the dataset and run a loop to print a dozen tickets:

```
x=load('c:/temp/yanMonthly.pickle')
stocks=x.index.unique()
for item in stocks[:10]:
    print item
    # add your codes here
```

The output of the previous code is shown as follows:

```
000001.SS
A
AA
AAPL
BC
BCF
C
CNC
COH
CPI
```

The previous program has no real meaning since we could simply type the following codes to see those tickers. However, we could add our own related codes as follows:

```
>>>stocks[0:10]
array(['000001.SS', 'A', 'AA', 'AAPL', 'BC', 'BCF', 'C', 'CNC', 'COH',
      'CPI'], dtype=object)
>>>
```


Assignment through a for loop

The following program assigns values to a variable:

```
>>>x=[0.0 for i in xrange(5)]
>>>x
[0.0, 0.0, 0.0, 0.0, 0.0]
>>>
```

The previous assignment is quite simple. Actually, we could use `x=zeros(5)` to achieve the same objective. The following program is an extension of the previous code:

```
>>>v = [[0.0 for j in xrange(i + 1)] for i in xrange(4 + 1)]
>>>v
[[0.0], [0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0], [0.0, 0.0,
0.0, 0.0, 0.0]]
>>>len(v)
5
>>>v[0]
[0.0]
>>>v[1]
[0.0, 0.0]
>>>v[3]
[0.0, 0.0, 0.0, 0.0]
>>>
```

Looping through a dictionary

An example related to a dictionary is given as follows:

```
>>>market_cap= {"IBM":200.97, "MSFT":311.30, "WMT":253.91, "C": 158.50}
```

For each stock, we have its corresponding market capitalization. Each pair has a key and a value. Again, stocks' names are keys while their market capitalizations are values. To show their keys and values, refer to the following code:

```
>>>market_cap.keys()
['C', 'IBM', 'MSFT', 'WMT']
>>>market_cap.values()
[158.5, 200.97, 311.3, 253.91]
```

To show both keys and values, we use the `items()` function as follows:

```
>>>market_cap.items()
[('C', 158.5), ('IBM', 200.97), ('MSFT', 311.3), ('WMT', 253.91)]
>>>
```

The following program demonstrates how to loop through a dictionary:

```
>>>market_cap= {"IBM":200.97, "MSFT":311.30, "WMT":253.91, "C": 158.50}
>>>for k,v in market_cap.items():
...     print k,v
...
C 158.5
IBM 200.97
MSFT 311.3
```

Retrieving option data from CBOE

The **Chicago Board Options Exchange** (CBOE) trades options and futures. There is a lot of free data available on the CBOE web pages. For example, we could enter a ticker to download its related option data. To download IBM's option data, we perform the following two steps:

1. Go to <http://www.cboe.com/DelayedQuote/QuoteTableDownload.aspx>.
2. Enter **IBM**, then click on **Download**.

The first few lines are shown in the following table. According to the original design, the put data is arranged side by side with the call data. In order to have a clearer presentation, we move the put option data under the call:

IBM (International Business Machines)	172.8	-0.57						
December 15, 2013 @ 10:30 ET	Bid	172.51	Ask	172.8	Size	2x6	Vol	4184836
Calls	Last Sale	Net	Bid	Ask	Vol	Open Int		
13 December 125.00 (IBM1313L125)	0	0	46.75	50	0	0		
13 December 125.00 (IBM1313L125-4)	0	0	46.45	50.45	0	0		

13 Dec 125.00 (IBM1313L125-8)	0	0	46.2	50.3	0	0
13 Dec 125.00 (IBM1313L125-A)	0	0	46.5	50.5	0	0
13 Dec 125.00 (IBM1313L125-B)	0	0	46.15	50.15	0	0
13 Dec 125.00 (IBM1313L125-E)	0	0	46.25	50.3	0	0
Puts	Last Sale	Net	Bid	Ask	Vol	Open Int
13 Dec 125.00 (IBM1313X125)	0	0	0	0.03	0	0
13 Dec 125.00 (IBM1313X125-4)	0	0	0	0.03	0	0
13 Dec 125.00 (IBM1313X125-8)	0	0	0	0.03	0	0
13 Dec 125.00 (IBM1313X125-A)	0	0	0	1.72	0	0
13 Dec 125.00 (IBM1313X125-B)	0	0	0	0.04	0	0
13 Dec 125.00 (IBM1313X125-E)	0	0	0	0.03	0	0

Assume that our dataset is saved under `C:\temp\`. The following code would retrieve the data from that dataset:

```
import numpy as np
x=pd.read_csv('c:/temp/QuoteData.dat',skiprows=2,header='infer')
y=np.array(x)
n=len(y)
```

To show the first and last several lines, we have the following code:

```
>>>print y[0:2]
[['13 Dec 125.00 (IBM1313L125)' 0.0 0.0 46.75 50.0 0L 0L
 '13 Dec 125.00 (IBM1313X125)' 0.0 0.0 0.0 0.03 0L 0L nan]
[['13 Dec 125.00 (IBM1313L125-4)' 0.0 0.0 46.45 50.45 0L 0L
 '13 Dec 125.00 (IBM1313X125-4)' 0.0 0.0 0.0 0.03 0L 0L nan]]
>>>print y[n-3:n-1]
```

```
[['16 Jan 250.00 (IBM1615A250-S)' 2.6 0.0 1.1 2.95 0L 219L
  '16 Jan 250.00 (IBM1615M250-S)' 66.0 0.0 80.75 83.65 0L 11L nan]
['16 Jan 250.00 (IBM1615A250-X)' 2.87 0.0 1.03 2.95 0L 219L
  '16 Jan 250.00 (IBM1615M250-X)' 0.0 0.0 80.75 83.65 0L 11L nan]]
>>>
```

Retrieving option data from Yahoo! Finance

There are many sources of option data that we could use for our investments, research, or teaching. One of them is Yahoo! Finance. To retrieve option data for IBM, we have the following procedure:

1. Go to <http://finance.yahoo.com>.
2. Type IBM in the search box (top left-hand side).
3. Click on **Options** on the left-hand side.

The web page address of Yahoo! Finance is <http://finance.yahoo.com/q/op?s=IBM+Options>. The screenshot of this web page is shown as follows:

International Business Machines Corporation (IBM) - NYSE							Add to Portfolio
172.80 +0.57 (0.33%) Dec 13, 4:01PM EST After Hours : 172.80 0.00 (0.00%) Dec 13, 7:13PM EST							
Options							Get Options for: nu
View By Expiration: Dec 13 Jan 14 Feb 14 Apr 14 Jun 14 Jul 14 Jan 15 Jan 16							
Call Options				Expire at close Friday, December 27, 2013			
Strike	Symbol	Last	Chg	Bid	Ask	Vol	Open Int
115.00	IBM131221C00115000	70.40	0.00	56.75	60.25	6	6
140.00	IBM131221C00140000	45.00	0.00	31.75	33.65	3	5
150.00	IBM131221C00150000	29.20	0.00	22.15	23.65	1	1

The following program will download option data from Yahoo! Finance:

```
>>>from pandas.io.data import Options
>>>ticker='IBM'
>>>x = Options(ticker)
>>>calls, puts = x.get_options_data()
```

We can use the `head()` and `tail()` functions to view the first and last several lines of the retrieved data:

```
>>>calls.head()
```

	Strike	Symbol	Last	Chg	Bid	Ask	Vol	Open	Int
0	100	IBM140118C00100000	78.25	0	83.65	87.10	2		12
1	125	IBM140118C00125000	53.30	0	58.70	61.90	2		1
2	130	IBM140118C00130000	48.30	0	53.70	56.90	1		1
3	135	IBM140118C00135000	45.80	0	48.70	50.70	5		13
4	140	IBM140118C00140000	35.35	0	43.70	46.95	10		125

```
>>>calls.tail()
```

	Strike	Symbol	Last	Chg	Bid	Ask	Vol	Open	Int
54	280	IBM140118C00280000	0.15	0	NaN	0.03	10		499
55	285	IBM140118C00285000	0.06	0	NaN	0.03	1		72
56	290	IBM140118C00290000	0.05	0	NaN	0.03	13		76
57	295	IBM140118C00295000	0.07	0	NaN	0.03	1		92
58	300	IBM140118C00300000	0.04	0	NaN	0.03	3		370

```
>>>puts.head()
```

	Strike	Symbol	Last	Chg	Bid	Ask	Vol	Open	Int
0	95	IBM140118P00095000	0.01	0.00	NaN	0.03	1		360
1	100	IBM140118P00100000	0.02	0.00	NaN	0.02	15		2569
2	105	IBM140118P00105000	0.02	0.00	NaN	0.03	12		1527
3	110	IBM140118P00110000	0.02	0.00	NaN	0.03	10		1176
4	115	IBM140118P00115000	0.01	0.04	NaN	0.03	10		1149

```
>>>puts.tail()
```

	Strike	Symbol	Last	Chg	Bid	Ask	Vol	Open	Int
58	230	IBM140118P00230000	44.97	0	42.95	46.3	4		76
59	235	IBM140118P00235000	53.20	0	47.95	51.3	1		8
60	240	IBM140118P00240000	67.56	0	52.95	56.3	3		1
61	245	IBM140118P00245000	43.11	0	57.95	61.3	0		9
62	250	IBM140118P00250000	51.75	0	62.95	66.3	5		27

```
>>>
```

Different expiring dates from Yahoo! Finance

For each stock, we have different exercise prices and various expiring dates. In the previous program, we retrieve the option data which has the shortest expiration date. To retrieve other expiration dates, we have to specify the month and year. First, let's look at some different web pages for different expirations:

- <http://finance.yahoo.com/q/op?s=IBM&m=2014-01?s=IBM140118C00100000+Options>
- <http://finance.yahoo.com/q/op?s=IBM&m=2014-02>
- <http://finance.yahoo.com/q/op?s=IBM&m=2014-03>
- <http://finance.yahoo.com/q/op?s=IBM&m=2014-04>
- <http://finance.yahoo.com/q/op?s=IBM&m=2014-06>
- <http://finance.yahoo.com/q/op?s=IBM&m=2014-07>
- <http://finance.yahoo.com/q/op?s=IBM&m=2015-01>
- <http://finance.yahoo.com/q/op?s=IBM&m=2016-01>

For example, we intend to download option data for year=2014 and month=2 with the following program:

```
from pandas.io.data import Options
ticker='IBM'
month=2
year=2014
x = Options(ticker)
calls, puts = x.get_options_data(month,year)
```

To show a few lines, we can use the `head()` function as follows:

```
>>>calls.head()
   Strike      Symbol  Last   Chg   Bid   Ask  Vol  Open  Int
0    150  IBM140222C00150000  30.00  0.00  33.95  37.00    8     10
1    160  IBM140222C00160000  23.31  0.00  24.10  27.00    3     62
2    165  IBM140222C00165000  18.47  0.00  19.85  21.35    1     50
3    170  IBM140222C00170000  16.60  0.00  16.15  16.70   23    467
4    175  IBM140222C00175000  12.20  0.04  11.95  12.50    5    767
>>>
```

Retrieving the current price from Yahoo! Finance

Using the following Python program, we can retrieve the current stock prices for a given set of ticker symbols:

```
import urllib
import re
stocks=['ibm', 'dell', 'goog']
for i in range(len(stocks)):
    file = urllib.urlopen("http://finance.yahoo.com/q?s=" +stocks[i] +
"&q1=1")
    text = file.read()
    pattern='<span id="yfs_l84_' + stocks[i] + '>(.*?)</span>'
    price = re.findall(re.compile(pattern), text)
print "For ",stocks[i].upper(), " the price is ", price
```

If the previous code ran on December 29, 2013, the output would be as follows:

```
>>>runfile('C:/tem/43750S_10_35_yahoo_price.py', wdir=r'C:/temp')
For IBM the price is ['185.08']
For DELL the price is ['13.86']
For GOOG the price is ['1,118.40']
>>>
```

The put-call ratio

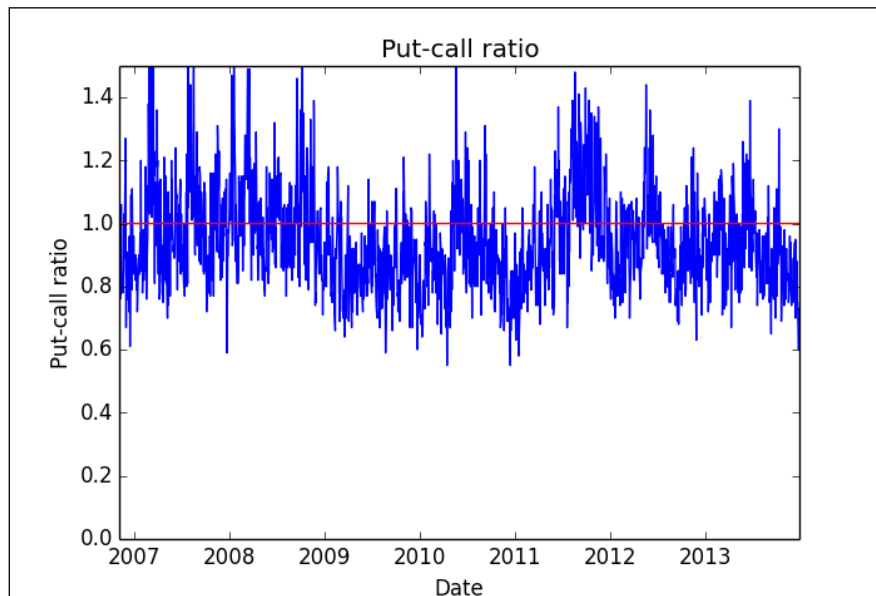
The put-call ratio represents the perception of investors jointly towards the future. If there is no obvious trend, that is, we expect a normal future, then the put-call ratio should be close to one. On the other hand, if we expect a much brighter future, the ratio should be lower than one. The following code shows a ratio of this type over the years. First, we have to download the data from CBOE. Perform the following steps:

1. Go to <http://www.cboe.com/>.
2. Click on **Quotes & Data** on the menu bar.
3. Click on **CBOE Volume & Put/Call Ratios**.
4. Click on **CBOE Total Exchange Volume and Put/Call Ratios (11-01-2006 to present)** under **Current**.

Assume that the file named `totalpc.csv` is saved under `C:\temp\`. The code is given as follows:

```
import pandas as pd
from matplotlib.pyplot import *
data=pd.read_csv('c:/temp/totalpc.csv',skiprows=2,index_col=0,parse_
dates=True)
data.columns=('Calls','Puts','Total','Ratio')
x=data.index
y=data.Ratio
y2=ones(len(y))
title('Put-call ratio')
xlabel('Date')
ylabel('Put-call ratio')
ylim(0,1.5)
plot(x, y, 'b-')
plot(x, y2,'r')
show()
```

The corresponding graph is shown in the following figure:



The put-call ratio for a short period with a trend

Based on the preceding program, we could choose a shorter period with a trend as shown in the following code:

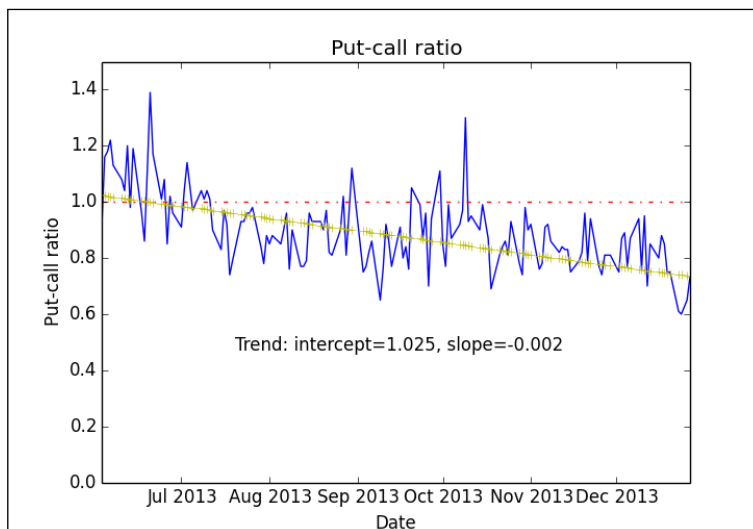
```
import pandas as pd
from matplotlib.pyplot import *
import matplotlib.pyplot as plt
from datetime import datetime
import statsmodels.api as sm

data=pd.read_csv('c:/temp/totalpc.csv',skiprows=2,index_col=0,parse_
dates=True)
data.columns=('Calls','Puts','Total','Ratio')
begdate=datetime(2013,6, 1)
enddate=datetime(2013,12,31)

data2=data[(data.index>=begdate) & (data.index<=enddate)]
x=data2.index
y=data2.Ratio
x2=range(len(x))
x3=sm.add_constant(x2)
model=sm.OLS(y,x3)
results=model.fit()
#print results.summary()
alpha=round(results.params[0],3)
slope=round(results.params[1],3)
y3=alpha+dot(slope,x2)
y2=ones(len(y))
title('Put-call ratio')
xlabel('Date')
ylabel('Put-call ratio')
ylim(0,1.5)
plot(x, y, 'b-')
plot(x, y2,'r-.')
plot(x,y3,'y+')
plt.figtext(0.3,0.35,'Trend: intercept='+str(alpha)+'',
```

```
slope='+str(slope))
show()
```

The corresponding graph is shown in the following figure:



Summary

In this chapter, we introduced different types of loops. Then, we demonstrated how to estimate the implied volatility based on a European option (Black-Scholes-Merton option model) and on an American option. We discussed the `for` loop and the `while` loop, and their applications. For a given set of input values, such as current stock price, the exercise price, the time to maturity, the continuously compounded risk-free rate, and a call price (or put price), we showed how to estimate a stock's implied volatility. In terms of efficiency, we explained the binary search method and compared it with other approaches when estimating an implied volatility. In addition, we demonstrated how to download option data, such as put-call ratio, from Yahoo! Finance and the CBOE web page.

In the next chapter, we will focus on applications of Monte Carlo simulations on option pricing. Using random numbers drawn from a normal distribution, we could mimic the movements of a stock for a given set of mean and standard deviations. After that, we will simulate the terminal values of a stock and its related payoffs for a call or for a put. The mean of those discounted terminal values using the risk-free rate as our discount rate would be our option price.

Exercises

1. How many types of loops are present in Python? What are the differences between them?
2. What are the advantages of using a `for` loop versus a `while` loop? What are the disadvantages?
3. Based on a `for` loop, write a Python program to estimate the implied volatility. For a given set of values $S=35$, $X=36$, $rf=0.024$, $T=1$, $\sigma=0.13$, and $c=2.24$, what is the implied volatility?
4. Write a Python program based on the Black-Scholes-Merton option model put option model to estimate the implied volatility.
5. Should we get different volatilities based on the Black-Scholes-Merton option model's call and put?
6. For a stock with multiple calls, we could estimate its implied volatility based on its call or put. Based on the Black-Scholes-Merton option model, could we get different values?
7. When estimating a huge number of implied volatilities, such as 5,000 stocks, how can we make our process more efficient?
8. We could apply the binary search method to estimate an implied volatility based on the Black-Scholes-Merton option model. Could we apply it to estimate multiple IRRs to speed up our process? Explain.
9. Is it necessary that we use the binary tree method to estimate an implied volatility?
10. After reading the chapter, we know that we could use the following function to estimate an implied volatility based on call:

```
def implied_vol_call(S,X,T,r,c):  
    from scipy import log,exp,sqrt,stats  
    for i in range(200):  
        sigma=0.005*(i+1)  
        d1=(log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))  
        d2 = d1-sigma*sqrt(T)  
        diff=c-(S*stats.norm.cdf(d1)-X*exp(-r*T)*stats.norm.cdf(d2))  
        if abs(diff)<=0.01:  
            return i,sigma, diff
```

With certain sets of input values, we could get no output; refer to the following examples:

```
>>>implied_vol_call(25,40,1,0.05,3.3)
>>>implied_vol_call(25,26,1,0.05,3.3)
>>>implied_vol_call(40,40,5,0.05,3.3)
```

Find reasons and modify this program accordingly.

11. From this chapter, we learn that we could use the following program to estimate an implied volatility based on the Black-Scholes-Merton option model:

```
def implied_vol_put_min(S,X,T,r,p):
    from scipy import log,exp,sqrt,stats
    implied_vol=1.0
    min_value=100.0
    for i in range(1,10000):
        sigma=0.0001*(i+1)
        d1=(log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
        d2 = d1-sigma*sqrt(T)
        put=X*exp(-r*T)*stats.norm.cdf(-d2)-S*stats.norm.cdf(-d1)
        abs_diff=abs(put-p)
        if abs_diff<min_value:
            min_value=abs_diff
            implied_vol=sigma
            k=i
            put_out=put
    print 'k, implied_vol, put, abs_diff'
    return k,implied_vol, put_out,min_value
```

Using the knowledge that the put premiums are a monotone function of the volatility, modify this program to make it more efficient.

12. What is wrong with the following program?

```
i=1
def while_less_than_n(n,k=1):
    i=1
    while True:
        if i<n:
```

```
        print i
        i+=k
    else:
        return 'done'
```

13. Write a Python program to estimate an implied volatility based on an American put.

14. Write a Python program to download option data from Yahoo! Finance. Then, estimate the implied volatility by using the average of bid and ask as call or put prices.

15. Perform the following steps to download the put-call ratio data from CBOE:

1. Go to <http://www.cboe.com/>.
2. Click on **Quotes & Data** on the menu bar.
3. Click on **CBOE Volume & Put/Call Ratios**.
4. Click on **CBOE Total Exchange Volume and Put/Call Ratios (11-01-2006 to present)** under **Current**.

Write a Python program to print the first and last dates.

16. Write a Python program to retrieve the put-call ratio, and draw a graph. The syntax of such a function can be `put_call_graph(path,begdate,enddate)`. To apply the function, we specify the path and two dates, for example, `put_call_graph('c:/temp/totalpc.csv',20130601,20131231)`.

11

Monte Carlo Simulation and Options

In finance, we study the trade-off between risk and return. The common definition of risk is uncertainty. For example, when evaluating a potential profitable project, we have to predict many factors in the life of the project, such as the annual sales, price of the final product, prices of raw materials, salary increase of employees, inflation rate, cost of borrowing, cost of new equity, and economic status. For those cases, the Monte Carlo simulation could be used to simulate many possible future outcomes, events, and their various combinations. In this chapter, we focus on the applications of the Monte Carlo simulation to price various options.

In this chapter, we will cover the following topics:

- Generating random numbers from standard normal distribution and normal distribution
- Generating random numbers from a uniform distribution
- A simple application: estimate π by the Monte Carlo simulation
- Generating random numbers from a Poisson distribution
- Bootstrapping with/without replacements
- The lognormal distribution and simulation of stock price movements
- Simulating terminal stock prices
- Simulating an efficient portfolio and an efficient frontier
- Using the Monte Carlo simulation to price European options that have closed-form solutions, that is, replicate the Black-Scholes-Merton model
- Path independent versus path dependent options
- Long term expected return forecast

- Exotic options
- Pricing lookback options with floating strikes
- Sobol sequence

Generating random numbers from a standard normal distribution

Normal distributions play a central role in finance. A major reason is that many finance theories, such as option theory and applications, are based on the assumption that stock returns follow a normal distribution. It is quite often that we need to generate n random numbers from a standard normal distribution. For this purpose, we have the following two lines of code:

```
>>>import scipy as sp
>>>x=sp.random.standard_normal(size=10)
```

The basic random numbers in SciPy/NumPy are created by Mersenne Twister PRNG in the `numpy.random` function. The random numbers for distributions in `numpy.random` are in cython/pyrex and are pretty fast. To print the first few observations, we use the `print()` function as follows:

```
>>>print x[0:5]
[-0.55062594 -0.51338547 -0.04208367 -0.66432268  0.49461661]
>>>
```

Alternatively, we could use the following code:

```
>>>import scipy as sp
>>>x=sp.random.normal(size=10)
```

This program is equivalent to the following one:

```
>>>import scipy as sp
>>>x=sp.random.normal(0,1,10)
```

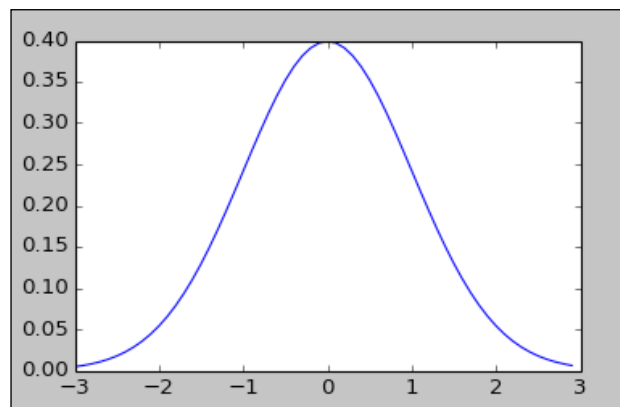
The first input is for mean, the second input is for standard deviation, and the last one is for the number of random numbers, that is, the size of the dataset. The default settings for mean and standard deviations are 0 and 1. We could use the `help()` function to find out the input variables. To save space, we show only the first few lines:

```
>>>help(sp.random.normal)
Help on built-in function normal:
```

```
normal(...)
    normal(loc=0.0, scale=1.0, size=None)
```

Drawing random samples from a normal (Gaussian) distribution

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently, is often called the bell curve because of its characteristic shape; refer to the following graph:



Again, the density function for a standard normal distribution is defined as follows:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \quad (1)$$

Generating random numbers with a seed

Sometimes, we like to produce the same random numbers repeatedly. For example, when a professor is explaining how to estimate the mean, standard deviation, skewness, and kurtosis of five random numbers, it is a good idea that students could generate exactly the same values as their instructor. Another example would be that when we are debugging our Python program to simulate a stock's movements, we might prefer to have the same intermediate numbers. For such cases, we use the `seed()` function as follows:

```
>>>import scipy as sp
>>>sp.random.seed(12345)
>>>x=sp.random.normal(0,1,20)
```



```
>>>print x[0:5]
[-0.20470766  0.47894334 -0.51943872 -0.5557303  1.96578057]
>>>
```

In this program, we use 12345 as our seed. The value of the seed is not important. The key is that the same seed leads to the same random values.

Generating n random numbers from a normal distribution

To generate n random numbers from a normal distribution, we have the following code:

```
>>>import scipy as sp
>>>sp.random.seed(12345)
>>>x=sp.random.normal(0.05,0.1,50)
>>>print x[0:5]
[ 0.02952923  0.09789433 -0.00194387 -0.00557303  0.24657806]
>>>
```

The difference between this program and the previous one is that the mean is 0.05 instead of 0, while the standard deviation is 0.1 instead of 1. The density of a normal distribution is defined by the following equation, where μ is the mean and σ is the standard deviation. Obviously, the standard normal distribution is just a special case of the normal distribution shown as follows:

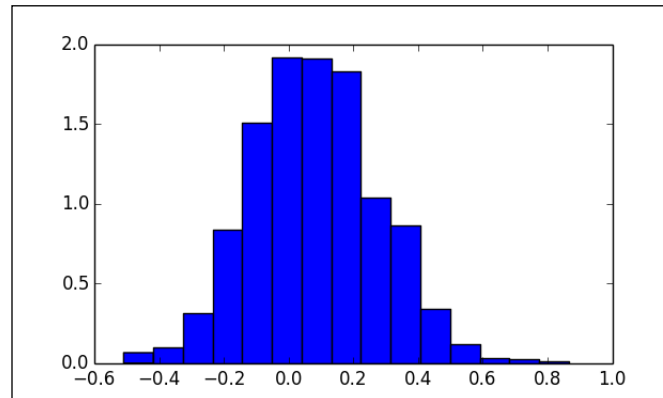
$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (2)$$

Histogram for a normal distribution

A histogram is used intensively in the process of analyzing the properties of datasets. To generate a histogram for a set of random values drawn from a normal distribution with specified mean and standard deviation, we have the following code:

```
>>>import scipy as sp
>>>import matplotlib.pyplot as plt
>>>sp.random.seed(12345)
>>>x=sp.random.normal(0.08,0.2,1000)
>>>plt.hist(x, 15, normed=True)
>>>plt.show()
```

The resultant graph is presented as follows:



Graphical presentation of a lognormal distribution

When returns follow a normal distribution, the prices would follow a lognormal distribution. The definition of a lognormal distribution is as follows:

$$f(x; \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln(x)-\mu)^2}{2\sigma^2}} \quad (3)$$

The following code shows three different lognormal distributions with three pairs of parameters, such as $(0, 0.25)$, $(0, 0.5)$, and $(0, 1.0)$. The first parameter is for mean (μ), while the second one is for standard deviation, σ :

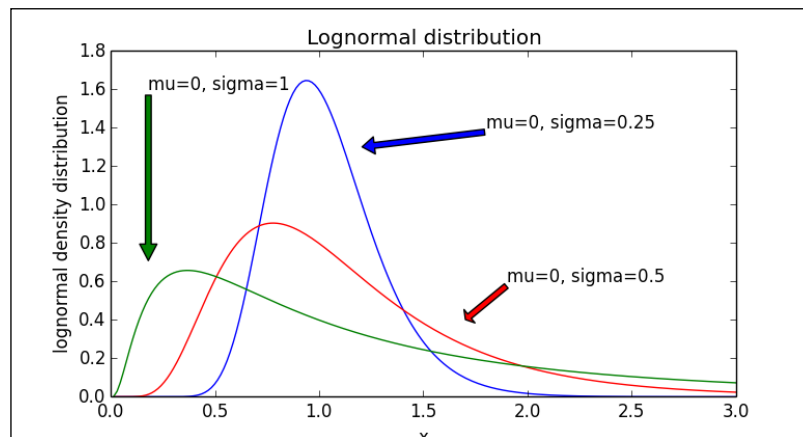
```
import scipy.stats as sp
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(0,3,200)
mu=0
sigma0=[0.25,0.5,1]
color=['blue','red','green']
target=[(1.2,1.3),(1.7,0.4),(0.18,0.7)]
start=[(1.8,1.4),(1.9,0.6),(0.18,1.6)]
for i in range(len(sigma0)):
    sigma=sigma0[i]
```

```

y=1/(x*sigma*sqrt(2*pi))*exp(-(log(x)-mu)**2/(2*sigma*sigma))
plt.annotate('mu='+str(mu)+' , sigma='+str(sigma), xy=target[i],
xytext=start[i],
            arrowprops=dict(facecolor=color[i],shrink=0.01),)
plt.plot(x,y,color[i])
plt.title('Lognormal distribution')
plt.xlabel('x')
plt.ylabel('lognormal density distribution')
plt.show()

```

The corresponding three graphs are put together to illustrate their similarities and differences:



Generating random numbers from a uniform distribution

When we plan to randomly choose m stocks from n available stocks, we could draw a set of random numbers from a uniform distribution. To generate 10 random numbers between one and 100 from a uniform distribution, we have the following code. To guarantee that we generate the same set of random numbers, we use the `seed()` function as follows:

```

>>>import scipy as sp
>>>sp.random.seed(123345)
>>>x=sp.random.uniform(low=1,high=100,size=10)

```

Again, `low`, `high`, and `size` are the three keywords for the three input variables. The first one specifies the minimum, the second one specifies the high end, while the `size` gives the number of the random numbers we intend to generate. The first five numbers are shown as follows:

```
>>>print x[0:5]
[ 30.32749021  20.58006409   2.43703988  76.15661293  75.06929084]
>>>
```

Using simulation to estimate the pi value

It is a good exercise to estimate π by the Monte Carlo simulation. Let's draw a square with $2R$ as its side. If we put the largest circle inside the square, its radius will be R . In other words, the areas for those two shapes have the following equations:

$$S_{circle} = \pi * R^2 \quad (4)$$

$$S_{square} = (2R) * (2R) = 4R^2 \quad (5)$$

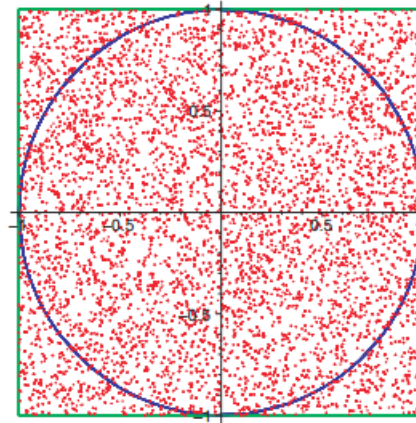
By dividing equation (4) by equation (5), we have the following result:

$$\frac{S_{circle}}{S_{square}} = \frac{\pi}{4}$$

In other words, the value of π will be $4 * S_{circle}/S_{square}$. When running the simulation, we generate n pairs of x and y from a uniform distribution with a range of zero and 0.5. Then we estimate a distance that is the square root of the summation of the squared x and y , that is, $d = \sqrt{x^2 + y^2}$. Obviously, when d is less than 0.5 (value of R), it will fall into the circle. We can imagine throwing a dart that falls into the circle. The value of the π will take the following form:

$$\pi = 4 * \frac{\text{number of darts in circle}}{\text{number of darts in square, ie., number of simulation}} \quad (6)$$

The following graph illustrates these random points within a circle and within a square:



The Python program to estimate the value of pi is presented as follows:

```
import scipy as sp
n=100000
x=sp.random.uniform(low=0,high=1,size=n)
y=sp.random.uniform(low=0,high=1,size=n)
dist=sqrt(x**2+y**2)
in_circle=dist[dist<=1]
our_pi=len(in_circle)*4./n
print ('pi=',our_pi)
print('error (%)=', (our_pi-pi)/pi)
```

The estimated pi value would change whenever we run the previous code as shown in the following code, and the accuracy of its estimation depends on the number of trials, that is, n :

```
('pi=', 3.15)
('error (%)=', 0.0026761414789406262)
>>>
```

Generating random numbers from a Poisson distribution

To investigate the impact of private information, Easley, Kiefer, O'Hara, and Paperman (1996) designed a **(PIN) Probability of informed trading** measure that is derived based on the daily number of buyer-initiated trades and the number of seller-initiated trades. The fundamental aspect of their model is to assume that order arrivals follow a Poisson distribution. The following code shows how to generate n random numbers from a Poisson distribution:

```
import scipy as sp
import matplotlib.pyplot as plt
x=sp.random.poisson(lam=1, size=100)
#plt.plot(x, 'o')
a = 5. # shape
n = 1000
s = np.random.power(a, n)
count, bins, ignored = plt.hist(s, bins=30)
x = np.linspace(0, 1, 100)
y = a*x**(a-1.)
normed_y = n*np.diff(bins)[0]*y
plt.plot(x, normed_y)
plt.show()
```

Selecting m stocks randomly from n given stocks

Based on the preceding program, we could easily choose 20 stocks from 500 available securities. This is an important step if we intend to investigate the impact of the number of randomly selected stocks on the portfolio volatility as shown in the following code:

```
import scipy as sp
n_stocks_available=500
n_stocks=20
x=sp.random.uniform(low=1,high=n_stocks_available,size=n_stocks)
```

```

y=[]
for i in range(n_stocks):
    y.append(int(x[i]))
#print y
final=unique(y)
print final
print len(final)

```

In the preceding program, we select 20 numbers from 500 numbers. Since we have to choose integers, we might end up with less than 20 values, that is, some integers appear more than once after we convert real numbers into integers. One solution is to pick more than we need. Then choose the first 20 integers. An alternative is to use the `randrange()` and `randint()` functions. In the next program, we choose n stocks from all available stocks. First, we download a dataset from <http://canisius.edu/~yany/yanMonthly.pickle>:

```

n_stocks=10
x=load('c:/temp/yanMonthly.pickle')
x2=unique(np.array(x.index))
x3=x2[x2<'ZZZZ']      # remove all indices
sp.random.seed(1234567)
nonStocks=['GOLDPRICE','HML','SMB','Mkt_Rf','Rf','Russ3000E_D','US_DEBT',
           'Russ3000E_X','US_GDP2009dollar','US_GDP2013dollar']
x4=list(x3)
for i in range(len(nonStocks)):
    x4.remove(nonStocks[i])
k=sp.random.uniform(low=1,high=len(x4),size=n_stocks)
y,s=[],[]
for i in range(n_stocks):
    index=int(k[i])
    y.append(index)
    s.append(x4[index])
final=unique(y)
print final
print s

```

In the preceding program, we remove non-stock data items. These non-stock items are a part of data items. First, we load a dataset called `yanMonthly.pickle` that includes over 200 stocks, gold price, GDP, unemployment rate, **SMB (Small Minus Big)**, **HML (High Minus Low)**, risk-free rate, price rate, market excess rate, and Russell indices.

The `.pickle` extension means that the dataset has a type from `Pandas`. Since `x.index` would present all indices for each observation, we need to use the `unique()` function to select all unique IDs. Since we only consider stocks to form our portfolio, we have to move all market indices and other non-stock securities, such as `HML` and `US_DEBT`. Because all stock market indices start with a caret (^), we use less than `zzzz` to remove them. For other IDs that are between `A` and `Z`, we have to remove them one after another. For this purpose, we use the `remove()` function available for a list variable. The final output is shown as follows:

```
[ 1  2  4 10 17 20 21 24 31 70]
['IO', 'A', 'AA', 'KB', 'DELL', 'IN', 'INF', 'IBM', 'SKK', 'BC']
>>>
```

Bootstrapping with/without replacements

Assume that we have the historical data, such as price and return, for a stock. Obviously, we could estimate their mean, standard deviation, and other related statistics. What are their expected annual mean and risk next year? The simplest, maybe naïve way is to use the historical mean and standard deviation. A better way is to construct the distribution of annual return and risk. This means that we have to find a way to use historical data more effectively to predict the future. In such cases, we could apply the bootstrapping methodology. For example, for one stock, we have its last 20-year monthly returns, that is, 240 observations.

To estimate next year's 12 monthly returns, we need to construct a return distribution. First, we choose 12 returns randomly from the historical return set without replacements and estimate their mean and standard deviations. We repeat this procedure 5,000 times. The final output will be our return-standard distribution. Based on such a distribution, we could estimate other properties as well. Similarly, we could do so with replacements.

One of the useful functions present in `SciPy` is called `permutation()`. Assume that we have 10 numbers from one to 10 (inclusive of one and 10). We could call the `permutation()` function to reshuffle them as follows:

```
import numpy as np
x=range(1,11)
print x
for i in range(5):
    y=np.random.permutation(x)
    print y
```


The output of this code is shown as follows:

```
***
[ 7  6  9  1  2 10  5  8  3  4]
[ 8  2  4  6  1  9 10  3  7  5]
[ 4  9  1  6  8  2  5  7 10  3]
[ 2  1  6  9 10  5  7  4  8  3]
[ 7  3  9  5 10  4  8  6  2  1]
>>>
```

Based on the `permutation()` function, we could define a function with three input variables: data, number of observations we plan to choose from the data randomly, and whether we choose to bootstrap with or without replacement as shown in the following code:

```
import numpy as np
def boots_f(data,n_obs,replacement=None):
    n=len(data)
    if (n<n_obs):
        print "n is less than n_obs"
    else:
        if replacement==None:
            y=np.random.permutation(data)
            return y[0:n_obs]
        else:
            y=[]
    for i in range(n_obs):
        k=np.random.permutation(data)
        y.append(k[0])
    return y
```

The constraint specified in the previous program is that the number of given observations should be larger than the number of random returns we plan to pick up. This is true for the bootstrapping without the replacement method. For the bootstrapping with the replacement method, we could relax this constraint; refer to the related exercise.

Distribution of annual returns

It is a good application to estimate annualized return distribution and represent it as a graph. To make our exercise more meaningful, we download Microsoft's daily price data. Then, we estimate its daily returns and convert them into annual ones. Based on those annual returns, we generate its distribution by applying bootstrapping with replacements 5,000 times as shown in the following code:

```
from matplotlib.finance import quotes_historical_yahoo
import matplotlib.pyplot as plt
import numpy as np
import scipy as sp

# Step 1: input area
ticker='MSFT'          # input value 1
begdate=(1926,1,1)     # input value 2
enddate=(2013,12,31)  # input value 3
n_simulation=5000      # input value 4

# Step 2: retrieve price data and estimate log returns
x=quotes_historical_yahoo(ticker,begdate,enddate,asobject=True,adjusted=True)

logret = log(x.aclose[1:]/x.aclose[:-1])

# Step 3: estimate annual returns
date=[]
d0=x.date

for i in range(0,size(logret)):
    date.append(d0[i].strftime("%Y"))

y=pd.DataFrame(logret,date,columns=['logret'],dtype=float64)
ret_annual=exp(y.groupby(y.index).sum())-1
ret_annual.columns=['ret_annual']
n_obs=len(ret_annual)

# Step 4: estimate distribution with replacement
sp.random.seed(123577)
final=zeros(n_obs,dtype=float)

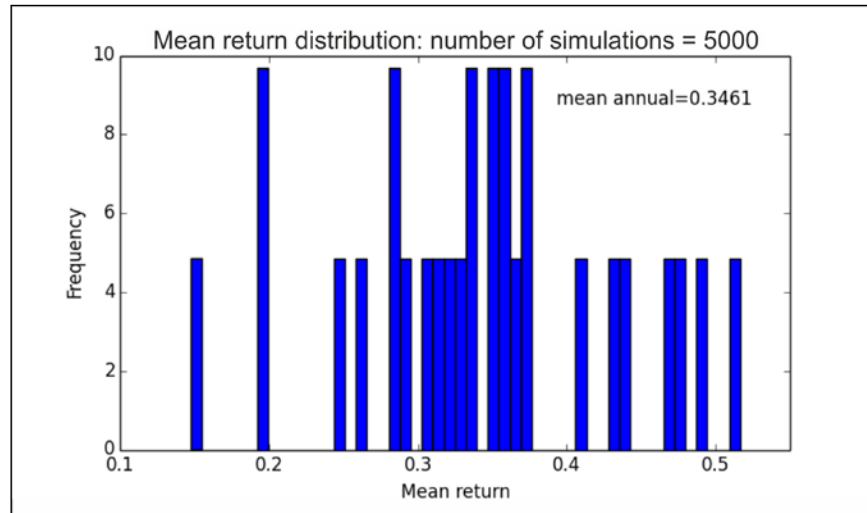
for i in range(0,n_obs):
    x=sp.random.uniform(low=0,high=n_obs,size=n_obs)
    y=[]
```

```

    for j in range(n_obs):
        y.append(int(x[j]))
    z=np.array(ret_annual)[y]
    final[i]=mean(z)
# step 5: graph
plt.title('Mean return distribution: number of simulations =' +str(n_
simulation))
plt.xlabel('Mean return')
plt.ylabel('Frequency')
mean_annual=round(np.mean(np.array(ret_annual)),4)
plt.figtext(0.63,0.8,'mean annual='+str(mean_annual))
plt.hist(final, 50, normed=True)
plt.show()

```

The corresponding graph is shown as follows:



Simulation of stock price movements

We mentioned in the previous sections that in finance, returns are assumed to follow a normal distribution, whereas prices follow a lognormal distribution. The stock price at time $t+1$ is a function of the stock price at t , mean, standard deviation, and the time interval as shown in the following formula:

$$S_{t+1} = S_t + \hat{\mu}S_t\Delta t + \sigma S_t \epsilon \sqrt{\Delta t} \quad (7)$$

In this formula, S_{t+1} is the stock price at $t+1$, $\hat{\mu}$ is the expected stock return, Δt is the time interval ($\Delta t = T/n$), T is the time (in years), n is the number of steps, ϵ is the distribution term with a zero mean, and σ is the volatility of the underlying stock. With a simple manipulation, equation (4) can lead to the following equation that we will use in our programs:

$$S_{t+1} = S_t \exp\left(\left(\hat{\mu} - \frac{1}{2}\sigma^2\right)\Delta t + \sigma \epsilon \sqrt{\Delta t}\right) \quad (8)$$

In a risk-neutral work, no investors require compensation for bearing risk. In other words, in such a world, the expected return on any security (investment) is the risk-free rate. Thus, in a risk-neutral world, the previous equation becomes the following equation:

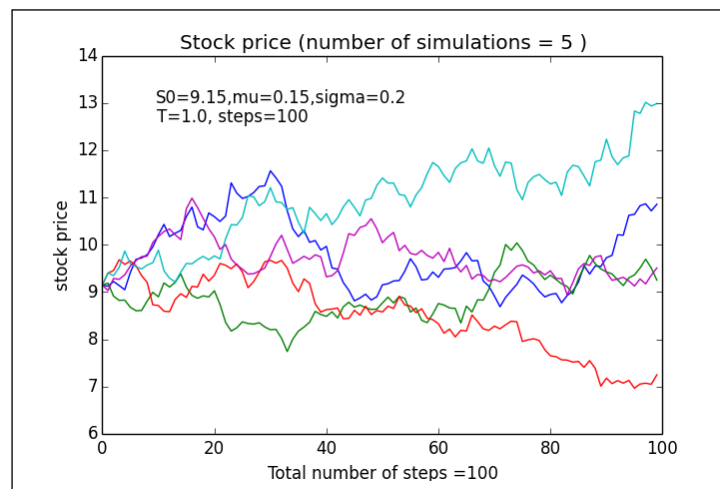
$$S_{t+1} = S_t \exp\left(\left(r - \frac{1}{2}\sigma^2\right)\Delta t + \sigma \epsilon \sqrt{\Delta t}\right) \quad (9)$$

If you want to learn more about the risk-neutral probability, refer to *Options, Futures and Other Derivatives*, 7th edition, John Hull, Pearson, 2009. The Python code to simulate a stock's movement (path) is as follows:

```
import scipy as sp
stock_price_today = 9.15 # stock price at time zero
T = 1. # maturity date (in years)
n_steps = 100. # number of steps
mu = 0.15 # expected annual return
sigma = 0.2 # volatility (annualized)
sp.random.seed(12345) # seed()
n_simulation = 5 # number of simulations
dt = T/n_steps
S = sp.zeros([n_steps], dtype=float)
x = range(0, int(n_steps), 1)
for j in range(0, n_simulation):
    S[0] = stock_price_today
    for i in x[:-1]:
        e = sp.random.normal()
        S[i+1] = S[i] + S[i] * (mu - 0.5*pow(sigma,2)) * dt + sigma * S[i] * sp.
sqrt(dt) * e;
    plot(x, S)
```

```
figtext(0.2,0.8,'S0='+str(S0)+' ,mu='+str(mu)+' ,sigma='+str(sigma))
figtext(0.2,0.76,'T='+str(T)+' , steps='+str(int(n_steps)))
title('Stock price (number of simulations = %d ' % n_simulation +')')
xlabel('Total number of steps =' +str(int(n_steps)))
ylabel('stock price')
show()
```

To make our graph more readable, we deliberately choose just five simulations. Since the `seed()` function is applied, you can replicate the following graph by running the previous code:



Graphical presentation of stock prices at options' maturity dates

Up to now, we have discussed that options are really path-independent, which means the option prices depend on terminal values. Thus, before pricing such an option, we need to know the terminal stock prices. To extend the previous program, we have the following code to estimate the terminal stock prices for a given set of values: `S0` (initial stock price), `n_simulation` (number of terminal prices), `T` (maturity date in years), `n_steps` (number of steps), `mu` (expected annual stock returns), and `sigma` (volatility):

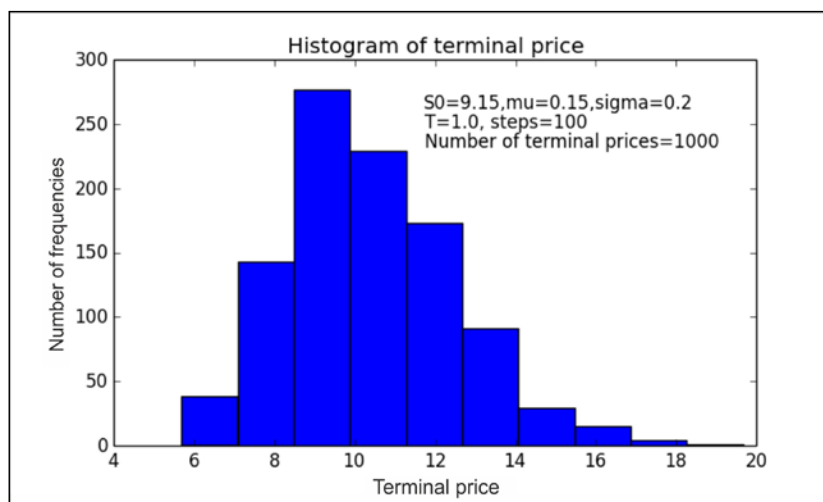
```
from scipy import zeros, sqrt, shape
import scipy as sp
S0 = 9.15          # stock price at time zero
T = 1.             # years
```

```

n_steps=100.          # number of steps
mu =0.15              # expected annual return
sigma = 0.2           # volatility (annual)
sp.random.seed(12345) # fix those random numbers
n_simulation = 1000    # number of simulation
dt =T/n_steps
S = zeros([n_simulation], dtype=float)
x = range(0, int(n_steps), 1)
for j in range(0, n_simulation):
    tt=S0
    for i in x[:-1]:
        e=sp.random.normal()
        tt+=tt*(mu-0.5*pow(sigma,2))*dt+sigma*tt*sqrt(dt)*e;
    S[j]=tt
title('Histogram of terminal price')
ylabel('Number of frequencies')
xlabel('Terminal price')
figtext(0.5,0.8,'S0='+str(S0)+' ,mu='+str(mu)+' ,sigma='+str(sigma))
figtext(0.5,0.76,'T='+str(T)+' , steps='+str(int(n_steps)))
figtext(0.5,0.72,'Number of terminal prices='+str(int(n_simulation)))
hist(S)

```

The histogram of our simulated terminal prices is shown as follows:



Finding an efficient portfolio and frontier

In this section, we show you how to use the Monte Carlo simulation to generate returns for a pair of stocks with known means, standard deviations, and correlation between them. By applying the maximize function, we minimize the portfolio risk of this two-stock portfolio. Then, we change the correlations between the two stocks to illustrate the impact of correlation on our efficient frontier. The last one is the most complex one since it constructs an efficient frontier based on n stocks.

Finding an efficient frontier based on two stocks

The following program aims at generating an efficient frontier based on two stocks with known means, standard deviations, and correlation. We have just six input values: two means, two standard deviations, the correlation (ρ), and the number of simulations. To generate the correlated $y1$ and $y2$ time series, we generate the uncorrelated $x1$ and $x2$ series first. Then, we apply the following formulae:

$$y_1 = x_1 \quad (10A)$$

$$y_2 = \rho x_1 + \sqrt{1 - \rho^2} x_2 \quad (10B)$$

Another important issue is how to construct an objective function to minimize. Our objective function is the standard deviation of the portfolio in addition to a penalty that is defined as the scaled absolute deviation from our target portfolio mean. In other words, we minimize both the risk of the portfolio and the deviation of our portfolio return from our target return as shown in the following code:

```
import numpy as np
import scipy as sp
import pandas as pd
from datetime import datetime as dt
from scipy.optimize import minimize
# Step 1: input area
mean_0=(0.15,0.25) # mean returns for 2 stocks
std_0= (0.10,0.20) # standard deviations for 2 stocks
corr_=0.2          # correlation between 2 stocks
```

```

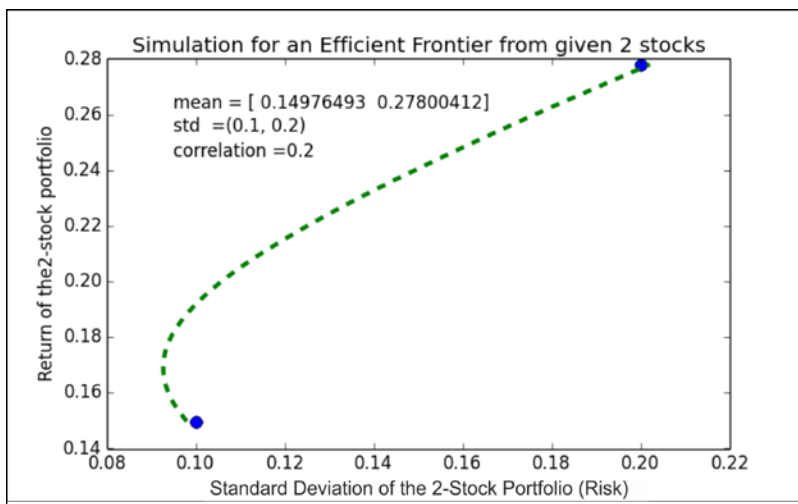
n=1000                # number of simulations (returns) for each stock
# Step 2: Generate two uncorrelated time series
n_stock=len(mean_0)
sp.random.seed(12345) # could generate the same random numbers
x1=sp.random.normal(loc=mean_0[0],scale=std_0[0],size=n)
x2=sp.random.normal(loc=mean_0[1],scale=std_0[1],size=n)
if(any(x1)<=-1.0 or any(x2)<=-1.0):
    print ('Error: return is <=-100%')
# Step 3: Generate two correlated time series
index=pd.date_range(start=dt(2001,1,1),periods=n,freq='d')
y1=pd.DataFrame(x1,index=index_)
y2=pd.DataFrame(corr_*x1+sqrt(1-corr_**2)*x2,index=index_)
# step 4: generate a return matrix called R
R0=pd.merge(y1,y2,left_index=True,right_index=True)
R=np.array(R0)
# Step 5: define a few functions
def objFunction(W, R, target_ret):
    stock_mean=np.mean(R,axis=0)
    port_mean=np.dot(W,stock_mean)          # portfolio mean
    cov=np.cov(R.T)                         # var-covar matrix
    port_var=np.dot(np.dot(W,cov),W.T)      # portfolio variance
    penalty = 2000*abs(port_mean-target_ret) # penalty 4 deviation
    return np.sqrt(port_var) + penalty       # objective function
# Step 6: estimate optimal portfolio for a given return
out_mean,out_std,out_weight=[],[],[]
stockMean=np.mean(R,axis=0)
for r in np.linspace(np.min(stockMean), np.max(stockMean), num=100):
    W = ones([n_stock])/n_stock             # start equal w
    b_ = [(0,1) for i in range(n_stock)]    # bounds
    c_ = ({'type':'eq', 'fun': lambda W: sum(W)-1. })# constraint
    result=minimize(objFunction,W,(R,r),method='SLSQP',constraints=c_,
    bounds=b_)
    if not result.success:                   # handle error
raise BaseException(result.message)
    out_mean.append(round(r,4))               # a few decimal places

```



```
std_=round(np.std(np.sum(R*result.x,axis=1)),6)
out_std.append(std_)
out_weight.append(result.x)
# Step 7: plot the efficient frontier
title('Simulation for an Efficient Frontier from given 2 stocks')
xlabel('Standard Deviation of the 2-stock Portfolio (Risk)')
ylabel('Return of the 2-stock portfolio')
figtext(0.2,0.80,' mean = '+str(stockMean))
figtext(0.2,0.75,' std  =' +str(std_0))
figtext(0.2,0.70,' correlation ='+str(corr_))
plot(np.array(std_0),np.array(stockMean),'o',markersize=8)
plot(out_std,out_mean,'--',linewidth=3)
```

The corresponding graph is shown as follows:



Impact of different correlations

Based on the previous program, we vary the correlations between the two stocks to illustrate the critical role played by this factor in terms of diversification, as shown in the following code:

```
import numpy as np
import scipy as sp
import pandas as pd
```

```

from datetime import datetime as dt
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# Step 1: input area
mean_0=(0.15,0.25) # mean returns for 2 stocks
std_0= (0.10,0.20) # standard deviations for 2 stocks
n=1000             # number of simulations (returns) for each stock
corr_=(0.1,0.5,0.8)
# Step 2: Generate two uncorrelated time series
n_stock=len(mean_0)
sp.random.seed(12345) # could generate the same random numbers
x11=sp.random.normal(loc=0,scale=1,size=n)
x12=sp.random.normal(loc=0,scale=1,size=n)
n_corr=len(corr_)
style_=['-.-','--','-']

for j in range(n_corr):
    # Step 3: Generate two correlated time series
    corr2=corr_[j]
    index_=pd.date_range(start=dt(2001,1,1),periods=n,freq='d')
    x21=pd.DataFrame(x11,index=index_)
    x22=pd.DataFrame(corr2*x11+sqrt(1-corr2**2)*x12,index=index_)
    y1=mean_0[0]+x21*std_0[0]
    y2=mean_0[1]+x22*std_0[1]
    # step 4: generate a return matrix called R
    R0=pd.merge(y1,y2,left_index=True,right_index=True)
    R=np.array(R0)
    # Step 5: define a few functions
def objFunction(W, R, target_ret):
    stock_mean=np.mean(R,axis=0)
    port_mean=np.dot(W,stock_mean)           # portfolio mean
    cov=np.cov(R.T)                          # var-covar matrix
    port_var=np.dot(np.dot(W,cov),W.T)       # portfolio variance
    penalty = 2000*abs(port_mean-target_ret) # penalty 4 deviation

```

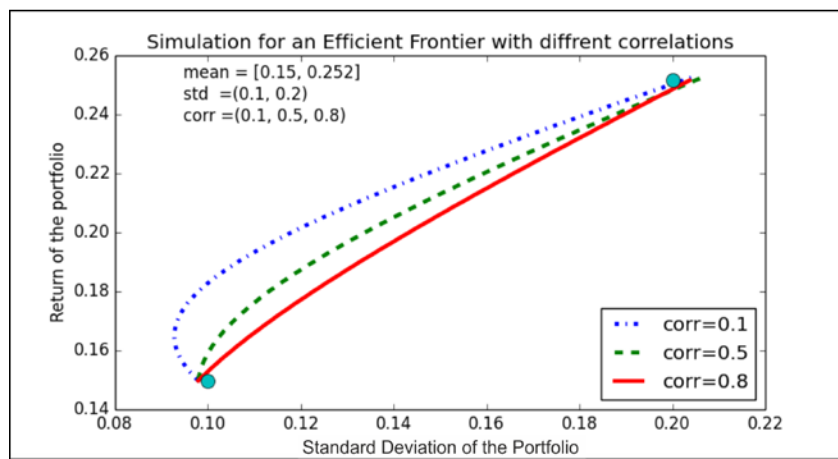
```
return np.sqrt(port_var) + penalty      # objective function
    #print('stock mean=',stockMean)

    # Step 6: estimate optimal portfolio for a given return
    out_mean,out_std,out_weight=[],[],[]
    stockMean=np.mean(R,axis=0)
    print('hahastyle[j]',stockMean)
    for r in np.linspace(np.min(stockMean), np.max(stockMean), num=100):
        W = ones([n_stock])/n_stock    # starting:equal w
        b_ = [(0,1) for i in range(n_stock)]    # bounds
        c_ = ({'type':'eq', 'fun': lambda W: sum(W)-1. })# constraint
        result=minimize(objFunction,W, (R,r),method='SLSQP',constraints
=c_, bounds=b_)
        if not result.success:
raise BaseException(result.message)
        out_mean.append(round(r,4))            # a few decimal places
        std_=round(np.std(np.sum(R*result.x,axis=1)),6)
        out_std.append(std_)
        out_weight.append(result.x)

    # Step 7A: plot the efficient frontier
    plt.plot(out_std,out_mean,style_[j],label='corr='+str(corr2),linewidth
th=3)

# Step 7B: plot the efficient frontier
stockMean2=[round(stockMean[0],3),round(stockMean[1],3)]
title('Simulation for an Efficient Frontier with different correlations')
xlabel('Standard Deviation of the Porfolio')
ylabel('Return of the portfolio')
figtext(0.2,0.85,' mean = '+str(stockMean2))
figtext(0.2,0.80,' std  ='+str(std_0))
figtext(0.2,0.75,' corr ='+str(corr_))
plt.plot(np.array(std_0),np.array(stockMean),'o',markersize=10)
plt.legend(loc='lower right')
plt.show()
```

The following graph suggests that the lower the correlation, the better our two-stock formed efficient frontier:



Constructing an efficient frontier with n stocks

When the number of stocks, n , increases, the correlation between each pair of stocks increases dramatically. For n stocks, we have $n*(n-1)/2$ correlations. For example, if n is 10, we have 45 correlations. Because of this, it is not a good idea to manually input those values. Instead, we generate means, standard deviations, and correlations by drawing random numbers from several uniform distributions. To produce correlated returns, first we generate n uncorrelated stock return time series and then apply Cholesky decomposition as follows:

```
import numpy as np
import scipy as sp
import pandas as pd
from datetime import datetime as dt
from scipy.optimize import minimize

# Step 1: input area
n_stocks=10

sp.random.seed(123456)                                # produce the same random
numbers                                                  # number of correlation
n_corr=n_stocks*(n_stocks-1)/2
```

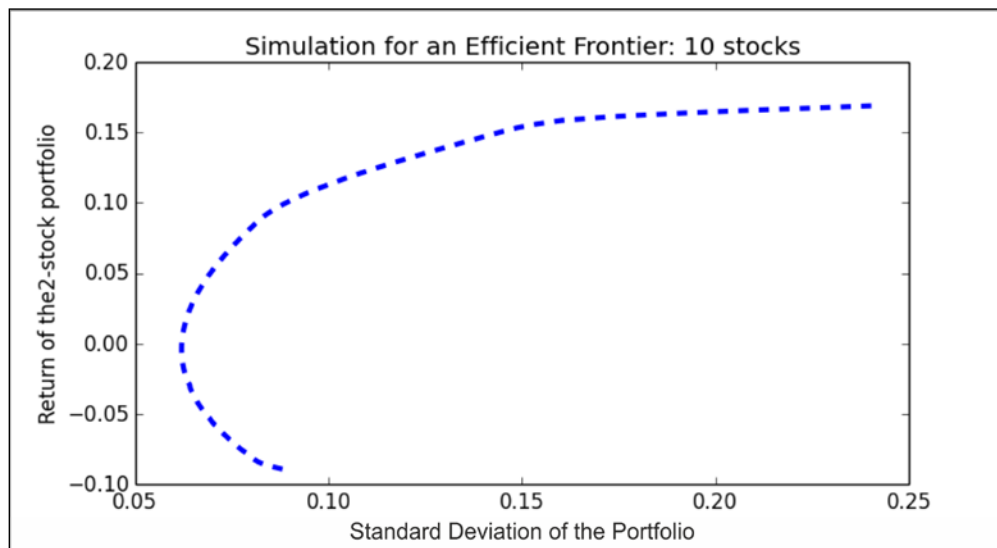
```
corr_0=sp.random.uniform(0.05,0.25,n_corr) # generate correlations
mean_0=sp.random.uniform(-0.1,0.25,n_stocks)# means
std_0=sp.random.uniform(0.05,0.35,n_stocks) # standard deviation
n_obs=1000          # number of simulations (returns) for each stock
# Step 2: produce correlation matrix: Cholesky decomposition
corr_=sp.zeros((n_stocks,n_stocks))
for i in range(n_stocks):
    for j in range(n_stocks):
        if i==j:
            corr_[i,j]=1
        else:
            corr_[i,j]=corr_0[i+j]
U=np.linalg.cholesky(corr_)
# Step 3: Generate two uncorrelated time series
R0=np.zeros((n_obs,n_stocks))
for i in range(n_obs):
    for j in range(n_stocks):
        R0[i,j]=sp.random.normal(loc=mean_0[j],scale=std_0[j],size=1)
if(any(R0)<=-1.0):
    print ('Error: return is <=-100%')
# Step 4: generate correlated return matrix: Cholesky
R=np.dot(R0,U)
R=np.array(R)
# Step 5: define a few functions
def objFunction(W, R, target_ret):
    stock_mean=np.mean(R,axis=0)
    port_mean=np.dot(W,stock_mean)          # portfolio mean
    cov=np.cov(R.T)                          # var-covar matrix
    port_var=np.dot(np.dot(W,cov),W.T)       # portfolio variance
    penalty = 2000*abs(port_mean-target_ret) # penalty 4 deviation
    return np.sqrt(port_var) + penalty       # objective function
# Step 6: estimate optimal portfolio for a given return
out_mean,out_std,out_weight=[],[],[]
stockMean=np.mean(R,axis=0)
```

```

for r in np.linspace(np.min(stockMean), np.max(stockMean), num=100):
    W = sp.ones([n_stocks])/n_stocks          # starting:equal w
    b_ = [(0,1) for i in range(n_stocks)]      # bounds
    c_ = ({'type':'eq', 'fun': lambda W: sum(W)-1. })# constraint
    result=minimize(objFunction,W,(R,r),method='SLSQP',constraints=c_,
bounds=b_)
    if not result.success:                      # handle error
raise BaseException(result.message)
    out_mean.append(round(r,4))                  # a few decimal places
    std_=round(np.std(np.sum(R*result.x,axis=1)),6)
    out_std.append(std_)
    out_weight.append(result.x)
# Step 7: plot the efficient frontier
title('Simulation for an Efficient Frontier: '+str(n_stocks)+' stocks')
xlabel('Standard Deviation of the Porfolio')
ylabel('Return of the2-stock portfolio')
#xlim(min(std_0), max(std_0))
plot(out_std,out_mean,'--',linewidth=3)

```

The related graph is as follows:



Geometric versus arithmetic mean

In the next section, we discuss long-term return forecasting. Since we apply the weighted arithmetic and geometric means, we need to familiarize ourselves with the geometric mean first. For n returns $(R_1, R_2, R_3, \dots, R_n)$ their arithmetic and geometric means are defined as follows:

$$\bar{R}_{arithmetic} = \frac{\sum_{i=1}^n R_i}{n} \quad (10)$$

$$\bar{R}_{geometric} = \left[\prod_{i=1}^n (1 + R_i) \right]^{\frac{1}{n}} - 1 \quad (11)$$

In this formula, R_i is the stock's i th return. For an arithmetic mean, we could use the `mean()` function. Most of the time, the arithmetic mean is used in our estimations because of its simplicity. Since geometric means consider the time of values, it is considered to be more accurate for returns' estimation based on historical data. One important feature is that the geometric mean is smaller than its corresponding arithmetic mean unless all input values, such as all returns, are all the same. Because of this feature, many argue that using arithmetic means to predict future returns would lead to an overestimation. In contrast, geometric means would lead to an underestimation. Since the geometric mean for returns is different from the normal definition of the geometric mean when the values are not returns, it is worthwhile to write our own function as shown in the following code:

```
def geomean_ret(returns):
    product = 1
    for ret in returns:
        product *= (1+ret)
    return product ** (1.0/len(returns)) - 1
```

For a set of n returns, we could estimate their arithmetic and geometric mean as follows:

```
>>>returns=[0.01,0.02,-0.03,0.015,0.10]
>>>geomean_ret(returns)
0.022140040774623948
>>>mean(returns)
0.023
```

Long-term return forecasting

Many researchers and practitioners argue that a long-term return forecast would be overestimated if it is based on the arithmetic mean of the past returns and underestimated based on a geometric mean. Using 80 years' historical returns to forecast the next 25-year future return, Jacquier, Kane, and Marcus (2003) suggest the following weighted scheme:

$$\text{long-term forecast} = \frac{25}{80} R_{\text{geometric}} + \frac{80-25}{80} R_{\text{arithmetic}} \quad (12)$$

The following program reflects equation (12):

```
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd

ticker='IBM'           # input value 1
begdate=(1926,1,1)     # input value 2
enddate=(2013,12,31)   # input value 3
n_forecast=15.         # input value 4

def geomean_ret(returns):
    product = 1
    for ret in returns:
        product *= (1+ret)
    return product ** (1.0/len(returns)) -1

x=quotes_historical_yahoo(ticker,begdate,enddate,asobject=True,
adjusted=True)
logret = log(x.aclose[1:]/x.aclose[:-1])
date=[]
d0=x.date
for i in range(0,size(logret)):
    date.append(d0[i].strftime("%Y"))
y=pd.DataFrame(logret,date,columns=['logret'],dtype=float64)
ret_annual=exp(y.groupby(y.index).sum())-1
ret_annual.columns=['ret_annual']
```



```
n_history=len(ret_annual)
a_mean=mean(np.array(ret_annual))
g_mean=geomean_ret(np.array(ret_annual))
future_ret=n_forecast/n_history*g_mean+(n_history-n_forecast)/n_
history*a_mean
print 'Arithmetic mean=',round(a_mean,3), 'Geomean=',round(g_
mean,3), 'forecast=',future_ret
```

We could print a few of the annual returns, the number of years, and final result as follows:

```
>>>ret_annual.head()
      ret_annual
1962    -0.326613
1963     0.347305
1964    -0.022222
1965     0.222727
1966     0.122677
>>>len(ret_annual)
52
>>>print 'Arithmetic omean=',round(a_mean,3), 'Geomean=',round(g_
mean,3), 'forecast=',future_ret
Arithmetic omean= 0.12 Geomean= 0.087 forecast= [ 0.11074861]
>>>
```

Pricing a call using simulation

After knowing the terminal prices, we could estimate the payoff for a call if the exercise price is given. The mean of those discounted payoffs using the risk-free rate as our discount rate will be our call price. The following code helps us estimate the call price:

```
from scipy import zeros, sqrt, shape
import scipy as sp
S0 = 40.      # stock price at time zero
X=  40.      # exercise price
T =0.5       # years
r =0.05      # risk-free rate
sigma = 0.2  # volatility (annual)
```

```

n_steps=100.          # number of steps
sp.random.seed(12345) # fix those random numbers
n_simulation = 5000    # number of simulation
dt =T/n_steps
call = zeros([n_simulation], dtype=float)
x = range(0, int(n_steps), 1)
for j in range(0, n_simulation):
    sT=S0
    for i in x[:-1]:
        e=sp.random.normal()
        sT*=exp((r-0.5*sigma*sigma)*dt+sigma*e*sqrt(dt))
        call[j]=max(sT-X,0)
call_price=mean(call)*exp(-r*T)
print 'call price = ', round(call_price,3)

```

The estimated call price is \$2.75. The same logic applies to pricing a put option.

Exotic options

Up to now, we discussed European and American options in *Chapter 9, The Black-Scholes-Merton Option Model*, which are also called **vanilla** options. One of the characters is path independent. On the other hand, exotic options are more complex since they might have several triggers relating to the determination of their payoffs. An exotic option could include nonstandard underlying instrument developed for particular investors, banks, or firms. Exotic options usually are traded **over-the-counter (OTC)**. For exotic options, we don't have closed-form solutions, such as the Black-Scholes-Merton model. Thus, we have to depend on other means to price them. The Monte Carlo simulation is one of the ways to price many exotic options. In the next several subsections, we show how to price Asian options, digit options, and barrier options.

Using the Monte Carlo simulation to price average options

European and American options are path-independent options. This means that an option's payoff depends only on the terminal stock price and strike price. One related issue for path-dependent options is market manipulation at the maturity date. Another issue is that some investors or hedgers might care more about the average price instead of a terminal price.

For example, a refinery is worried about the oil, its major raw material, and price movement in the next three months. They plan to hedge the potential price jumps in crude oil. The company could buy a call option. However, since the firm consumes a huge amount of crude oil every day, naturally it cares more about the average price instead of just the terminal price on which a vanilla call option depends. For such cases, average options will be more effective. Average options are a type of Asian options. For an average option, its payoff is determined by the average underlying prices over some preset period of time. There are two types of averages: arithmetic average and geometric average.

The payoff function of an Asian call (average price) is given as follows:

$$\text{payoff}(\text{call}) = \text{Max}(P_{\text{average}} - X, 0) \quad (13)$$

The payoff function of an Asian put (average price) is given below.

$$\text{payoff}(\text{put}) = \text{Max}(X - P_{\text{average}}, 0) \quad (14)$$

Asian options are one of the basic forms of exotic options. Another advantage of Asian options is that their costs are cheaper compared to European and American vanilla options since the variation of an average will be much smaller than a terminal price. The following Python program is for an Asian option with an arithmetic average price:

```
import scipy as sp
s0=40.          # today stock price
x=40.           # exercise price
T=0.5           # maturity in years
r=0.05          # risk-free rate
sigma=0.2       # volatility (annualized)
n_simulation=100 # number of simulations
n_steps=100.
dt=T/n_steps
call=sp.zeros([n_simulation], dtype=float)
for j in range(0, n_simulation):
    sT=s0
    total=0
    for i in range(0,int(n_steps)):
        e=sp.random.normal()
```

```

sT*=sp.exp((r-0.5*sigma*sigma)*dt+sigma*e*sp.sqrt(dt))
total+=sT
price_average=total/n_steps
call[j]=max(price_average-x,0)
call_price=mean(call)*exp(-r*T)
print 'call price = ', round(call_price,3)

```

Pricing barrier options using the Monte Carlo simulation

Unlike the Black-Scholes-Merton option model's call and put options, which are path independent, a barrier option is path-dependent. A barrier option is similar in many ways to an ordinary option except there exists a trigger. An "in" option starts its life worthless unless the underlying stock reaches a predetermined knock-in barrier. On the contrary, an "out" barrier option starts its life active and turns useless when a knock-out barrier price is breached. In addition, if a barrier option expires inactive, it may be worthless, or there may be a cash rebate paid out as a fraction of the premium. The four types of barrier options are given as follows:

- **Up-and-out:** In this barrier option, the price starts from below a barrier level. If it reaches the barrier, it is knocked out.
- **Down-and-out:** In this barrier option, the price starts from above a barrier. If it reaches the barrier, it is knocked out.
- **Up-and-in:** In this barrier option, the price starts below a barrier and has to reach the barrier to be activated.
- **Down-and-in:** In this barrier option, the price starts above a barrier and has to reach the barrier to be activated.

The next Python program is for an up-and-out barrier option with a European call:

```

import scipy as sp
import p4f
def up_and_out_call(s0,x,T,r,sigma,n_simulation,barrier):
    n_steps=100.
    dt=T/n_steps
    total=0
    for j in range(0, n_simulation):
        sT=s0
        out=False

```

```
    for i in range(0,int(n_steps)):
        e=sp.random.normal()
        sT*=sp.exp((r-0.5*sigma*sigma)*dt+sigma*e*sp.sqrt(dt))
        if sT>barrier:
            out=True
    if out==False:
        total+=p4f.bs_call(s0,x,T,r,sigma)
    return total/n_simulation
```

The basic design is that we simulate the stock movement n times, such as 100 times. For each simulation, we have 100 steps. Whenever the stock price reaches the barrier, the payoff will be zero. Otherwise, the payoff will be a vanilla European call. The final value will be the summation of all call prices that are not knocked out, divided by the number of simulations, as shown in the following code:

```
s0=40.          # today stock price
x=40.           # exercise price
barrier=42      # barrier level
T=0.5           # maturity in years
r=0.05          # risk-free rate
sigma=0.2       # volatility (annualized)
n_simulation=100 # number of simulations
result=up_and_out_call(s0,x,T,r,sigma,n_simulation,barrier)
print 'up-and-out-call = ', round(result,3)
up-and-out-call = 0.606
```

The Python code for the down-and-in put option is shown as follows:

```
def down_and_in_put(s0,x,T,r,sigma,n_simulation,barrier):
    n_steps=100.
    dt=T/n_steps
    total=0
    for j in range(0, n_simulation):
        sT=s0
        in_=False
        for i in range(0,int(n_steps)):
            e=sp.random.normal()
            sT*=sp.exp((r-0.5*sigma*sigma)*dt+sigma*e*sp.sqrt(dt))
            if sT<barrier:
```

```

        in_=True
        #print 'sT=',sT
    #print 'j=',j , 'out=',out
    if in_==True:
        total+=p4f.bs_put(s0,x,T,r,sigma)
    return total/n_simulation

```

Barrier in-and-out parity

If we buy an up-and-out European call and an up-and-in European call, then the following parity should hold good:

$$call_{up-and-out} + call_{up-and-in} = call \quad (14)$$

The logic is very simple—if the stock price reaches the barrier, then the first call is worthless and the second call will be activated. If the stock price never touches the barrier, the first call will remain active, while the second one is never activated. Either way, one of them is active. The following Python program illustrates such scenarios:

```

def up_call(s0,x,T,r,sigma,n_simulation,barrier):
    import scipy as sp
    import p4f
    n_steps=100.
    dt=T/n_steps
    inTotal=0
    outTotal=0
    for j in range(0, n_simulation):
        sT=s0
        inStatus=False
        outStatus=True
    for i in range(0,int(n_steps)):
        e=sp.random.normal()
        sT*=sp.exp((r-0.5*sigma*sigma)*dt+sigma*e*sp.sqrt(dt))
    if sT>barrier:
        outStatus=False
        inStatus=True

```

```
        #print 'sT=',sT
    #print 'j=',j , 'out=',out
    if outStatus==True:
        outTotal+=p4f.bs_call(s0,x,T,r,sigma)
    else:
        inTotal+=p4f.bs_call(s0,x,T,r,sigma)
    return outTotal/n_simulation, inTotal/n_simulation
```

We input a set of values to test whether the summation of an up-and-out call and an up-and-in call will be the same as a vanilla call:

```
s0=40.          # today stock price
x=40.           # exercise price
barrier=42      # barrier level
T=0.5           # maturity in years
r=0.05          # risk-free rate
sigma=0.2       # volatility (annualized)
n_simulation=100 # number of simulations
upOutCall,upInCall=up_call(s0,x,T,r,sigma,n_simulation,barrier)
print 'upOutCall=', round(upOutCall,2), 'upInCall=',round(upInCall,2)
print 'Black-Scholes call', round(p4f.bs_call(s0,x,T,r,sigma),2)
```

The following output proves the parity mentioned in the preceding paragraph:

```
upCall= 0.8 upInCall= 1.96
Black-Scholes call 2.76
```

Graphical presentation of an up-and-out and up-and-in parity

It is a good idea to use the Monte Carlo simulation to present such a parity. The following code is designed to achieve this. To make our simulation clearer, we deliberately choose just five simulations:

```
import scipy as sp
s0=9.15          # stock price at time zero
x=9.15           # exercise price
barrier=10.15    # barrier
T =0.5           # maturity date (in years)
```

```

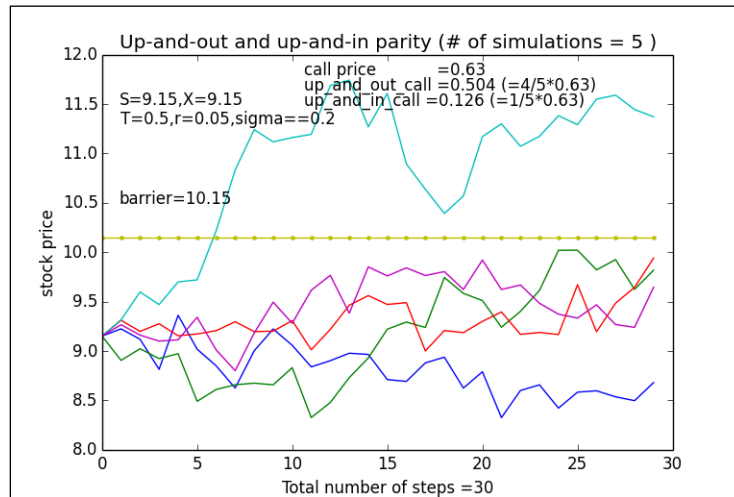
n_steps=30.          # number of steps
r =0.05              # expected annual return
sigma = 0.2          # volatility (annualized)
sp.random.seed(125)  # seed()
n_simulation = 5      # number of simulations
dt =T/n_steps
S = sp.zeros([n_steps], dtype=float)
time_ = range(0, int(n_steps), 1)
c=p4f.bs_call(s0,x,T,r,sigma)
sp.random.seed(124)
outTotal, inTotal= 0.,0.
n_out,n_in=0,0
for j in range(0, n_simulation):
    S[0]= s0
    inStatus=False
    outStatus=True
    for i in time_[:-1]:
        e=sp.random.normal()
        S[i+1]=S[i]*exp((r-0.5*pow(sigma,2))*dt+sigma*sp.sqrt(dt)*e)
        if S[i+1]>barrier:
            outStatus=False
            inStatus=True
        plot(time_, S)
    if outStatus==True:
        outTotal+=c;n_out+=1
    else:
        inTotal+=c;n_in+=1
S=sp.zeros(int(n_steps))+barrier
plot(time_,S,'.-')
upOutCall=round(outTotal/n_simulation,3)
upInCall=round(inTotal/n_simulation,3)
figtext(0.15,0.8, 'S='+str(S0)+' ,X='+str(x))
figtext(0.15,0.76, 'T='+str(T)+' ,r='+str(r)+' ,sigma=='+str(sigma))
figtext(0.15,0.6, 'barrier='+str(barrier))
figtext(0.40,0.86, 'call price          =' +str(round(c,3)))

```



```
figtext(0.40,0.83,'up_and_out_call =' +str(upOutCall)+' (' +str(n_
out)+'/' +
str(n_simulation)+'*' +str(round(c,3))+'')')
figtext(0.40,0.80,'up_and_in_call =' +str(upInCall)+' (' +str(n_in)+'/' +
str(n_simulation)+'*' +str(round(c,3))+'')')
title('Up-and-out and up-and-in parity (# of simulations = %d ' % n_
simulation +')')
xlabel('Total number of steps =' +str(int(n_steps)))
ylabel('stock price')
show()
```

The price of a vanilla call is \$0.65. Since there is one simulation that reached the barrier, the up-and-out call will be $4/5 \times 0.63$, while the up-and-in call will be $1/5 \times 0.63$. The corresponding graph is shown as follows:



Pricing lookback options with floating strikes

The lookback options depend on the paths (history) travelled by the underlying security. Thus, they are called path-dependent exotic options as well. One of them is named floating strikes. The payoff function of a call when the exercise price is the minimum price achieved during the life of the option is given as follows:

$$\text{Payoff} = \text{Max}(S_T - S_{\min}, 0) = S_T - S_{\min} \quad (15)$$

The Python code for this lookback option is shown as follows:

```
def lookback_min_price_as_strike(s,T,r,sigma,n_simulation):
    n_steps=100.
    dt=T/n_steps
    total=0
    for j in range(n_simulation):
        min_price=100000.    # a very big number
        sT=s
        for i in range(int(n_steps)):
            e=sp.random.normal()
            sT*=sp.exp((r-0.5*sigma*sigma)*dt+sigma*e*sp.sqrt(dt))
        if sT<min_price:
            min_price=sT
            #print 'j=',j,'i=',i,'total=',total
            total+=p4f.bs_call(s,min_price,T,r,sigma)
    return total/n_simulation
```

Remember that the previous function needs two modules. Thus, we have to import those modules before we call the function as shown in the following code:

```
>>>import scipy as sp
>>>import p4f
>>>s=40.          # today stock price
>>>T=0.5          # maturity in years
>>>r=0.05         # risk-free rate
>>>sigma=0.2      # volatility (annualized)
>>>n_simulation=1000 # number of simulations
>>>result=lookback_min_price_as_strike(s,T,r,sigma,n_simulation)
>>>print 'lookback min price as strike = ', round(result,3)
```

The result for one run is shown as follows:

```
lookback min price as strike =  5.304
```

Using the Sobol sequence to improve the efficiency

When applying the Monte Carlo simulation to solve various finance related problems, we need to generate a certain number of random numbers. When the accuracy is very high, we have to draw a huge amount of such random numbers. For example, when pricing options, we use very small interval or a large number of steps to increase the number of decimal places of our final option prices. Thus, the efficiency of our Monte Carlo simulation would be a vital issue in terms of computational time and costs. This is especially true if we have a thousand options to price. One way to increase the efficiency is to apply a correct or better algorithm, that is, optimize our code. Another way is to use some special types of random number generators, such as the Sobol sequence.

Sobol sequences belong to the so-called low-discrepancy sequences, which satisfy the properties of random numbers but are distributed more evenly. Thus, they are usually called quasi random. Based on the related Python Sobol library developed, we could have the programs from the given links. First, we go to the web page at http://people.sc.fsu.edu/~jburkardt/py_src/sobol/sobol.html and download a Python program called `sobol_lib.py` written by Corrado Chisari. Another web page related to the Sobol sequence is https://github.com/naught101/sobol_seq.

Summary

In this chapter, we discussed several types of distributions: normal, standard normal, lognormal, and Poisson. Since the assumption that stocks follow a lognormal distribution and returns follow a normal distribution is the cornerstone for option theory, the Monte Carlo simulation is used to price European options. Under certain scenarios, Asian options might be more effective in terms of hedging. Exotic options are more complex than the vanilla options since the former have no closed-form solution, while the latter could be priced by the Black-Scholes-Merton option model. One way to price these exotic options is to use the Monte Carlo simulation. The Python programs to price an Asian option and lookback options are discussed in detail.

In the next chapter, we will discuss various volatility measures, such as our conventional standard deviation, **Lower Partial Standard Deviation (LPSD)**. Using the standard deviation of returns as a risk measure is based on a critical assumption that stock returns follow a normal distribution. Because of this, we introduce several normality tests. In addition, we graphically show volatility clustering—high volatility is usually followed by a high-volatility period, while low volatility is usually followed by a low-volatility period. To deal with this phenomenon, the **Autoregressive conditional heteroskedasticity (ARCH)** process was developed by Angel (1982), and the **Generalized AutoRegressive Conditional Heteroskedasticity (GARCH)** processes, which are an extension of ARCH was developed by Bollerslev (1986). Their graphical presentations and related Python programs will be also covered in the next chapter.

Exercises

1. Download daily price from Yahoo! Finance for DELL. Estimate daily returns and convert them into monthly returns. Assume its monthly returns follow a normal distribution. Draw a graph with the mean and standard deviation from the previous monthly returns.

2. Debug the following program:

```
import scipy as sp
S0 = 9.15 ;T=1;n_steps=10;mu =0.15;sigma = 0.2
n_simulation = 10
dt =T/n_steps
S = sp.zeros([n_steps], dtype=float)
x = range(0, int(n_steps), 1)
for j in range(0, n_simulation):
    S[0]= S0
    for i in x[:-1]:
        e=sp.random.normal()
        S[i+1]=S[i,j]+S[i]*(mu-0.5*pow(sigma,2))*dt+sigma*S[i]*sp.
sqrt(dt)*e;
    plot(x, S)
figtext(0.2,0.8,'S0='+str(S0)+' ,mu='+str(mu)+' ,sigma='+str(sigma))
figtext(0.2,0.76,'T='+str(T)+' , steps='+str(int(n_steps)))
```

```
title('Stock price (number of simulations = %d ' % n_simulation +'))  
xlabel('Total number of steps =' + str(n_steps))  
ylabel('stock price')  
show()
```

3. Write a Python program to price an Asian average price put based on the arithmetic mean.
4. Write a Python program to price an Asian average price put based on the geometric mean.
5. Write a Python program to price an up-and-in call (barrier option).
6. Write a Python program to price a down-and-out put (barrier option).
7. Write a Python program to show the down-and-out and down-and-in parity.
8. Write a Python program to use `permutation()` from `SciPy` to select 12 monthly returns randomly from the past five-year data without placement. To test your program, you can use Citigroup and the time period January 1, 2009 to December 31, 2014 from Yahoo! Finance.
9. Write a Python program to run bootstrapping with n given returns. For each time, we select m returns where $m > n$.

12

Volatility Measures and GARCH

In finance, we know that risk is defined as uncertainty since we are unable to predict the future more accurately. Based on the assumption that prices follow a lognormal distribution and returns follow a normal distribution, we could define risk as standard deviation or variance of the returns of a security. We call this our conventional definition of volatility (uncertainty). Since a normal distribution is symmetric, it will treat a positive deviation from a mean in the same manner as it would a negative deviation. This is against our conventional wisdom since we treat them differently. To overcome this, Sortino (1983) suggests a lower partial standard deviation. Up to now, we assume that the volatility of a time series is a constant. Obviously this is not true. Another observation is volatility clustering, which means that high volatility is usually followed by a high-volatility period, and this is true for low volatility that is usually followed by a low-volatility period. To model this, Angel (1982) develops an **AutoRegressive Conditional Heteroskedasticity (ARCH)** process, and Bollerslev (1986) extends it to a **Generalized AutoRegressive Conditional Heteroskedasticity (GARCH)** process.

In this chapter, we will cover the following topics:

- Conventional volatility measure – standard deviation – based on a normality assumption
- Test of normality
- Testing fat tail
- An estimation of **lower partial standard deviation (LPSD)** given by Sortino (1983)
- Test of equivalency of volatility over two periods

- Test of heteroskedasticity, Breusch and Pagan (1979)
- Retrieving option data from Yahoo! Finance
- Volatility smile and skewness
- Definition of an **AutoRegressive Conditional Heteroskedasticity (ARCH)** process
- Simulation of an ARCH (1) process
- Definition of a **Generalized AutoRegressive Conditional Heteroskedasticity (GARCH)** process
- Simulation of an GARCH (1,1) process
- Simulation of an GARCH (p,q) process by modifying the `garchSim()` function borrowed from R
- Modeling a GJR_GARCH process by Glosten, Jagannathan, and Runkle (1993)

Conventional volatility measure – standard deviation

In most finance textbooks, we use the standard deviation of returns as a risk measure. This is based on a critical assumption that log returns follow a normal distribution. Even both standard deviation and variance could be used to measure uncertainty; the former is usually called volatility itself. For example, if we say that the volatility of IBM is 20 percent, it means that its annualized standard deviation is 20 percent. Using IBM as an example, the following program is used to estimate its annualized volatility:

```
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
ticker='IBM'
begdate=(2009,1,1)
enddate=(2013,12,31)
p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
ret = (p.aclose[1:] - p.aclose[:-1])/p.aclose[1:]
std_annual=np.std(ret)*np.sqrt(252)
```

From the following output, we know that the volatility is 20.87 percent for IBM:

```
>>>print 'volatility (std)=',round(std_annual,4)
volatility (std)= 0.2087
>>>
```

Tests of normality

The Shapiro-Wilk test is a normality test. The following Python program verifies whether IBM's returns are following a normal distribution. The last five-year daily data from Yahoo! Finance is used for the test. The null hypothesis is that IBM's daily returns are drawn from a normal distribution:

```
from scipy import stats
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
ticker='IBM'
begdate=(2009,1,1)
enddate=(2013,12,31)
p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
ret = (p.aclose[1:] - p.aclose[:-1])/p.aclose[1:]
print 'ticker=',ticker,'W-test, and P-value'
print stats.shapiro(ret)
```

The results are shown as follows:

```
>>> print( 'ticker-',ticker,stats.shapiro(ret))
ticker- IBM (0.9316935539245605, 1.6079492248361013e-23)
>>> print('ticker=',ticker,'W-test, and P-value')
ticker= IBM W-test, and P-value
>>> print(stats.shapiro(ret))
(0.9316935539245605, 1.6079492248361013e-23)
>>>
```

The first value of the result is the test statistic, and the second one is its corresponding p-value. Since this p-value is so close to zero, we reject the null hypothesis. In other words, we conclude that IBM's daily returns do not follow a normal distribution.

For the normality test, we could also apply the Anderson-Darling test, which is a modification of the Kolmogorov-Smirnov test, to verify whether the observations follow a particular distribution. The `stats.anderson()` function has tests for normal, exponential, logistic, and Gumbel (Extreme Value Type I) distributions. The default test is for a normal distribution. After calling the function and printing the testing results, we see the following result:

```
>>>>print stats.anderson(ret)
(14.727130515534327, array([ 0.574,  0.654,  0.785,  0.915,  1.089]),
array([ 15. ,  10. ,  5. ,  2.5,  1. ]))
```


Here, we have three sets of values: the Anderson-Darling test statistic, a set of critical values, and a set of corresponding confidence levels, such as 15 percent, 10 percent, 5 percent, 2.5 percent, and 1 percent as shown in the previous output. If we choose a 1 percent confidence level—the last value of the third set—the critical value is 1.089, the last value of the second set. Since our testing statistic is 14.73, which is much higher than the critical value of 1.089, we reject the null hypothesis. Thus, our Anderson-Darling test leads to the same conclusion as our Shapiro-Wilk test.

Estimating fat tails

One of the important properties of a normal distribution is that we could use mean and standard deviation, the first two moments, to fully define the whole distribution. For n returns of a security, its first four moments are defined in equation (1). The mean or average is defined as follows:

$$\bar{R} = \mu = \frac{\sum_{i=1}^n R_i}{n} \quad (1)$$

Its (sample) variance is defined by the following equation. The standard deviation, that is, σ , is the squared root of the variance:

$$\sigma^2 = \frac{\sum_{i=1}^n (R_i - \bar{R})^2}{n-1} \quad (2)$$

The skewness defined by the following formula indicates whether the distribution is skewed to the left or to the right. For a symmetric distribution, its skewness is zero:

$$skew = \frac{\sum_{i=1}^n (R_i - \bar{R})^3}{(n-1)\sigma^3} \quad (3)$$

The kurtosis reflects the impact of extreme values because of its power of four. There are two types of definitions with and without minus three; refer to the following two equations. The reason behind the deduction of three in equation (4B), is that for a normal distribution, its kurtosis based on equation (4A) is three:

$$kurtosis = \frac{\sum_{i=1}^n (R_i - \bar{R})^4}{(n-1)\sigma^4} \quad (4A)$$

$$(\text{excess})\text{kurtosis} = \frac{\sum_{i=1}^n (R_i - \bar{R})^4}{(n-1)\sigma^4} - 3 \quad (4B)$$

Some books distinguish these two equations by calling equation (4B) excess kurtosis. However, many functions based on equation (4B) are still named kurtosis. Since we know that a standard normal distribution has a zero mean, unit standard deviation, zero skewness, and zero kurtosis (based on equation 4B). The following output confirms these facts:

```
>>> from scipy import stats, random
>>> import numpy as np
>>> ret = random.normal(0,1,500000)
>>> print( 'mean    =', np.mean(ret))
mean    = -0.00143927422656
>>> print( 'std     =', np.std(ret))
std     = 0.999594342278
>>> print( 'skewness=', stats.skew(ret))
skewness= 0.00198825271944
>>> print( 'kurtosis=', stats.kurtosis(ret))
kurtosis= -0.0106677615011
>>>
```

The mean, skewness, and kurtosis are all close to zero, while the standard deviation is close to one. Next, we estimate the four moments for S&P500 based on its daily returns as follows:

```
from scipy import stats
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
ticker='^GSPC'
begdate=(1926,1,1)
enddate=(2013,12,31)
p = quotes_historical_yahoo(ticker, begdate, enddate, asobject=True,
adjusted=True)
ret = (p.aclose[1:] - p.aclose[:-1])/p.aclose[1:]
print( 'S&P500  n          =', len(ret))
print( 'S&P500  mean     =', round(np.mean(ret), 8))
print( 'S&P500  std      =', round(np.std(ret), 8))
print( 'S&P500  skewness=', round(stats.skew(ret), 8))
print( 'S&P500  kurtosis=', round(stats.kurtosis(ret), 8))
```

The output for the five values mentioned in the previous code, including the number of observations, is given as follows:

```
>>> print( 'S&P500 n =',len(ret))
('S&P500 n =', 16102)
>>> print( 'S&P500 mean =',round(np.mean(ret),8))
('S&P500 mean =', 0.00024465)
>>> print( 'S&P500 std =',round(np.std(ret),8))
('S&P500 std =', 0.00981226)
>>> print( 'S&P500 skewness=',round(stats.skew(ret),8))
('S&P500 skewness=', -1.50009687)
>>> print( 'S&P500 kurtosis=',round(stats.kurtosis(ret),8))
('S&P500 kurtosis=', 38.21956524)
>>>
```

This result is very close to the result in the paper titled *Study of Fat-tail Risk by Cook Pine Capital*, which can be downloaded from <http://www.cookpinecapital.com/pdf/Study%20of%20Fat-tail%20Risk.pdf>. Using the same argument, we conclude that the S&P500 daily returns are skewed to the left, that is, a negative skewness, and have fat tails (kurtosis is 38.22 instead of zero).

Lower partial standard deviation

One issue with using standard deviation of returns as a risk measure is that the positive deviation is also viewed as bad. The second issue is that the deviation is from the average instead of a fixed benchmark, such as a risk-free rate. To overcome these shortcomings, Sortino (1983) suggests the lower partial standard deviation, which is defined as the average of squared deviation from the risk-free rate conditional on negative excess returns, as shown in the following formula:

$$LPSD = \frac{\sum_{i=1}^m (R_i - R_f)^2}{n-1}, \text{ where } R_i - R_f > 0 \quad (5)$$

Because we need the risk-free rate in this equation, we could generate a Fama-French dataset that includes the risk-free rate as one of their time series. First, download their daily factors from http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html. Then, unzip it and delete the non-data part at the end of the text file. Assume the final text file is saved under C:/temp/:

```
import pandas as pd
import datetime
file=open("c:/temp/F-F_Research_Data_Factors_daily.txt","r")
data=file.readlines()
f=[]
```

```

index=[]
for i in range(5,size(data)):
    t=data[i].split()
    t0_n=int(t[0])
    y=int(t0_n/10000)
    m=int(t0_n/100)-y*100
    d=int(t0_n)-y*10000-m*100
    index.append(datetime.datetime(y,m,d))
    for j in range(1,5):
        k=float(t[j])
        f.append(k/100)
n=len(f)
f1=np.reshape(f,[n/4,4])
ff=pd.DataFrame(f1,index=index,columns=['Mkt_Rf','SMB','HML','Rf'])
ff.to_pickle("c:/temp/ffDaily.pickle")

```

The name of the final dataset is `ffDaily.pickle`. It is a good idea to generate this dataset yourself. However, the dataset could be downloaded from <http://canisius.edu/~yany/ffDaily.pickle>. Using the last five years' data (January 1, 2009 to December 31, 2013), we could estimate IBM's LPSD as follows:

```

from scipy import stats
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
import pandas as pd
ticker='IBM'
begdate=(2009,1,1)
enddate=(2013,12,31)
p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
ret = (p.aclose[1:] - p.aclose[:-1])/p.aclose[1:]
date_=p.date
x=pd.DataFrame(data=ret,index=date_[:-1],columns=['ret'])
ff=load('c:/temp/ffDaily.pickle')
final=pd.merge(x,ff,left_index=True,right_index=True)
k=final.ret-final.Rf
k2=k[k>0]
LPSD=np.std(k2)*np.sqrt(252)
print(' LPSD (annualized) for ', ticker, 'is ',round(LPSD,3))

```

The following output shows that IBM's LPSD is 14.8 percent quite different from 20.9 percent shown in the previous section:

```
>>> print(' LPSD (annualized) for ', ticker, 'is ',round(LPSD,3))
LPSD (annualized) for  IBM is  nan
>>>
```

Test of equivalency of volatility over two periods

We know that the stock market fell dramatically in October, 1987. We could choose a stock to test the volatility before and after October, 1987. For instance, we could use Ford Motor Corp, with a ticker of `F`, to illustrate how to test the equality of variance before and after the market crash in 1987. In the following Python program, we define a function called `ret_f()` to retrieve daily price data from Yahoo! Finance and estimate its daily returns:

```
import scipy as sp
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
# input area
ticker='F'                # stock
begdate1=(1982,9,1)        # starting date for period  #1
enddate1=(1987,9,1)        # ending   date for period  #1
begdate2=(1987,12,1)       # starting date for period  #2
enddate2=(1992,12,1)       # ending   date for period  #2
# define a function
def ret_f(ticker,begdate,enddate):
    p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
    ret = (p.aclose[1:] - p.aclose[:-1])/p.aclose[1:]
    date_=p.date
    return pd.DataFrame(data=ret,index=date_[:-1],columns=['ret'])
# call the above function twice
ret1=ret_f(ticker,begdate1,enddate1)
ret2=ret_f(ticker,begdate2,enddate2)
# output
```

```
print('Std period #1  vs. std period #2')
print(round(sp.std(ret1.ret),6), round(sp.std(ret2.ret),6))
print('T value ,    p-value ')
print(sp.stats.bartlett(ret1.ret,ret2.ret))
```

The very-high T value and close to zero p-value in the following screenshot suggest the rejection of the hypothesis that during these two periods, the stock has the same volatility. The corresponding output is given as follows:

```
Std period #1  vs. std period #2
(0.020194, 0.017916)
T value ,    p-value
(18.059642154305671, 2.1409163403767813e-05)
>>>
```

Test of heteroskedasticity, Breusch, and Pagan (1979)

Breusch and Pagan (1979) designed a test to confirm or reject the null assumption that the residuals from a regression is homogeneous, that is, with a constant volatility. The following formula represents their logic. First, we run a linear regression of y against x :

$$y_t = \alpha + \beta x_t + \epsilon_t \quad (6)$$

Here, y is the independent variable, x is the independent variable, a is the intercept, β is the coefficient and ϵ_t is an error term. After we get the error term (residual), we run the second regression:

$$\epsilon_t^2 = \gamma_0 + \gamma_1 x_t + v_t \quad (7)$$

Assume that the fitted values from running the previous regression is $f v_t$, then the Breusch-Pangan (1979) measure is given as follows, and it follows a χ^2 distribution with a k degree of freedom:

$$BP = \frac{1}{2} \sum_{i=1}^n f v_i^2 \quad (8)$$

The following example is borrowed from an R package called `lm.test` (test linear regression), and its authors are Hothorn et al. (2014). We generate a time series of x , y_1 and y_2 . The independent variable is x , and the dependent variables are y_1 and y_2 . By our design, y_1 is homogeneous, that is, with a constant variance (standard deviation), and y_2 is non-homogeneous (heterogeneous), that is, the variance (standard deviation) is not constant. For a variable x , we have the following 100 values:

$$x = [-1, 1, -1, 1, \dots, -1, 1] \quad (9)$$

Then, we generate two error terms with 100 random values each. For the `error1`, its 100 values are drawn from the standard normal distribution, that is, with zero mean and unit standard deviation. For `error2`, its 100 values are drawn from a normal distribution with a zero mean and 2 as the standard deviation. The y_1 and y_2 time series are defined as follows:

$$y_1 = x + \text{error1} \quad (10)$$

$$y_2 = x + e_{1,i} [i = 1, 3, \dots, 99] + e_{2,i} [i = 2, 4, 6, \dots, 100] \quad (11)$$

For the odd scripts of y_2 , the error terms are derived from `error1`, while for the even scripts, the error terms are derived from `error2`. To find more information about the PDF file related to `lm.test`, or an R package, we have the following six steps:

1. Go to <http://www.r-project.org>.
2. Click on **CRAN** under **Download, Packages**.
3. Choose a close-by server.
4. Click on **Packages** on the left-hand side of the screen.
5. Choose a list and search **lm.test**.
6. Click the link and download the PDF file related to `lm.test`.

The following is the related Python code:

```
import numpy as np
import statsmodels.api as sm
import scipy as sp

def breusch_pagan_test(y, x):
    results=sm.OLS(y, x).fit()
    resid=results.resid
```

```

n=len(resid)
sigma2 = sum(resid**2)/n
f = resid**2/sigma2 - 1
results2=sm.OLS(f,x).fit()
fv=results2.fittedvalues
bp=0.5 * sum(fv**2)
df=results2.df_model
p_value=1-sp.stats.chi.cdf(bp,df)
return round(bp,6), df, round(p_value,7)

sp.random.seed(12345)
n=100
x=[]
error1=sp.random.normal(0,1,n)
error2=sp.random.normal(0,2,n)
for i in range(n):
    if i%2==1:
        x.append(1)
    else:
        x.append(-1)

y1=x+np.array(x)+error1
y2=zeros(n)

for i in range(n):
    if i%2==1:
        y2[i]=x[i]+error1[i]
    else:
        y2[i]=x[i]+error2[i]

print ('y1 vs. x (we expect to accept the null hypothesis)')
bp=breusch_pagan_test(y1,x)
print('BP value,  df,      p-value')
print 'bp =', bp
bp=breusch_pagan_test(y2,x)

```



```
print ('y2 vs. x (we expect to rject the null hypothesis)')
print('BP value, df, p-value')
print('bp =', bp)
```

For the result of running regression by using y_1 against x , we know that its residual vale would be homogeneous, that is, variance or standard deviation is a constant. Thus, we expect to accept the null hypothesis. The opposite is true for y_2 against x , since, based on our design, the error terms for y_2 are heterogeneous. Thus, we expect to reject the null hypothesis. The corresponding output is shown as follows:

```
y1 vs. x (we expect to accept the null hypothesis)
BP value, df, p-value
bp = (0.596446, 1.0, 0.5508776)
y2 vs. x (we expect to rject the null hypothesis)
BP value, df, p-value
('bp =', (17.611054, 1.0, 0.0))
>>>
```

Retrieving option data from Yahoo! Finance

In the previous chapter, we discussed in detail how to estimate implied volatility with a hypothetic set of input values. To use real-world data to estimate implied volatility, we could define a function with three input variables: `ticker`, `month`, and `year` as follows:

```
def get_option_data(tickrr,exp_date):
    x = Options(ticker,'yahoo')
    puts,calls = x.get_options_data(expiry=exp_date)
    return puts, calls
```

To call the function, we enter three values, such as `IBM`, `2`, and `2014`, when we plan to retrieve options expired in February, 2014. The code with these three values is shown as follows:

```
def from pandas.io.data import Options
import datetime
ticker='IBM'
exp_date=datetime.date(2014,2,28)
puts, calls =get_option_data(ticker,exp_date)
print puts.head()
```

Strike	Symbol	Last	Chg	Bid	Ask	Vol	Open	Int
0	100	IBM140222P00100000	0.01	0	NaN	0.03	16	16
1	105	IBM140222P00105000	0.04	0	NaN	0.03	10	10
2	115	IBM140222P00115000	0.01	0	NaN	0.05	1	2
3	120	IBM140222P00120000	0.02	0	0.01	0.06	10	20
4	130	IBM140222P00130000	0.03	0	0.02	0.06	1	146

Strike	Symbol	Last	Chg	Bid	Ask	Vol	Open	Int
0	150	IBM140222C00150000	30.00	0.00	37.0	40.00	8	10
1	160	IBM140207C00160000	25.30	0.00	27.2	30.00	1	1
2	160	IBM140222C00160000	29.80	0.00	27.1	30.00	2	64
3	165	IBM140222C00165000	25.27	0.00	22.2	24.10	3	55
4	170	IBM140222C00170000	18.82	1.63	18.3	18.65	1	386

>>>

From Yahoo! Finance, we could just retrieve call data and save it. This is also true for the put data. The two output datasets with the Pandas' pickle format can be downloaded from <http://canisius.edu/~yany/callsFeb2014.pickle> and <http://canisius.edu/~yany/putsFeb2014.pickle>:

```
from pandas.io.data import Options
import datetime
import pandas as pd
def call_data(tickrr,exp_date):
    x = Options(ticker,'yahoo')
    data= x.get_call_data(expiry=exp_date)
    return data
ticker='IBM'
exp_date=datetime.date(2014,2,28)
c=call_data(ticker,exp_date)
print c.head()
callsFeb2014=pd.DataFrame(c,columns=['Strike','Symbol','Chg','Bid','Ask',
'Vol','Open Int'])
callsFeb2014.to_pickle('c:/temp/callsFeb2014.pickle')
def put_data(tickrr,exp_date):
    x = Options(ticker,'yahoo')
    data= x.get_put_data(expiry=exp_date)
    return data
```

```
p=put_data(ticker,exp_date)
putsFeb2014=pd.DataFrame(p,columns=['Strike','Symbol','Chg','Bid','Ask','Vol','Open Int'])
putsFeb2014.to_pickle('c:/temp/putsFeb2014.pickle')
```

Volatility smile and skewness

Obviously, each stock should possess just one volatility. However, when estimating implied volatility, different strike prices might offer us different implied volatilities. More specifically, the implied volatility based on out-of-the-money options, at-the-money options, and in-the-money options might be quite different. Volatility smile is the shape going down then up with the exercise prices, while the volatility skewness is downward or upward sloping. The key is that investors' sentiments and the supply and demand relationship have a fundamental impact on the volatility skewness. Thus, such a smile or skewness provides information on whether investors such as fund managers prefer to write calls or puts, as shown in the following code:

```
from pandas.io.data import Options
from matplotlib.finance import quotes_historical_yahoo
# Step 1: define two functions
def call_data(tickrr,exp_date):
    x = Options(ticker,'yahoo')
    data= x.get_call_data(expiry=exp_date)
    return data
def implied_vol_call_min(S,X,T,r,c):
    from scipy import log,exp,sqrt,stats
    implied_vol=1.0
    min_value=1000
    for i in range(10000):
        sigma=0.0001*(i+1)
        d1=(log(S/X)+(r+sigma*sigma/2.)*T)/(sigma*sqrt(T))
        d2 = d1-sigma*sqrt(T)
        c2=S*stats.norm.cdf(d1)-X*exp(-r*T)*stats.norm.cdf(d2)
        abs_diff=abs(c2-c)
        if abs_diff<min_value:
            min_value=abs_diff
            implied_vol=sigma
            k=i
```

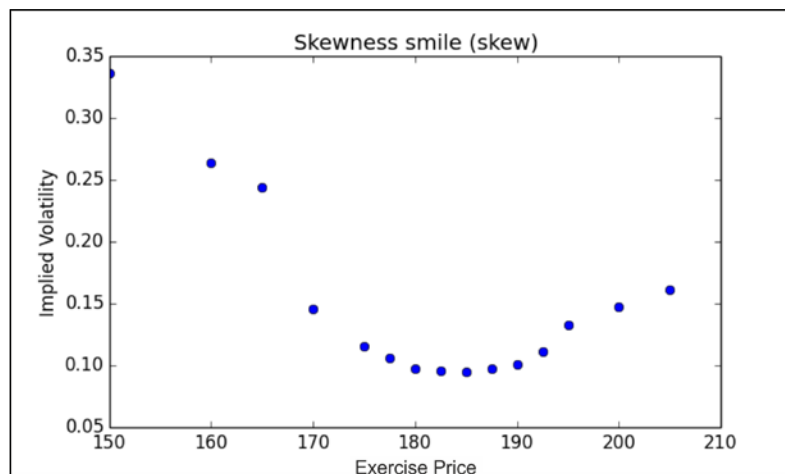
```

    return implied_vol
# Step 2: input area
ticker='IBM'
exp_date=datetime.date(2014,2,28) # first try not exact
r=0.0003 # estimate
begdate=datetime.date(2010,1,1) # this is arbitrary since we care
about current price
# Step 3: get call option data
calls=call_data(ticker,exp_date)
exp_date0=int('20'+calls.Symbol[0][len(ticker):9]) # find examt expiring
date
today=datetime.date.today()
p = quotes_historical_yahoo(ticker, begdate, today, asobject=True,
adjusted=True)
s=p.close[-1] # get current stock price
y=int(exp_date0/10000)
m=int(exp_date0/100)-y*100
d=exp_date0-y*10000-m*100
exp_date=datetime.date(y,m,d) # get exact expiring date
T=(exp_date-today).days/252.0 # T in years
# Step 4: run a loop to estimate the implied volatility
n=len(calls.Strike) # number of strike
strike=[] # initialization
implied_vol=[] # initialization
call2=[] # initialization
x_old=0 # used when we choose the first strike
for i in range(n):
    x=calls.Strike[i]
    c=(calls.Bid[i]+calls.Ask[i])/2.0
    if c > 0:
        print ('i=',i,', c=',c)
        if x!=x_old:
            vol=implied_vol_call_min(s,x,T,r,c)
            strike.append(x)
            implied_vol.append(vol)
            call2.append(c)

```

```
print x,c,vol
x_old=x
# Step 5: draw a smile
title('Skewness smile (skew)')
xlabel('Exercise Price')
ylabel('Implied Volatility')
plot(strike,implied_vol,'o')
```

In this program, if multiple implied volatilities for the same strike price exist, we choose the first implied volatility. Alternatively, we could take the average of several implied volatilities for the same exercise price. The graph of the volatility smile is shown as follows:



Again, if anyone wants to reproduce the previous graph, they can download the call options dataset from <http://canisius.edu/~yany/callsFeb2014.pickle>.

Graphical presentation of volatility clustering

One of the observations is labeled as volatility clustering, which means that high volatility is usually followed by a high-volatility period, while low volatility is usually followed by a low-volatility period. The following program shows this phenomenon by using S&P500 daily returns from 1988 to 2006. Note that, in the following code, in order to show 1988 on the x axis, we add a few months before 1988:

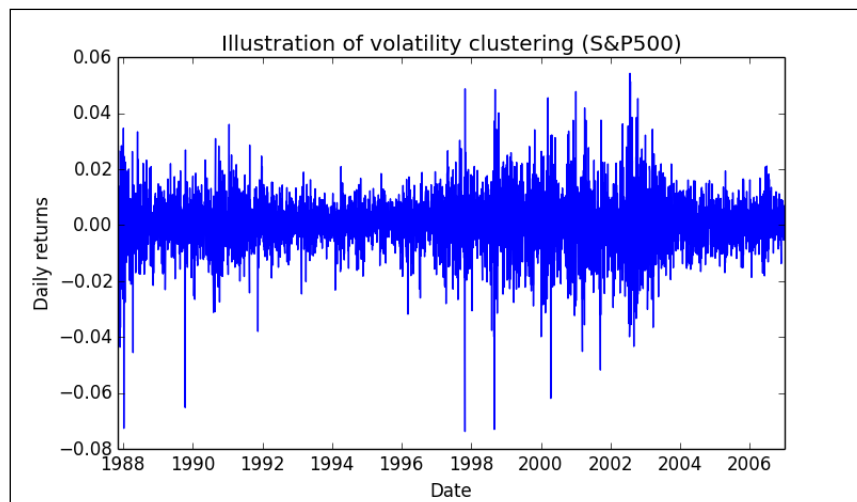
```
from matplotlib.finance import quotes_historical_yahoo
import numpy as np
ticker='^GSPC'
begdate=(1987,11,1)
```

```

enddate=(2006,12,31)
p = quotes_historical_yahoo(ticker, begdate, enddate,asobject=True,
adjusted=True)
ret = (p.aclose[1:] - p.aclose[:-1])/p.aclose[1:]
title('Illustration of volatility clustering (S&P500)')
ylabel('Daily returns')
xlabel('Date')
x=p.date[1:]
plot(x,ret)

```

This program is inspired by the graph drawn by M.P. Visser; refer to <http://staff.science.uva.nl/~marvisse/volatility.html>. The graph corresponding to the previous code is shown as follows:



The ARCH model

Based on previous arguments, we know that the volatility or variance of stock returns is not constant. According to the ARCH model, we could use the error terms from pervious estimation to help us predict the next volatility or variance. This model was developed by *Robert F. Engle*, the winner of the 2003 Nobel Prize in Economics. The formula for an ARCH (q) model is presented as follows:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^q \alpha_i e_{t-1}^2 \quad (12)$$

Here, σ_t^2 is the variance at time t , α_i is the i^{th} coefficient, e_{t-i}^2 is the squared error term for the period of $t-i$, and q is the order of error terms. When q is 1, we have the simplest ARCH (1) process as follows:

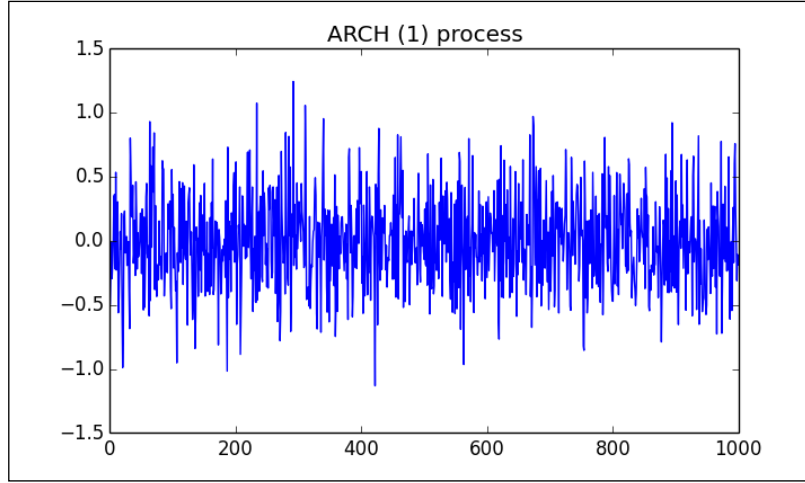
$$\sigma_t^2 = \alpha_0 + \alpha_1 e_{t-1}^2 \quad (13)$$

Simulating an ARCH (1) process

It is a good idea that we simulate an ARCH (1) process and have a better understanding of the volatility clustering, which means that high volatility is usually followed by a high-volatility period while low volatility is usually followed by a low-volatility period. The following code reflects this phenomenon:

```
import scipy as sp
sp.random.seed(12345)
n=1000          # n is the number of observations
n1=100          # we need to drop the first several observations
n2=n+n1         # sum of two numbers
a=(0.1,0.3)     # ARCH (1) coefficients alpha0 and alpha1, see Equation (3)
errors=sp.random.normal(0,1,n2)
t=sp.zeros(n2)
t[0]=sp.random.normal(0,sp.sqrt(a[0]/(1-a[1])),1)
for i in range(1,n2-1):
    t[i]=errors[i]*sp.sqrt(a[0]+a[1]*t[i-1]**2)
y=t[n1-1:-1]    # drop the first n1 observations
title('ARCH (1) process')
x=range(n)
plot(x,y)
```

From the following graph, we see that indeed a higher volatility period is usually followed with high volatility while this is also true for a low-volatility clustering:



The GARCH (Generalized ARCH) model

Generalized AutoRegressive Conditional Heteroskedasticity (GARCH) is an important extension of ARCH, by Bollerslev (1986). The GARCH (p,q) process is defined as follows:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^q \alpha_i \epsilon_{t-i}^2 + \sum_{i=1}^p \beta_i \sigma_{t-i}^2 \quad (14)$$

Here, σ_t^2 is the variance at time t , q is the order for the error terms, p is the order for the variance, α_0 is a constant, α_i is the coefficient for the error term at $t-i$, β_i is the coefficient for the variance at time $t-i$. Obviously, the simplest GARCH process is when both p and q are set to 1, that is, GARCH (1,1), which has following formula:

$$\sigma_t^2 = \alpha_0 + \alpha_1 \epsilon_{t-1}^2 + \beta \sigma_{t-1}^2 \quad (15)$$

Simulating a GARCH process

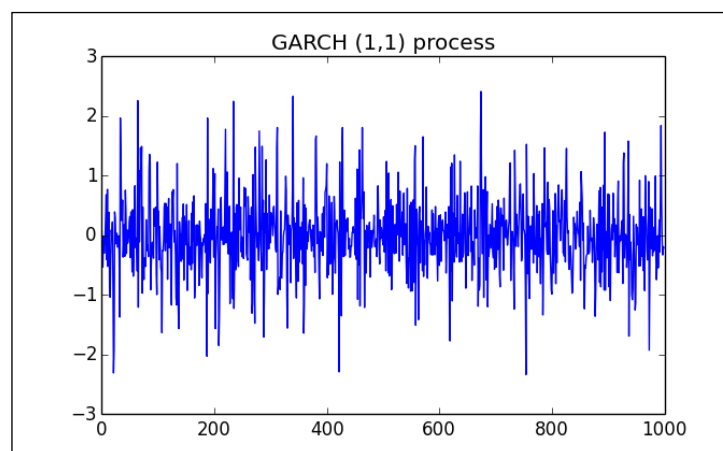
Based on the previous program related to ARCH (1), we could simulate a GARCH (1,1) process as follows:

```
import scipy as sp
sp.random.seed(12345)
n=1000          # n is the number of observations
n1=100          # we need to drop the first several observations
n2=n+n1         # sum of two numbers
alpha=(0.1,0.3) # GARCH (1,1) coefficients alpha0 and alpha1, see
Equation (3)
beta=0.2
errors=sp.random.normal(0,1,n2)
t=sp.zeros(n2)
t[0]=sp.random.normal(0,sp.sqrt(alpha[0]/(1-alpha[1])),1)

for i in range(1,n2-1):
    t[i]=errors[i]*sp.sqrt(alpha[0]+alpha[1]*errors[i-1]**2+beta*t[i-1]**2)

y=t[n1-1:-1]    # drop the first n1 observations
title('GARCH (1,1) process')
x=range(n)
plot(x,y)
```

Honestly speaking, the following graph is quite similar to the previous one under the ARCH (1) process. The graph corresponding to the previous code is shown as follows:



Simulating a GARCH (p,q) process using modified garchSim()

The following code is based on the R function called `garchSim()`, which is included in the R package called `fGarch`. The authors for `fGarch` are *Diethelm Wuertz* and *Yohan Chalabi*. To find the related manual, we perform the following steps:

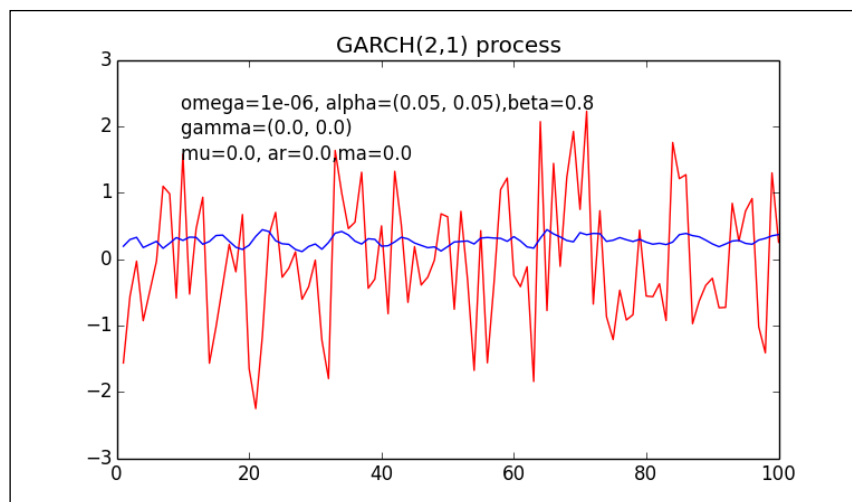
1. Go to <http://www.r-project.org>.
2. Click on **ACRAN** under **Download, Packages**.
3. Choose a close-by server.
4. Click on **Packages** on the left-hand side of the screen.
5. Choose a list and search for **fgarch**.
6. Click on the link and download the PDF file related to `fgarch`.

The Python program based on the R program is given as follows:

```
import scipy as sp
import numpy as np
sp.random.seed(12345)
m=2
n=100          # n is the number of observations
nDrop=100      # we need to drop the first several observations
delta=2
omega=1e-6
alpha=(0.05,0.05)
beta=0.8
mu=ar=ma=ar=0.0
gamma=(0.0,0.0)
order_ar      =size(ar)
order_ma      =size(ma)
order_beta    =size(beta)
order_alpha   =size(alpha)
z0=sp.random.standard_normal(n+nDrop)
deltainv=1/delta
spec_1=spec_2=spec_3=np.array([2])
z = np.hstack((spec_1,z0))
t=np.zeros(n+nDrop)
```

```
h = np.hstack((spec_2,t))
y = np.hstack((spec_3,t))
eps0 = h**deltainv * z
for i in range(m+1,n +nDrop+m-1):
    t1=sum(alpha[:-1]*abs(eps0[i-2:i])) # reverse alpha =alpha[:-1]
    t2=eps0[i-order_alpha-1:i-1]
    t3=t2*t2
    t4=np.dot(gamma,t3.T)
    t5=sum(beta* h[i-order_beta:i-1])
    h[i]=omega+t1-t4+ t5
    eps0[i] = h[i]**deltainv * z[i]
    t10=ar * y[i-order_ar:i-1]
    t11=ma * eps0[i -order_ma:i-1]
    y[i]=mu+sum(t10)+sum(t11)+eps0[i]
garch=y[nDrop+1:]
sigma=h[nDrop+1:]**0.5
eps=eps0[nDrop+1:]
x=range(1,len(garch)+1)
plot(x,garch,'r')
plot(x,sigma,'b')
#plot(x,eps,'g')
title('GARCH(2,1) process')
figtext(0.2,0.8,'omega='+str(omega)+' , alpha='+str(alpha)+' ,beta='+str(beta))
figtext(0.2,0.75,'gamma='+str(gamma))
figtext(0.2,0.7,'mu='+str(mu)+' , ar='+str(ar)+' ,ma='+str(ma))
show()
```

In the preceding program, ω is the constant in equation (10), while α is associated with error terms and β is associated with variance. There are two items in $\alpha[a,b]$: a is for $t-1$, while b is for $t-2$. However, for $\text{eps0}[t-2:i]$, they stand for $t-2$ and $t-1$. The α and eps0 terms are not consistent with each other. Thus, we have to reverse the order of a and b . This is the reason why we use $\alpha[::-1]$. Since several values are zero, such as μ , ar , and ma , the time series of $garch$ is identical with eps . Thus, we show just two time series in the following graph. The high volatility is for $garch$, while the other one is for standard deviation:



GJR_GARCH by Glosten, Jagannathan, and Runkle (1993)

Glosten, Jagannathan, and Runkle (1993) models asymmetry in the GARCH process. They suggest to model $\epsilon_t = \sigma_t z_t$ where z_t is the i.i.d. hypothesis. GJR_GARCH (1,1,1) has the following format:

$$\sigma_t^2 = \omega + \alpha \epsilon_{t-1}^2 + \beta \sigma_{t-1}^2 + \gamma \epsilon_{t-1}^2 I_{t-1} \quad (16)$$

Here, the condition $I_{t-1} = 0$ if $\epsilon_{t-1} \geq 0$ and $I_{t-1} = 1$ if $\epsilon_{t-1} < 0$ holds true. The following code is taken from the Kevin Sheppard website located at:

http://nbviewer.ipython.org/url/www.kevinsheppard.com/images/9/9e/Example_GJR-GARCH.ipynb:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import size, log, pi, sum, diff, array, zeros, diag, dot, mat,
asarray, sqrt
from numpy.linalg import inv
from scipy.optimize import fmin_slsqp
from matplotlib.mlab import csv2rec
```

```
def gjr_garch_likelihood(parameters, data, sigma2, out=None):
    mu = parameters[0]
    omega = parameters[1]
    alpha = parameters[2]
    gamma = parameters[3]
    beta = parameters[4]
    T = size(data,0)
    eps = data-mu
    for t in xrange(1,T):
        sigma2[t]=(omega+alpha*eps[t-1]**2+gamma*eps[t-1]**2*(eps[t-1]<0)+beta*sigma2[t-1])
        logliks = 0.5*(log(2*pi) + log(sigma2) + eps**2/sigma2)
        loglik = sum(logliks)
    if out is None:
        return loglik
    else:
        return loglik, logliks, copy(sigma2)

def gjr_constraint(parameters,data, sigma2, out=None):
    alpha = parameters[2]
    gamma = parameters[3]
    beta = parameters[4]
    return array([1-alpha-gamma/2-beta]) # Constraint
    alpha+gamma/2+beta<=1

def hessian_2sided(fun, theta, args):
    f = fun(theta, *args)
    h = 1e-5*np.abs(theta)
    thetah = theta + h
    h = thetah-theta
    K = size(theta,0)
    h = np.diag(h)
    fp = zeros(K)
    fm = zeros(K)
    for i in xrange(K):
        fp[i] = fun(theta+h[i], *args)
        fm[i] = fun(theta-h[i], *args)
```

```

fpp = zeros((K,K))
fmm = zeros((K,K))
for i in xrange(K):
    for j in xrange(i,K):
        fpp[i,j] = fun(theta + h[i] + h[j], *args)
        fpp[j,i] = fpp[i,j]
        fmm[i,j] = fun(theta-h[i]-h[j], *args)
        fmm[j,i] = fmm[i,j]
hh = (diag(h))
hh = hh.reshape((K,1))
hh = dot(hh,hh.T)
H = zeros((K,K))
for i in xrange(K):
    for j in xrange(i,K):
        H[i,j] = (fpp[i,j]-fp[i]-fp[j] + f+ f-fm[i]-fm[j] +
fmm[i,j])/hh[i,j]/2
        H[j,i] = H[i,j]
return H

```

We can write a function called `GJR_GARCH()` by including all initial values, constraints, and bounds as follows:

```

def GJR_GARCH(ret):
    startV=array([ret.mean(),ret.var()*0.01,0.03,0.09,0.90])
    finfo=np.finfo(np.float64)
    t=(0.0,1.0)
    bounds=[(-10*ret.mean(),10*ret.mean()),(finfo.eps,2*ret.var()),t,t,t]
    T=size(ret,0)
    sigma2=np.repeat(ret.var(),T)
    inV=(ret,sigma2)
    return fmin_slsqp(gjr_garch_likelihood,startV,f_ieqcons=gjr_constraint,
t,bounds=bounds,args=inV)

```

In order to replicate our result, we could use the `random.seed()` function to fix our returns obtained from generating a set of random numbers from a uniform distribution:

```

sp.random.seed(12345)
returns=sp.random.uniform(-0.2,0.3,100)
tt=GJR_GARCH(returns)

```

After we call the `GJR_GARCH()` function by inputting returns, we expect five printed outputs as follows:

```
Optimization terminated successfully. (Exit mode 0)
Current function value: -54.0664734199
Iterations: 12
Function evaluations: 94
Gradient evaluations: 12
>>>
```

The interpretations of these five outputs are given in the following table:

#	Meaning
1	Message describing the exit mode from the optimizer
2	The final value of the objective function
3	The number of iterations
4	Function evaluations
5	Gradient evaluations

The descriptions of various exit modes are listed in the following table:

Exit mode	Description
-1	Gradient evaluation required (g and a)
0	Optimization terminated successfully
1	Function evaluation required (f and c)
2	More equality constraints than independent variables
3	More than $3*n$ iterations in LSQ sub problem
4	Inequality constraints incompatible
5	Singular matrix E in LSQ subproblem
6	Singular matrix C in LSQ subproblem
7	Rank-deficient equality constraint subproblem HFTI
8	Positive directional derivative for line search
9	Iteration limit exceeded

To show our final parameter values, we print our results with the help of the following code:

```
>>>print tt
[ 7.73958583e-02  6.65856138e-03  1.00386156e-12 -1.67115250e-12
 6.61947977e-01]
>>>
```

Summary

In this chapter, we focused on several issues, especially on volatility measures and ARCH/GARCH. For the volatility measures, first we discussed the widely used standard deviation, which is based on the normality assumption. To show that such an assumption might not hold, we introduced several normality tests, such as the Shapiro-Wilk test and the Anderson-Darling test. To show a fat tail of many stocks' real distribution benchmarked on a normal distribution, we vividly used various graphs to illustrate it. To show that the volatility might not be constant, we presented the test to compare the variance over two periods. Then, we showed a Python program to conduct the Breusch-Pagan (1979) test for heteroskedasticity. ARCH and GARCH are used widely to describe the evolvments of volatility over time. For these models, we simulate their simple form such as ARCH (1) and GARCH (1,1) processes. In addition to their graphical presentations, the Python codes of Kevin Sheppard are included to solve the GJR_GARCH (1,1,1) process.

Exercises

1. What is the definition of volatility?
2. How can you measure risk (volatility)?/
3. What are the issues related to the widely used definition of risk (standard deviation)?
4. How can you test whether stock returns follow a normal distribution? For given sets of stocks, test whether they follow a normal distribution.
5. What is the lower partial standard deviation? What are its applications?
6. Choose five stocks, such as DELL, IBM, Microsoft, Citi Group, and Walmart, and compare their standard deviation with LPSD based on the last three-years' daily data.

7. Is a stock's volatility constant over the years?
8. Use the Breusch-Pagan (1979) test to confirm or reject the hypothesis that daily returns for IBM is homogeneous.
9. How can you test whether a stock's volatility is constant?
10. What does "fat tail" mean ? Why should we care about fat tail?
11. How can you download the option data?
12. What is an ARCH (1) process?
13. What is a GARCH (1,1) process?
14. Apply GARCH (1,1) process to DELL.
15. Write a Python program to show the volatility smile by using a set of put options.

Index

Symbols

`_` expression 36
52-week high and low trading strategy 196
`%d` 40
`%` operator 29

A

ActivatePython installation
URL 124
American call
used, for estimating implied volatility 288, 289
American option
about 242
versus European option 242
Amihud's model for illiquidity (2002) 198
Anaconda
Python, launching from 96, 97
URL 96
Anaconda command prompt
used, for launching Python 169
Anaconda installation
URL 125
Anderson-Darling test 349
annotate() function 142
annualized return distribution
estimating 319, 320
annual percentage rate (APR) 55, 99
annual returns
daily returns, converting to 190, 191
annuity
estimating 54, 55
ARCH
about 347, 363, 364

ARCH (1) process, simulating 364
ARCH (1) process
simulating 364
arithmetic average 336
arithmetic mean
versus geometric mean 332
array
logic relationships 110
looping through 108, 293
working with 104
performing 105
array operations
item by item multiplication operation, performing 107
matrix multiplication operation, performing 105, 106
minus operation, performing 105
plus operation, performing 105
Asian options
about 336
advantages 336
Asset-Backed Security (ABS) 10
AutoRegressive Conditional Heteroskedasticity. *See* ARCH

B

barrier options
Down-and-out option 337
pricing, Monte Carlo simulation used 337, 338
Up-and-in option 337
Up-and-out option 337
bear spread with calls trading strategy 251
bear spread with puts trading strategy 251

- binary file**
 - data, reading from 222
 - data, saving to 222
- binary search** 290, 291
- binomial_grid() function** 249, 263, 265
- binomial tree (CRR) method**
 - about 261
 - for American options 268, 269
 - for European options 268
 - graphical representation 262-267
- Black-Scholes-Merton option model** 242, 247, 248
- Bondsonline**
 - URL 174
- bootstrapping**
 - without replacements 317, 318
 - with replacements 317, 318
- Breusch** 355-358
- bs_call() function** 248, 279
- built-in functions**
 - listing 32
- bull spread with calls trading strategy** 251
- bull spread with puts trading strategy** 251
- Bureau of Labor Statistics**
 - URL 174
- butterfly with calls trading strategy** 251, 256, 257
- butterfly with puts trading strategy** 251

C

- calendar spread** 254
- calendar spread trading strategy** 251
- call**
 - pricing, simulation used 334, 335
- call buyer** 243
- call option** 243
- candlesticks**
 - used, to represent daily price 151, 152
- Capital asset pricing model.** *See* **CAPM**
- capitalize() function** 38
- CAPM**
 - about 117-203
 - Fama-French three-factor model 204-206
 - Fama-MacBeth regression 206, 207
 - rolling beta estimation 207-209
 - VaR, using 210, 211
- cash flow** 243
- CBOE**
 - option data, retrieving from 295, 296
 - URL 300
- ceil() function** 48
- Census Bureau**
 - URL 174
- certain files**
 - displaying, in specific subdirectory 63
- Chicago Board Options Exchange.** *See* **CBOE**
- clipboard**
 - data, inputting from 176
- closing price**
 - and trading volume, viewing 156
- Cluster** 109
- CND** 72, 109, 110, 245, 246
- CND() function** 73
- colors**
 - using 137-139
- comment-all-out method**
 - about 75
 - example 75
- comments types**
 - first type comment 51
 - second type comment 52
- compounded interest**
 - defining 129, 130
- Consolidated Quote dataset.** *See* **CQ dataset**
- Consolidated Trade dataset.** *See* **CT dataset**
- Constants** 109
- Consumer Price Index (CPI)** 231
- continuously compounded**
 - interest rate 57, 58
- Cook Pine Capital**
 - URL 352
- CPI (consumer price index)** 26
- CQ dataset** 227-229
- CSV file**
 - data, inputting from 180
- CT dataset**
 - about 226
 - URL 226
- cumulative standard normal distribution.**
 - See* **CND**
- current price**
 - retrieving, from Yahoo! Finance 300

D

daily price

representing, candlesticks used 151, 152

daily returns

converting, to annual

returns 190, 191

converting, to monthly

returns 187-190

data

inputting, from clipboard 176

inputting, from CSV file 180

inputting, from Excel file 179, 180

inputting, from MATLAB dataset 181

inputting, from text file 178, 179

outputting, to external files 221

outputting, to text file 222

reading, from binary file 222

retrieving, from external text file 118

retrieving, from web page 180, 181

saving, to binary file 222

DataFrame

looping through 293

used, for working with time series 183-185

dataset

URL 231, 316

datasets

merging, by date 191, 192

n-stock portfolio, forming 192, 193

data types 58, 119

date

datasets, merging by 191, 192

datetime.date.today() function 145

date variables

used, for working with time series 183

default input values

used, for functions 45

default precision

choosing 31

del() function 27

delta 258

delta_call() function 258

delta_put function 258

dictionary

looping through 294

different correlations

impact of 326, 329

different shapes

using 139

dir2() function 45, 63

dir() function

about 33, 46, 63, 102

using, for finding variables 26

DOS window

Python, launching 15

used, for launching Python 169

Down-and-out option 337

DuPont identity

working with 133, 134

E

e (2.71828) 34

Economics module 91

effective annual rate (EAR) 55

efficiency

measuring, by time spent 289

efficient frontier

constructing 211

constructing, n stocks used 217, 219

constructing, with n stocks 329

finding, based on two stocks 324, 326

optimal portfolio,

constructing 215-217

variance-covariance matrix,

estimating 212-214

variance-covariance matrix optimization

214, 215

empty shell method

about 73

describing 73, 74

Enter key 18

enumerate() function 60, 281

equal means test

performing 195

equal variances test

performing, sp.stats.bartlett used 194

error message

abcde variable, error message 25

about 16, 25

European option

about 242

versus American option 242

- European options**
 - with known dividends 250, 251
- Excel file**
 - data, inputting from 179, 180
- existence**
 - checking, of functions 46, 47
- exit modes** 372
- exotic options**
 - about 335
 - barrier options pricing, Monte Carlo simulation used 337, 338
 - Monte Carlo simulation, using 335
- exp() function** 72
- expiration dates**
 - retrieving, from Yahoo! Finance 299
 - URL 299
- external text file**
 - data, retrieving from 118

F

- Fama-French dataset** 352
- Fama-French three-factor model** 204-206
- Fama-MacBeth regression** 206, 207
- fat tails**
 - estimating 350, 351
- Federal Reserve Bank Data Library**
 - URL 174
- Fftpack** 109
- fGarch** 367
- figure**
 - saving, to file 159, 160
- file**
 - figure, saving to 159, 160
- File | New Window Ctrl + N** 75
- fin101() function** 64
- finance related Python modules**
 - Economics 91
 - Finance 91
 - FinDates 92
 - Quant 91
 - trytond_account_statement 91
 - trytond_analytic_account 91
 - trytond_currency 91
 - trytond_project 91

- trytond_stock_forecast 91
- trytond_stock_split 91
- Ystockquote 91
- financial calculator**
 - Python, using as 64
- FinDates module** 92
- first type comment** 51
- floating strikes**
 - lookback options, pricing with 342, 343
- floor function**
 - using 28
- for loop**
 - about 277
 - assigning through 294
 - IRR, estimating via 282, 283
 - used, for estimating implied volatility 278, 279
- from math import *** 82
- F-test** 194
- functions**
 - activating, import function used 48
 - default input values, using for 45
 - defining, from Python editor 47
 - displaying, in NumPy 102
 - displaying, in SciPy 102
 - existence, checking 46, 47
 - finding, from imported module 116
 - print() function 36
 - type() function 36
 - upper() function 37, 38

G

- GARCH**
 - about 365
 - GARCH (p,q) process simulating, modified garchSim() used 367, 368
 - process, simulating 366
- GARCH (p,q) process**
 - simulating, modified garchSim() used 367, 368
- garchSim() R function** 367
- Generalized AutoRegressive Conditional Heteroskedasticity.** *See* GARCH
- genfromtxt() function** 118
- geometric average** 336

- geometric mean**
 - versus arithmetic mean 332
- GJR_GARCH() function** 371, 372
- Glosten model** 369
- Google Finance**
 - URL 174
- graph**
 - mathematical formulae, adding to 157, 158
 - simple images, adding to 158
 - texts, adding to 131, 132
- Greek letters**
 - for options 258, 259
- GUI**
 - used, for Python launching 13

H

- head() function** 298
- hedging strategies** 269, 270
- help function**
 - using 108
- help() function** 31, 32, 108, 308
- help(round) function** 32
- help window**
 - finding 18, 19
- heteroskedasticity** 355
- high-frequency data**
 - about 223
 - retrieving, from Google Finance 223, 224
 - spread estimation based 227-229
 - TAQ 226
 - TAQ database 223, 224
 - TORQ database 226
- High minus Low (HML)** 231
- histogram**
 - about 310
 - used, for displaying return distribution 145-148
- historical price data**
 - retrieving, from Yahoo! Finance 144, 177
- HML (High Minus Low)** 317

I

- IBM option data**
 - URL 295
- if() function** 53, 54

- implied volatility**
 - about 276
 - estimating, American call used 288, 289
 - estimating, for loop used 278, 279
 - estimating, while loop used 286, 287
- implied volatility function**
 - based on European call 279
 - based on put option model 280, 281
- imported module**
 - all functions, displaying 82
 - deleting 83
 - functions, finding from 116
 - location, finding 87, 88
 - short name, adopting for 81
- import function**
 - used, for activating functions 48
- import math** 83
- in-and-out parity** 339, 340
- indentation**
 - in Python 45, 46
- input values, and option values**
 - relationship 257
- installation**
 - Python 12
- Integrate** 109
- interest rates**
 - converting 55-57
- internal rate of return.** *See* IRR
- International Business Machines (IBM)** 10
- Interpolate** 109
- interpolation technique** 220, 221
- intra-day graphical representations** 154-156
- intra-day pattern**
 - URL 156
- Io** 109
- IRR**
 - about 136
 - defining 61, 63
 - estimating, via for loop 282, 283
- IRR rule**
 - defining 61, 63
- isnan() function** 282
- item by item multiplication operation**
 - performing 107
- items() function** 295

J

Jagannathan model 369

January effect

testing 195

join() function 189

K

Kolmogorov-Smirnov test 349

kurtosis 351

L

LaTeX

URL 158

LEGB rule 30

len() function 39, 145

Linalg 109

linear equations

solving, SciPy used 113, 114

Linear regression and Capital Assets Pricing

Model. *See* CAPM

linspace() function 127

list data type 103, 104

loadmat() function 181

loadtxt() function 118

lo() function 76

log() function 72

logic relationships 110

lognormal distribution

graphical presentation 311, 312

long-term return forecast 333, 334

lookback options

pricing, with floating strikes 342, 343

loss function

for call option 240

lower partial standard

deviation (LPSD) 347, 352, 353

M

manuals, Python

finding 19, 20

online tutorials 21

PDF version 21

market returns

and stock, comparing 148

mathematical formulae

adding, to graph 157, 158

math import * 34

math module

about 72

e (2.71828) 34

importing 33

pi (3.14159265) 34

MATLAB dataset

data, inputting from 181

matplotlib

alternative installation, via Anaconda 125

installing, via ActivatePython 124, 125

URL 163

using 125-128

matplotlib module

about 87

installing 163

matrix multiplication operation

performing 105, 106

mean() function 332

meaningful variable names

choosing 25, 26

min_value variable 281

module

about 80-89

available modules, finding 86, 87

built-in modules 85

dependency approaches 91

exp() function, importing 84

importing 80

log() function, importing 84

short name, adopting for 81

specific uninstalled module, finding 90

sqrt(), importing 84

Monte Carlo simulation

used, for pricing barrier options 337, 338

using 335

monthly returns

daily returns, converting to 187-190

M.P. Visser 363

m stocks

random selection, from n

given stocks 315, 316

multiple IRRs

estimating 283

N

- Ndimimage** 109
- Nested (multiple) for loops** 288
- Net present value.** *See* NPV
- normal distribution**
 - about 243
 - drawing 244
 - histogram 310
 - n random numbers, generating from 310
 - random samples, drawing from 309
- normality test** 349, 350
- normdist() function** 74
- np.argmax() function** 108
- np.array() function** 98
- np.irr() function** 137
- np.linspace() function** 112
- np.min() function** 108
- np.npv() function** 99
- np.random.normal() function** 128
- np.size() function** 97
- np.std() function** 97
- NPV**
 - about 59, 60
 - defining 135
- npv_f() function** 282
- NPV() function** 99, 281
- NPV profile**
 - about 135, 136
 - colors, using 137, 139
 - different shapes, using 139
- NPV rule** 59, 60
- n random numbers**
 - generating, from normal distribution 310
- n-stock portfolio**
 - forming 192, 193
- n stocks**
 - efficient frontier, constructing with 329
 - used, for constructing an efficient frontier 217-219
- NumPy**
 - functions, displaying in 102
 - installing 96, 119
- NumPy module** 87
- numpy.random function** 308
- NumPy, using**
 - examples 97, 98

O

- Odr** 109
- OLS regression**
 - using 173
- one dimensional time series**
 - DataFrame, using 183-185
 - date variables, using 183
 - generating, pd.Series()
 - function used 182, 183
- open data sources**
 - Bondsonline 174
 - Bureau of Labor Statistics 174
 - Census Bureau 174
 - Federal Reserve Bank Data Library 174
 - Google Finance 174
 - Prof. French's Data Library 174
 - Russell indices 174
 - U.S. Department of the Treasury 174
 - Yahoo! Finance 174
 - Yahoo! Finance, downloading from 175
- optimal portfolio**
 - constructing 215-217
- optimization** 116
- optimization, variance-covariance**
 - matrix 214, 215
- optimize** 109
- option data**
 - retrieving, from CBOE 295, 296
 - retrieving, from Yahoo! Finance 297
 - retrieving, Yahoo! Finance 358
- ordinary least square regression.** *See* OLS regression
- over-the-counter (OTC)** 335
- own module**
 - generating 50

P

- p4f module**
 - for options 248, 249
- Pagan** 355-358
- Pandas**
 - installing 168
 - used, for data manipulation 171-173
- Pandas module** 11

- Pandas pickle format**
 - URL 292
- Pastor and Stambaugh (2003) liquidity measure** 199, 201
- path**
 - project directory, adding to 65
- Path Browser** 89
- path function** 65
- payback period**
 - defining 60
- payback period rule**
 - defining 60
- payoff function**
 - for call option 238, 239
- pd.DataFrame() function** 185
- pd.interpolate() function** 220
- pd.ols function** 207
- pd.read_clipboard() function** 176
- pd.read_csv() function** 180
- pd.Series() function**
 - about 182
 - used, for generating one dimensional time series 182, 183
- permutation() function** 317
- pi (3.14159265)** 34
- PIN (Probability of informed trading)** 315
- pi value**
 - estimating, simulation used 313, 314
- plt.bar() function** 134
- plus operation**
 - performing 105
- Poisson distribution**
 - random numbers, generating from 315
- portfolio diversification effect**
 - graphical representation 140-142
 - number of stocks 142-144
 - portfolio risk 142-144
- power function**
 - about 30
 - using 28, 29
- print() function** 36, 308
- Prof. French's Data Library**
 - URL 174
- profit function**
 - for call option 240
- program**
 - debugging 76
 - debugging, from Python editor 48, 49
- project directory**
 - adding, to path 65
- put-call parity**
 - about 259
 - graphical representation 260, 261
- put-call ratio**
 - about 300
 - for shorter period 302, 303
- put option 241, 243**
 - about 47, 52
 - calling 49
- pv() function** 97
- Python**
 - about 10
 - addition operation 28
 - benefits 10, 11
 - division operation 28
 - help window, finding 18
 - installing 12
 - launching, Anaconda command prompt used 169
 - launching, DOS window used 169
 - launching, from Anaconda 96, 97
 - launching, from own DOS window 15
 - launching, from Python command line 14
 - launching, Spyder used 170, 171
 - launching, ways 12
 - launching, with GUI 13, 14
 - modules 11
 - multiplication operation 28
 - quitting, ways 16, 17
 - shortcoming 11
 - subtraction operation 28
 - used, as financial calculator 64
 - version, finding 21
 - versions 12
- Python code**
 - about 71
 - for down-and-in put option 338
- Python command line**
 - Python, launching from 14
- Python editor**
 - functions, defining from 47
 - program, debugging from 48, 49

- Python function**
 - writing 44
- Python home documents** 20
- Python Manuals**
 - finding 20
- Python module.** *See* **module**
- Python Package Index**
 - URL 92

Q

- quant** 80
- Quant module** 91

R

- randint() function** 316
- random access**
 - versus sequential access 292
- random numbers**
 - generating, from Poisson distribution 315
 - generating, from standard normal distribution 308
 - generating, from uniform distribution 312, 313
 - generating, with seed 114, 115, 309, 310
- random.rand() function** 115
- random samples**
 - drawing, from normal distribution 309
- random.seed() function** 371
- randrange() function** 316
- range() function** 277
- read_csv() function** 181
- read_table() function** 179
- remainder** 28
- remove() function** 317
- ret_f() function** 354
- return**
 - versus volatility, comparing 161, 162
- return distribution**
 - displaying, histogram used 145-148
- return estimation**
 - about 185-187
 - daily returns, converting to annual returns 190, 191
 - daily returns, converting to monthly returns 187, 189

- Return on Equity (ROE)** 133
- rolling beta**
 - estimating 207-209
- Roll's model to estimate spread (1984)** 197, 198
- round() function** 32
- r.sort() function** 145
- Runkle model** 369
- Run Module F5** 72

S

- SciPy**
 - functions, displaying in 102
 - installing 96
 - interpolating in 112
 - stats 111, 112
 - subpackages 109
 - used, for solving linear equations 113, 114
- SciPy module** 72
- SciPy, subpackages**
 - Cluster 109
 - Constants 109
 - Fftpack 109
 - Integrate 109
 - Interpolate 109
 - Io 109
 - Linalg 109
 - Ndimimage 109
 - Odr 109
 - optimize 109
 - signal 109
 - sparse 109
 - spatial 109
 - special 109
 - stats 109
- SciPy, using**
 - examples 98-101
- second type comment** 52
- Securities and Exchange Commission (SEC)** 10
- seed**
 - random numbers, generating with 114, 115, 309, 310
- seed() function** 309, 312, 322
- sequential access**
 - versus random access 292

- Shapiro-Wilk test** 349
- signal** 109
- sign() function** 287
- simple images**
 - adding, to graph 158
- simple interest**
 - defining 129, 130
- simulation**
 - used, for pi value estimation 313, 314
 - used, for pricing call 334, 335
- skewness** 351, 360
- Small minus Big (SMB)** 231
- SMB (Small Minus Big)** 316
- smile** 360
- Sobol sequence**
 - URL 344
 - used, for improving efficiency 344
- Sortino** 347
- sparse** 109
- spatial** 109
- special** 109
- specific function** 103
- specific subdirectory**
 - certain files, displaying 63
- specific uninstalled module**
 - finding 90
- sp.fv() function** 100
- sp.npv() function** 100
- sp.pmt() function** 99
- sp.prod() function** 101
- sp.pv() function** 100
- spread estimation**
 - based on, high-frequency data 227-229
- sp.stats.bartlett function**
 - used, for testing equal variance 194
- Spyder**
 - URL 229
 - used, for launching Python 170, 171
 - using 229, 230
- sqrt(3) command** 24
- sqrt() function** 25, 34, 72, 80, 84
- standard normal distribution**
 - about 244
 - random numbers, generating from 308
- stats** 109, 111, 112
- stats.anderson() function** 349

- statsmodels**
 - about 12
 - installing 168
 - OLS regression method 173
 - using, for statistical analysis 173
- stats.norm.cdf() function** 246, 248
- stats.norm.pdf() function** 244
- std() function** 103
- stock**
 - and market returns, comparing 148
 - performance, comparing among 160
- stock price movements**
 - simulating 320-322
- straddle trading strategy** 251, 253, 254
- strangle trading strategy** 251
- strap trading strategy** 251
- string.replace() function** 223
- strip() function** 37, 38
- strip trading strategy** 251
- sys module** 21

T

- tail() function** 298
- terminal stock prices**
 - estimating 322, 323
- text file**
 - data, inputting from 178, 179
 - data, outputting to 222
- texts**
 - adding, to graph 131, 132
- time value, money**
 - defining 150
- TORQ database**
 - about 226
 - URL 226
- Trade, Order, Report, and Quotation. *See***
TORQ database
- trading strategies**
 - about 251
 - bear spread with calls 251
 - bear spread with puts 251
 - bull spread with calls 251
 - bull spread with puts 251
 - butterfly with calls 251, 256, 257
 - butterfly with puts 251
 - calendar spread 251, 254

- covered call 252
- straddle 251-254
- strangle 251
- strap 251
- strip 251
- trading volume**
 - and closing price, viewing 156
- trytond_account_statement module** 91
- trytond_currency module** 91
- trytond_project module** 91
- trytond_stock_forecast module** 91
- trytond_stock_split module** 91
- T-test**
 - about 193
 - equal means test, performing 194
 - equal variances test, performing 194
 - January effect, testing 195
 - performing 193, 194
- ttest_1samp() function** 111
- tuple data type** 39, 40
- two strings**
 - combining 37
- two-year price movement**
 - graphical representation 153, 154
- type() function** 36

U

- uniform distribution**
 - random numbers, generating from 312, 313
- unique() function** 228, 317
- Up-and-in option** 337
- up-and-in parity**
 - graphical representation 340-342
- Up-and-out option** 337
- up-and-out parity**
 - graphical representation 340-342
- upper() function** 37, 38
- U.S. Department of the Treasury**
 - URL 174
- useful applications**
 - 52-week high and low trading strategy 196
 - Amihud's model for
 - illiquidity (2002) 198, 199
 - Pastor and Stambaugh (2003)
 - liquidity measure 199- 201
 - Roll's model to estimate

- spread (1984) 197, 198

V

- Value at Risk. See VaR**
- values**
 - assigning, to variables 24
- vanilla options** 335
- VaR**
 - using 210, 211
- variable**
 - deleting 27
 - initializing 17
 - unsigned 27
 - values, assigning to 24
 - values, displaying 24
- variance-covariance matrix**
 - estimating 212, 214
 - optimization 214, 215
- versions, Python**
 - finding 21
- Visual financial statements**
 - URL 163
- volatility**
 - about 348, 360
 - over two periods, equivalency testing 354
 - versus return, comparing 161, 162
- volatility clustering** 362, 363
- volatility skewness** 360, 362
- volatility smile** 360, 362

W

- web page**
 - data, retrieving from 180, 181
- web page examples**
 - URL 163
- while loop**
 - about 284
 - used, for estimating implied volatility 286, 287

X

- xlim() function** 130
- x.sum() dot function** 107

Y

Yahoo! Finance

- current price, retrieving from 300
- different expiring dates 299
- historical price data, retrieving
 - from 144, 177
- option data, retrieving from 297, 358
- URL 174, 297

yanMonthly.pickle

- URL 293

ylim() function 130

Ystockquote 91



Thank you for buying Python 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, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

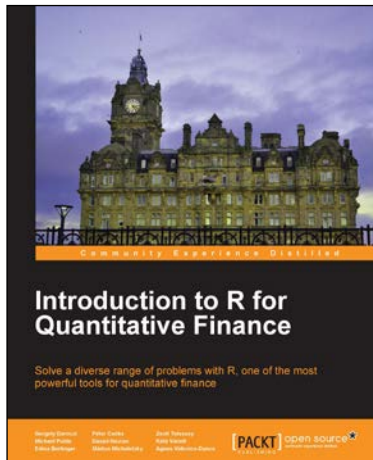
About Packt Open Source

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 licences, 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, 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.



Introduction to R for Quantitative Finance

ISBN: 978-1-78328-093-3 Paperback: 164 pages

Solve a diverse range of problems with R, one of the most powerful tools for quantitative finance

1. Use time series analysis to model and forecast house prices.
2. Estimate the term structure of interest rates using prices of government bonds.
3. Detect systemically important financial institutions by employing financial network analysis .

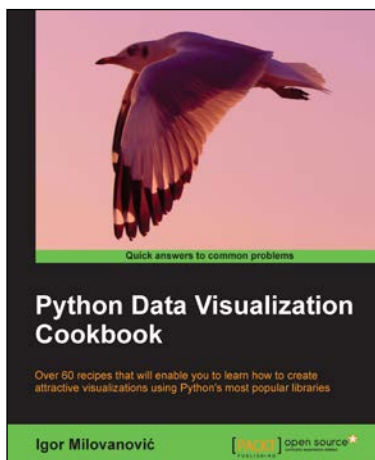


Python High Performance Programming

ISBN: 978-1-78328-845-8 Paperback: 108 pages

Boost the performance of your Python programs using advanced techniques

1. Identify the bottlenecks in your applications and solve them using the best profiling techniques.
2. Write efficient numerical code in NumPy and Cython.
3. Adapt your programs to run on multiple processors with parallel programming.

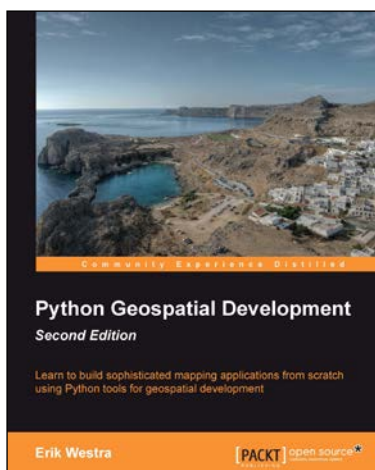


Python Data Visualization Cookbook

ISBN: 978-1-78216-336-7 Paperback: 280 pages

Over 60 recipes that will enable you to learn how to create attractive visualizations using Python's most popular libraries

1. Learn how to set up an optimal Python environment for data visualization.
2. Understand the topics such as importing data for visualization and formatting data for visualization.
3. Understand the underlying data and how to use the right visualizations.



Python Geospatial Development Second Edition

ISBN: 978-1-78216-152-3 Paperback: 508 pages

Learn to build sophisticated mapping applications from scratch using Python tools for geospatial development

1. Build your own complete and sophisticated mapping applications in Python.
2. Walks you through the process of building your own online system for viewing and editing geospatial data.
3. Practical, hands-on tutorial that teaches you all about geospatial development in Python.

Please check www.PacktPub.com for information on our titles