



1ST EDITION

Python for Algorithmic Trading Cookbook

Recipes for designing, building, and deploying
algorithmic trading strategies with Python



JASON STRIMPEL



Python for Algorithmic Trading Cookbook

Copyright © 2024 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 or its dealers and distributors, will be held liable for any damages caused or alleged to have been 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.

Group Product Manager: Apeksha Shetty

Publishing Product Manager: Nilesh Kowadkar

Book Project Manager: Hemangi Lotlikar

Senior Editor: Nazia Shaikh

Technical Editor: Sweety Pagaria

Copy Editor: Safis Editing

Proofreader: Nazia Shaikh

Indexer: Rekha Nair

Production Designer: Ponraj Dhandapani

Senior DevRel Marketing Coordinator: Nivedita Singh

First published: August 2024

Production reference: 1120724

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83508-470-0

www.packtpub.com

Contributors

About the author

Jason Strimpel is the founder of PyQuant News and co-founder of Trade Blotter. His career America, Europe, and Asia over the last 20+ years. He previously traded for a Chicago-based hedge fund, was a risk manager at JPMorgan, and managed production risk technology for an energy derivatives trading firm in London. In Singapore, he served as APAC CIO for an agricultural trading firm and built the data science team for a global metals trading firm. Jason holds degrees in finance and economics and a Master's in Capitalize Quantitative Finance from the Illinois Institute of Technology. He shares his expertise through the PyQuant Newsletter, social media, and teaches *Getting Started With Python for Quant Finance*.

About the reviewers

Sudarshan Sawal specializes in quantitative finance with a background in economics and engineering. As Head of Products (Quant) at PredictNow.ai, he enhanced trading models and boosted client portfolios, with notable research in portfolio optimization using **Conditional Portfolio Optimization (CPO)** methodology. Previously at QTS Capital Management, Sudarshan researched trading algorithms and assisted clients with machine learning models. He's represented PredictNow.ai at various financial conferences.

Brian Schneider holds a Bachelor of Science degree in electrical engineering and a Master of Science in financial engineering. He has 25 years of experience in systems engineering, software design and architecture, algorithm performance, data science, and analytics. He has a passion for financial markets and automating algorithmic trading systems and co-founded an investment management company in that space.

Chris Conlan is the founder and CEO of Conlan Scientific, a financial data science consultancy based out of Charlotte, North Carolina. He works with his team of data scientists to build machine learning solutions for banks, lenders, investors, traders, and fintech companies. Chris holds a Bachelor's in statistics from the University of Virginia where he later taught data science capstone courses at the undergraduate and graduate levels. Chris serves on the Board of Advisors for the Masters of Science in Finance and Analytics degree program at Queens University of Charlotte.

Table of Contents

Preface

1

Acquire Free Financial Market Data with Cutting-Edge Python Libraries

Technical requirements

Diving into continuous futures data with Nasdaq Data Link

Getting ready...

How to do it...

How it works...

There's more...

See also

Exploring S&P 500 ratios data with Nasdaq Data Link

How to do it...

How it works...

There's more...

See also

Working with stock market data with the OpenBB Platform

Getting ready...

How to do it...

How it works...

There's more...

See also

Fetching historic futures data with the OpenBB Platform

Getting ready...

How to do it...

There's more...

See also

Navigating options market data with the OpenBB Platform

Getting ready...

How to do it...

How it works...

There's more...

See also

Harnessing factor data using pandas_datareader

Getting ready...

How to do it...

How it works...

There's more...

See also

2

Analyze and Transform Financial Market Data with
pandas

Diving into pandas index types

How to do it...

How it works...

There's more...

See also

Building pandas Series and DataFrames

Getting ready

How to do it...

How it works...

There's more...

See also

Manipulating and transforming DataFrames

Getting ready...

How to do it...

How it works...

There's more...

See also

Examining and selecting data from DataFrames

How to do it...

How it works...

There's more...

See also

Calculating asset returns using pandas

How to do it...

How it works...

There's more...

See also

Measuring the volatility of a return series

How to do it...

How it works...

There's more...

See also

Generating a cumulative return series

Getting ready...

How to do it...

How it works...

See also

Resampling data for different time frames

How to do it...

How it works...

There's more...

See also

Addressing missing data issues

Getting ready...

How to do it...

How it works...

There's more...

See also

Applying custom functions to analyze time series data

Getting ready...

How to do it...

How it works...

There's more...

See also

3

Visualize Financial Market Data with Matplotlib, Seaborn, and Plotly Dash

Quickly visualizing data using pandas

How to do it...

How it works...

There's more...

See also

Animating the evolution of the yield curve with Matplotlib

How to do it...

How it works...

There's more...

See also

Plotting options implied volatility surfaces with Matplotlib

Getting ready...

How to do it...

How it works...

There's more...

See also

Visualizing statistical relationships with Seaborn

How to do it...

How it works...

There's more...

See also

Creating an interactive PCA analytics dashboard with Plotly Dash

Getting ready...

How to do it...

How it works...

There's more...

See also

4

Store Financial Market Data on Your Computer

Storing data on disk in CSV format

How to do it...

How it works...

There's more...

See also...

Storing data on disk with SQLite

Getting ready...

How to do it...

How it works...

There's more...

See also...

Storing data in a PostgreSQL database server

Getting ready...

How to do it...

How it works...

There's more...

See also...

Storing data in ultra-fast HDF5 format

Getting ready...

How to do it...

How it works...

There's more...

See also...

5

Build Alpha Factors for Stock Portfolios

Identifying latent return drivers using principal component analysis

Getting ready

How to do it...

How it works...

There's more...

See also

Finding and hedging portfolio beta using linear regression

Getting ready

How to do it...

How it works...

There's more...

See also

Analyzing portfolio sensitivities to the Fama-French factors

Getting ready

How to do it...

How it works...

There's more...

See also

Assessing market inefficiency based on volatility

How to do it...

How it works...

There's more...

See also

Preparing a factor ranking model using Zipline Pipelines

Getting ready

How to do it...

How it works...

There's more...

See also

6

Vector-Based Backtesting with VectorBT

Building technical strategies with VectorBT

Getting ready

How to do it...

How it works...

There's more...

See also

Conducting walk-forward optimization with VectorBT

Getting ready

How to do it...

How it works...

There's more...

See also

Optimizing the SuperTrend strategy with VectorBT Pro

Getting ready

How to do it...

How it works...

There's more...

See also

7

Event-Based Backtesting Factor Portfolios with Zipline Reloaded

Technical Requirements

For Windows, Unix/Linux, and Mac Intel users

For Mac M1/M2 users

Backtesting a momentum factor strategy with Zipline Reloaded

Getting ready

How to do it...

How it works...

There's more...

See also

Exploring a mean reversion strategy with Zipline Reloaded

Getting ready

How to do it...

How it works...

There's more...

See also

8

Evaluate Factor Risk and Performance with Alphalens Reloaded

Preparing backtest results

Getting ready...

How to do it...

How it works...

There's more...

See also

Evaluating the information coefficient

Getting ready...

How to do it...

How it works...

There's more...

See also

Examining factor return performance

How to do it...

How it works...

There's more...

See also

Evaluating factor turnover

How to do it...

How it works...

There's more...

See also

9

Assess Backtest Risk and Performance Metrics with Pyfolio

Preparing Zipline backtest results for Pyfolio Reloaded

Getting ready...

How to do it...

How it works...

There's more...

See also

Generating strategy performance and return analytics

Getting ready...

How to do it...

How it works...

There's more...

See also

Building a drawdown and rolling risk analysis

Getting ready...

How to do it...

How it works...

There's more...

See also

Analyzing strategy holdings, leverage, exposure, and sector allocations

Getting ready...

How to do it...

How it works...

There's more...

See also

Breaking Down Strategy Performance to Trade Level

Getting ready...

How to do it...

How it works...

There's more...

See also

10

Set Up the Interactive Brokers Python API

[Building an algorithmic trading app](#)

[Getting ready...](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Creating a Contract object with the IB API](#)

[Getting ready...](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Creating an Order object with the IB API](#)

[Getting ready...](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Fetching historical market data](#)

Getting ready...

How to do it...

How it works...

There's more...

See also

Getting a market data snapshot

Getting ready...

How to do it...

How it works...

There's more...

See also

Streaming live market data

Getting ready...

How to do it...

How it works...

There's more...

See also

Storing live tick data in a local SQL database

Getting ready...

How to do it...

How it works...

There's more...

See also

11

Manage Orders, Positions, and Portfolios with the IB API

Executing orders with the IB API

Getting ready

How to do it...

How it works...

There's more...

See also

Managing orders once they're placed

Getting ready

How to do it...

How it works...

There's more...

See also

Getting details about your portfolio

Getting ready

How to do it...

How it works...

There's more...

See also

Inspecting positions and position details

Getting ready

How to do it...

How it works...

There's more...

See also

Computing portfolio profit and loss

Getting ready

How to do it...

How it works...

There's more...

See also

Deploy Strategies to a Live Environment

Calculating real-time key performance and risk indicators

Getting ready

How to do it...

How it works...

There's more...

See also

Sending orders based on portfolio targets

Getting ready

How to do it...

How it works...

There's more...

See also

Deploying a monthly factor portfolio strategy

Getting ready

How to do it...

How it works...

There's more...

See also

Deploying an options combo strategy

Getting ready

How to do it...

How it works...

There's more...

See also

Deploying an intraday multi-asset mean reversion strategy

Getting ready

How to do it...

How it works...

There's more...

See also

13

Advanced Recipes for Market Data and Strategy Management

Streaming real-time options data with ThetaData

Getting ready

How to do it...

How it works...

There's more...

See also

Using the ArcticDB DataFrame database for tick storage

Getting ready

How to do it...

How it works...

There's more...

See also

Triggering real-time risk limit alerts

Getting ready

How to do it...

How it works...

There's more...

See also

Storing trade execution details in a SQL database

Getting ready

How to do it...

How it works...

There's more...

[**See also**](#)

[**Index**](#)

[**Other Books You May Enjoy**](#)

Preface

Algorithmic trading is the art of using statistical models, programming, and math to trade financial assets. With the vast volumes of data available in today's markets, it is essential to have powerful tools at your disposal to remain competitive. In *Python for Algorithmic Trading Cookbook*, you'll get code to design, backtest, and deploy your own algorithmic trading strategies with Python. Python is not only accessible and easy to learn, but also boasts thousands of powerful libraries that can help you implement sophisticated trading strategies.

Many resources available today cover basic trading strategies largely focused on technical analysis. Unfortunately, these strategies often fail over the long run. This book aims to bring professional techniques and tools to non-professionals through small, digestible recipes. These recipes will not only guide you in creating, testing, and deploying algorithmic trading strategies, but build a strong foundation in the tools and techniques to prepare for creating, testing, and deploying algorithmic trading strategies.

In this book, I will share insights and methodologies drawn from my 20+ years of experience in algorithmic trading and Python programming. I have taught the techniques presented in this book to over 1,000+ students in my course, *Getting Started with Python for Quant Finance*. The course has been praised for its practical application of Python to algorithmic trading. Whether you are a seasoned programmer looking to expand into trading or a trader aiming to enhance your technical skills, this book is built to provide you with the tools and knowledge you need.

With Python, you can easily access financial data, perform complex calculations, and backtest your models efficiently. This book will guide you through the entire process, from setting up your Python environment to implementing advanced trading algorithms.

By the end of this journey, you will have a robust toolkit to develop and refine your own trading strategies. As the financial markets evolve, so too must our approaches. With the right knowledge and tools, you can navigate these changes and find opportunities that others might miss.

Who this book is for

Traders, investors, and Python developers can gain practical insights into designing, backtesting, and deploying algorithmic trading strategies from this book. The three main personas who are the target audience of this content are as follows:

Active traders and investors: Individuals who are already investing in the stock market and want to leverage algorithmic strategies to enhance their trading performance. They will learn to use Python to develop, test, and implement advanced trading models, including acquiring and processing freely available market data with OpenBB and building a research environment populated with financial market data.

Python developers with market interest: Developers with a solid understanding of Python data structures and libraries, such as pandas, who are looking to apply their programming skills to the financial markets. This book will help them bridge the gap between coding and trading by providing practical recipes and techniques used in algorithmic trading. They will learn to identify alpha factors, engineer them into signals, and use VectorBT for walk-forward optimization to find strategy parameters.

Aspiring algorithmic traders: For those who aspire to enter the field of algorithmic trading and have basic experience in Python programming, this book will provide them with the foundational knowledge and tools to start designing and deploying their trading strategies. They will learn to build production-ready backtests with Zipline, evaluate factor performance, set up the code framework to connect and send orders to Interactive Brokers, and deploy trading strategies to a live trading environment using the IB API.

This book will equip you with the skills to acquire and analyze financial data and build and refine algorithmic trading strategies using Python. Whether you are an experienced market participant looking to enhance your technical capabilities or a Python programmer with a keen interest in the financial markets, this book provides actionable insights and techniques to succeed in algorithmic trading.

What this book covers

[**Chapter 1**](#), *Acquire Free Financial Market Data with Cutting-edge Python Libraries*, provides an in-depth exploration of acquiring various types of financial market data. It covers continuous futures data, S&P 500 ratios, and short volume using Nasdaq Data Link. You will learn to work with stock market, historic futures, and options market data using the OpenBB Platform, and harness factor data using pandas-datareader.

[**Chapter 2**](#), *Analyze and Transform Financial Market Data with pandas*, dives into the powerful pandas library for data manipulation. This chapter explains pandas index types, building series and DataFrames, and transforming data. You will learn to calculate asset returns, measure volatility, generate cumulative returns, resample data, address missing data issues, and apply custom functions to analyze time series data.

[**Chapter 3**](#), *Visualize Financial Market Data with Matplotlib, Seaborn, and Plotly Dash*, covers techniques for visualizing financial data. You will quickly visualize data using pandas, animate yield curve evolution with Matplotlib, plot options implied volatility surfaces, visualize statistical relationships with Seaborn, and create an interactive PCA analytics dashboard with Plotly Dash.

[**Chapter 4**](#), *Store Financial Market Data on Your Computer*, discusses methods for efficiently storing financial data. You will learn to store data in CSV format, SQLite, a networked Postgres database, and the ultra-fast HDF5 format, ensuring your data is easily accessible and well organized for analysis and backtesting.

[**Chapter 5**](#), *Build Alpha Factors for Stock Portfolios*, focuses on creating alpha factors. It covers identifying latent return drivers with principal component analysis, hedging portfolio beta using linear regression, analyzing portfolio sensitivities to Fama-French factors, assessing market inefficiency based on volatility, and preparing a factor ranking model using Zipline pipelines.

[**Chapter 6**](#), *Vector-Based Backtesting with VectorBT*, introduces vector-based backtesting. This chapter guides you through experimenting with millions of strategy combinations, conducting walk-forward optimization, and implementing a risk parity backtest using VectorBT, providing a robust framework for strategy evaluation.

[**Chapter 7**](#), *Event-Based Backtesting Factor Portfolios with Zipline Reloaded*, explores event-based backtesting. You will backtest a momentum factor strategy and explore a mean reversion strategy using Zipline Reloaded, helping you understand the dynamics and performance of various trading strategies.

[Chapter 8, Evaluate Factor Risk and Performance with Alphalens Reloaded](#), examines factor risk and performance. You will prepare backtest results, evaluate the information coefficient, examine factor return performance, and evaluate factor turnover, ensuring a comprehensive analysis of your trading strategies.

[Chapter 9, Assess Backtest Risk and Performance Metrics with Pyfolio](#), covers risk and performance assessment. This chapter explains preparing Zipline backtest results for pyfolio, generating strategy performance analytics, building a drawdown and rolling risk analysis, analyzing strategy holdings, leverage, exposure, sector allocations, and breaking down performance to the trade level.

[Chapter 10, Set Up the Interactive Brokers Python API](#), provides a guide to building an algorithmic trading app. You will create contract and order objects with the IB API, fetch historical market data, get market data snapshots, stream live tick data, and store live tick data in a local SQL database, enabling real-time trading and data management.

[Chapter 11, Manage Orders, Positions, and Portfolios with the IB API](#), explains managing trades and portfolios. You will learn to execute orders, manage placed orders, get portfolio details, inspect positions, and compute portfolio profit and loss, providing comprehensive tools to manage your trading operations.

[Chapter 12, Deploy Strategies to a Live Environment](#), focuses on live trading strategy deployment. This chapter covers calculating real-time performance and risk indicators, sending orders based on portfolio targets, and deploying monthly factor, options combo, and intraday multi-asset mean reversion strategies, ensuring your strategies are effective and responsive in live markets.

[Chapter 13, Advanced Recipes for Market Data and Strategy Management](#), offers advanced techniques for managing market data and strategies. You will learn to stream real-time options data with ThetaData, use the ArcticDB DataFrame database for tick storage, trigger real-time risk limit alerts, and store trade execution details in a SQL database, enhancing your data management and strategy implementation capabilities.

To get the most out of this book

You will need Python version 3.10 to work with the examples in this book. Ideally, you will install Python using the Anaconda distribution. All code examples have been tested using Python version 3.10 on macOS with an M2 chip.

Software/hardware covered in the book	Operating system requirements
Python version 3.10	Windows, macOS, or Linux

pandas version 2+	
OpenBB Platform version 4+	
PostgreSQL	

Where necessary, specific installation instructions are included in the given chapters.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book’s GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at

<https://github.com/PacktPublishing/Algorithmic-Trading-with-Python-Cookbook>. If there’s an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at

<https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example:
“`save_data_range` uses the pandas `to_csv` method to store the data to disk using GZIP compression in the CSV format.”

A block of code is set as follows:

```
import datetime as dt
import pandas as pd
from openbb_terminal.sdk import openbb
```

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

```
pip install thetadata
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “In the **Actions** pane, click on **Create Basic Task**.”

TIPS OR IMPORTANT NOTES

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Python for Algorithmic Trading Cookbook*, we'd love to hear your thoughts! Please [click here to go straight to the Amazon review page for this book and share your feedback](#).

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781835084700>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Acquire Free Financial Market Data with Cutting-Edge Python Libraries

A May 2017 Economist cover declared data to be the world’s most valuable resource. It’s none truer than in algorithmic trading. As algorithmic traders, it’s our job to acquire and make sense of billions of rows of market data for use in trading algorithms. In this context, it’s crucial to gather high-quality, reliable data that can adequately support trading algorithms and market research. Luckily for us, it’s possible to acquire high-quality data for free (or nearly free).

This chapter offers recipes for a series of different Python libraries—including the cutting-edge OpenBB Platform—to acquire free financial market data using Python. One of the primary challenges most non-professional traders face is getting all the data required for analysis together in one place. The OpenBB Platform addresses this issue. We’ll dive into acquiring data for a variety of assets, including stocks, options, futures (both continuous and individual contracts), and Fama-French factors.

One crucial point to remember is that data can vary across different sources. For instance, prices from two sources might differ due to distinct data sourcing methods or different adjustment methods for corporate actions. Some of the libraries we’ll cover might download data for the same asset from the same source. However, libraries vary in how they return that data based on options that help you preprocess the data in preparation for research.

Lastly, while we’ll focus heavily on mainstream financial data in this chapter, financial data is not limited to prices. The concept of “alternative data,” which includes non-traditional data sources such as satellite images, web traffic data, or customer reviews, can be an important source of information for developing trading strategies. The Python tools to acquire and process this type of data are outside the scope of this book. We’ve intentionally left out the methods of acquiring and processing this type of data since it’s covered in other resources dedicated to the topic.

In this chapter, we’ll cover the following recipes:

- Diving into continuous futures data with Nasdaq Data Link
- Exploring S&P 500 ratios data with Nasdaq Data Link
- Working with stock market data with the OpenBB Platform
- Fetching historic futures data with the OpenBB Platform
- Navigating options market data with the OpenBB Platform

- Harnessing factor data using `pandas_datareader`

Technical requirements

This book relies on the Anaconda distribution of Python. We'll use Jupyter Notebook and Python script files to write the code. Unless specified otherwise, all the code can be written in Jupyter Notebooks.

Download and install the Anaconda distribution of Python. You can do this by going to <https://www.anaconda.com/download>. Depending on your operating system, the instructions for downloading and installing will vary. Please refer to the Anaconda documentation for detailed instructions.

Anaconda ships with a package manager called `conda`. Package managers make it easy to install, remove, and update Python packages. There's a great cheat sheet for the `conda` package manager that you can download from

https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf.

Once you've installed the Anaconda distribution, open your Terminal on Mac or Linux or the Anaconda Prompt on Windows. If you're a Windows user, make sure to use the Command Prompt instead of the Powershell prompt. Then follow these steps:

1. Update the `conda` package manager:

```
conda update -n base conda -y
```

2. Create a virtual environment:

```
conda create -n my-quant-stack python=3.10 -y
```

3. After the installation process is complete, activate the environment:

```
conda activate my-quant-stack
```

4. Install Jupyter Notebook using the package manager that ships with Python, `pip`:

```
pip install notebook matplotlib
```

This will set up a virtual environment using Python 3.10 and install Jupyter Notebook.

This chapter will use three Python libraries to acquire financial market data: the Nasdaq Data Link client, the OpenBB Platform, and `pandas_datareader`. The good news is that installing the OpenBB Platform installs many of the libraries you will need to acquire financial market data, including the Nasdaq Data Link client and `pandas_datareader`. As such, there is no need to install the libraries separately.

Install the OpenBB Platform with all extensions and providers (both officially supported and community-maintained ones) using `pip`:

```
pip install openbb[all]
```

This is the easiest way to set up the OpenBB Platform for this book.

IMPORTANT NOTE

In a macOS zsh Terminal shell, add quotation marks around the library name: "openbb[all]"

To install a single extension:

```
pip install openbb[charting]
pip install openbb[ta]
```

Or install a single provider:

```
pip install openbb[yfinance]
```

To install the Nightly distribution (this installs all extras by default):

```
pip install openbb-nightly
```

IMPORTANT NOTE

At the time of writing, installing the OpenBB Platform using `pip` isn't compatible with environments such as Google Colab and Kaggle since they come with preinstalled packages that can conflict with the ones used with the OpenBB Platform. If you run into trouble installing the OpenBB Platform, please check the online documentation for the most up-to-date instructions.

Diving into continuous futures data with Nasdaq Data Link

Futures are one of the most common derivatives for trading and hedging purposes. Commercial traders such as food companies and energy traders use futures to hedge against the risk of price changes in the physical commodity. Hedge funds and high-frequency traders take advantage of minute price discrepancies, adding liquidity to the market, and lowering the cost of trading.

Speculators like us use futures to express our views on market direction in a risk-defined way. Futures have a defined expiration date, after which point the contract stops trading. Market dynamics and trading strategies shift traders' attention from one contract to the next through time. While the flexibility of futures is great for trading, it makes it hard to acquire data for research and backtesting.

The most common way to get started analyzing futures contracts is through continuous futures data. As the name implies, continuous futures provide a continuously adjusted data feed that takes into consideration the **roll** from one expiration to the next. Continuous futures data lets traders and

analysts backtest long-term trading strategies. Since futures expire, traders buy or sell positions in the expiring contract to enter a similar position in the next expiring contract. It's this process of contract expiration that makes continuous futures data important for long-term analysis. The downside of continuous futures data is that the impact of the roll over long time frames can be significant, so it needs to be taken into consideration in strategies that rely on different roll strategies.

With Nasdaq Data Link, previously known as Quandl, you can access continuous contract data for 600 futures contracts built on top of raw data from CME, ICE, LIFFE, and other exchanges. This recipe will guide you through the process of obtaining continuous futures data using the Nasdaq Data Link client.

Getting ready...

You can use the Nasdaq Data Link client to acquire data without an account or an API key, but you're limited in the number of API calls you can make. Without a key, you can make 20 API calls in 10 minutes or up to 50 API calls per day.

Since an account is free, it makes sense to sign up. This is something you can do via the Nasdaq Data Link website (<https://data.nasdaq.com/>). Your API key is in your profile (<https://data.nasdaq.com/account/profile>). Also, make sure to install `nasdaqdatalink` with `pip` if it's not installed already:

```
pip install nasdaq-data-link
```

How to do it...

The Nasdaq Data Link client is easy to use:

1. Import the necessary library:

```
import nasdaqdatalink
```

2. Set your API key:

```
nasdaqdatalink.ApiConfig.api_key = "your_api_key"
```

3. Download the continuous futures data:

```
data = nasdaqdatalink.get("CHRIS/CME_ES1")
```

4. Inspect the downloaded data:

```
print(data)
```

Running the preceding code generates a pandas DataFrame and prints the data to the screen:

Date	Open	High	Low	Last	Change	Settle	Volume	Previous Day Open	Interest
1997-09-09	934.00	942.00	933.00	934.00	NaN	934.00	7034.0		1109.0
1997-09-10	934.00	935.00	915.00	915.00	NaN	915.00	11387.0		2325.0
1997-09-11	916.00	918.00	900.00	908.00	NaN	908.00	2523.0		2549.0
1997-09-12	908.00	926.00	904.00	924.00	NaN	924.00	928.0		2163.0
1997-09-15	925.00	930.00	920.00	922.00	NaN	922.00	208.0		2107.0
...
2021-06-14	4248.00	4257.50	4233.50	4255.00	9.00	4254.75	1089658.0		1833460.0
2021-06-15	4255.25	4267.50	4238.00	4247.25	-8.25	4246.50	944668.0		1265301.0
2021-06-16	4248.00	4251.25	4200.75	4212.75	-23.50	4223.00	482760.0		744020.0
2021-06-17	4214.25	4232.50	4193.00	4225.75	-0.75	4222.25	305144.0		590729.0
2021-06-18	4225.75	4231.00	4183.75	4185.50	NaN	4187.25	35117.0		539808.0

Figure 1.1: Historic settlement prices of the ES continuous futures contract

How it works...

The `nasdaqdatalink` library provides a simple way to download data from Nasdaq Data Link. The format of the symbol is `CHRIS/{EXCHANGE}_{CODE}{NUMBER}`, where `{EXCHANGE}` is the exchange the contract trades on (`CME`, for example), `{CODE}`, is the contract code (`CL`, for example), and `{NUMBER}` is the “depth” associated with the chained contract. For instance, the front-month contract has a depth of 1, the second-month contract has a depth of 2, and so on. In the preceding example, we are retrieving the continuous contract data for the E-Mini S&P 500 Futures contract from the **Chicago Mercantile Exchange (CME)**. The `get` function downloads the data as a pandas DataFrame.

There's more...

The `nasdaqdatalink` library provides different options for downloading most data feeds:

1. Instead of a pandas DataFrame, return a NumPy array:

```
data = nasdaqdatalink.get("CHRIS/CME_ES1", returns="numpy")
```

2. Define a start and end date for the data series:

```
data = nasdaqdatalink.get(
    "CHRIS/CME_ES1",
    start_date="2001-12-31",
    end_date="2005-12-31"
)
```

3. You can request specific columns by adding a dot and the column number to the symbol:

```
data = nasdaqdatalink.get(  
    "CHRIS/CME_ES1.8",  
    start_date="2001-12-31",  
    end_date="2005-12-31"  
)
```

4. We'll cover resampling in detail in [Chapter 2, Analyze and Transform Financial Market Data with pandas](#), but the **nasdaqdatalink** library can resample data to a monthly interval:

```
data = nasdaqdatalink.get(  
    "CHRIS/CME_ES1",  
    collapse="monthly"  
)
```

5. Finally, you can request multiple symbols at once. In this example, we're creating a futures curve using four sequential futures contracts and plotting the curve on the last date:

```
contracts = [  
    "CHRIS/CME_ES1.6",  
    "CHRIS/CME_ES2.6",  
    "CHRIS/CME_ES3.6",  
    "CHRIS/CME_ES4.6",  
]  
data = nasdaqdatalink.get(  
    contracts,  
    start_date="2015-01-01",  
    end_date="2015-12-31"  
)  
data.iloc[0].plot(title=f"ES on {data.index[0]}");
```

By running the preceding code, a plot is generated showing the settlement prices across expirations:

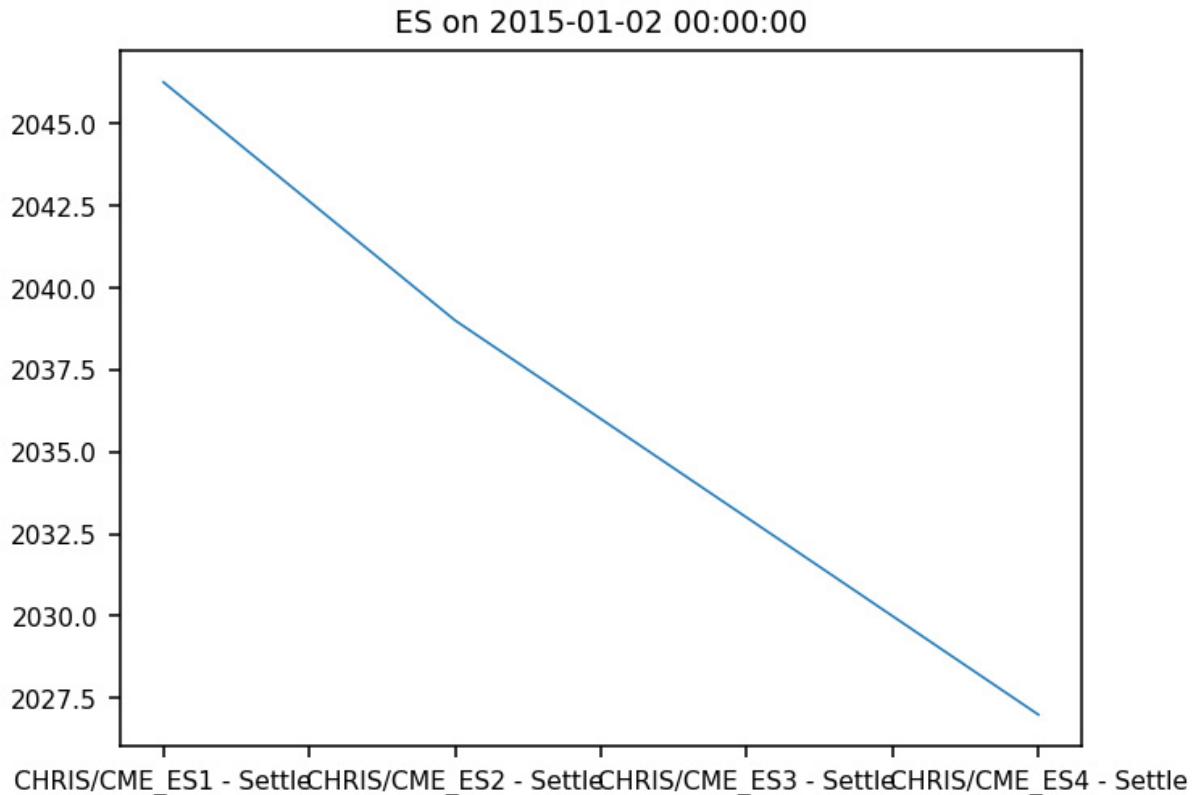


Figure 1.2: The first four ES continuous contracts on 2015-01-02

See also

A great way to get free resources for using Python for market data analysis is the free, twice-weekly newsletter from PyQuant News. You can sign up here: <https://www.pyquantnews.com/subscribe-to-the-pyquant-newsletter>

To dive deeper into Nasdaq Data Link data, check out the following resources:

- Documentation on acquiring time series data using the Nasdaq Data Link library: <https://docs.data.nasdaq.com/docs/python-time-series>
- Documentation on the web service API calls that are used under the hood of the Python client: <https://docs.data.nasdaq.com/docs/time-series>
- A list of futures contracts is available for download: <https://static.quandl.com/Ticker+CSV%27s/Futures/continuous.csv>
- For more information on processing alternative data, see the advanced book *Machine Learning for Algorithmic Trading*, by Stefan Jansen

Exploring S&P 500 ratios data with Nasdaq Data Link

The performance of professional investment managers that actively trade is normally compared against a benchmark. A common benchmark for comparison is the S&P 500 index of US stocks. Non-professional traders can also use the S&P 500 as a benchmark for risk and performance analysis and to hedge systematic risk from portfolios. Getting compiled data for the underlying stocks that make up the S&P 500—such as the price-to-earnings ratio—has historically been difficult. These ratios are now available for download through the Nasdaq Data Link Python client. This recipe will guide you through how to use the library to download and analyze S&P 500 ratios.

How to do it...

If you're extending the code from the previous recipe, then your API is already set. If you're starting from scratch, follow the first two steps:

1. Import the necessary library:

```
import nasdaqdatalink
```

2. Set your API key:

```
nasdaqdatalink.ApiConfig.api_key = "your_api_key"
```

3. Fetch the monthly dividend yield for the S&P 500:

```
data = nasdaqdatalink.get("MULTPL/SP500_DIV_YIELD_MONTH")
```

4. Plot the dividend yield through time:

```
data.plot(title="S&P 500 dividend yield (12 month dividend per share)/price")
```

Running the preceding code generates a plot of the S&P 500 dividend yield through time:

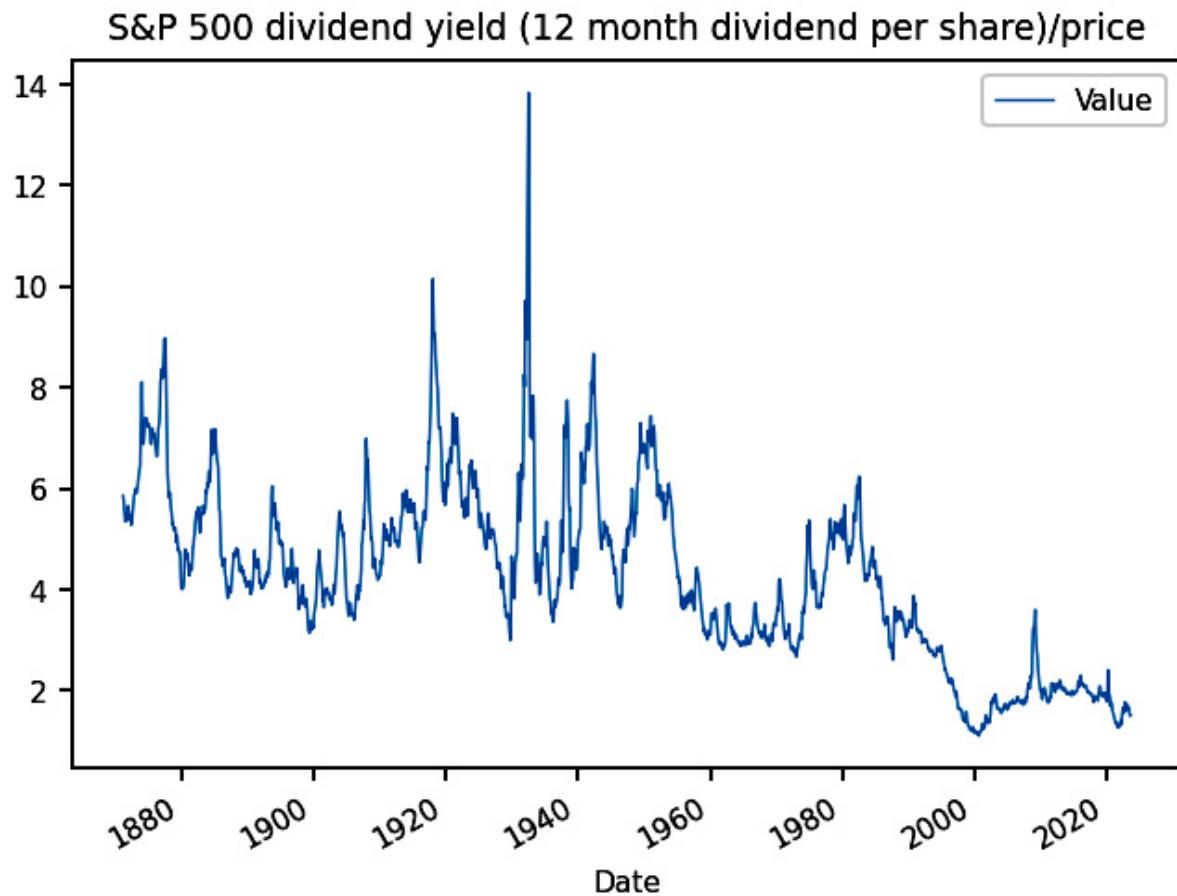


Figure 1.3: S&P 500 dividend yield from index inception

How it works...

Similar to the first recipe, we import the Nasdaq Data Link client, set the API key, and use the same `get` method. The format of the symbol is `MULTPL/{RATIO}`, where `{RATIO}` is the name of the ratio to download. The result is a pandas DataFrame unless you set additional options (see the previous recipe).

There's more...

There are 36 S&P 500 ratios available with up-to-date data. Some of the most popular are as follows:

- `SP500_PE_RATIO_MONTH`: S&P 500 price-to-earnings ratio
- `SHILLER_PE_RATIO_MONTH`: S&P 500 Shiller PE ratio. This is also known as the **Cyclically Adjusted PE ratio (CAPE ratio)**.

- **SP500_EARNINGS_YIELD_MONTH**: Earnings yield of the S&P 500 based on the trailing 12-month earnings divided by index price. The current yield is estimated from the latest reported earnings and current index value.
- **SP500_REAL_PRICE_MONTH**: S&P 500 index not adjusted for inflation.

See also

You can find a complete list of the ratios that are available for download at
<https://data.nasdaq.com/data/MULTPL-sp-500-ratios>.

Working with stock market data with the OpenBB Platform

You may remember the **meme stock** hysteria that sent GameStop's stock up 1,744% in January 2021. One of the good things that came from that episode was the GameStonk terminal, now rebranded as OpenBB. OpenBB is the most popular open-source finance projects on GitHub for good reason: it provides a single interface to access hundreds of data feeds from one place in a standard way. OpenBB has a command-line interface that is great for manual investment research. But when it's time to get data into Python, you want the OpenBB Platform. This recipe will guide you through the process of using the OpenBB Platform to fetch stock market data.

Getting ready...

By now, you should have the OpenBB Platform installed in your virtual environment. If not, go back to the beginning of this chapter and get it set up. The OpenBB Platform is free to use and offers a web-based UI to manage your configuration files, store API keys, and get code walkthroughs and examples. Sign up for a free Hub account at <https://my.openbb.co/login>. The popular course, *Getting Started with Python for Quant Finance*, uses OpenBB exclusively for all the code. Check out <https://www.pyquantnews.com/getting-started-with-python-for-quant-finance> for information on how to join.

How to do it...

Using the OpenBB Platform involves one import:

1. Import the OpenBB Platform:

```
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Use the `historical` method to download price data for the SPY ETF:

```
data = obb.equity.price.historical("SPY", provider="yfinance")
```

3. Inspect the resulting DataFrame:

```
print(data)
```

Running the preceding code generates a pandas DataFrame and prints the data to the screen:

	open	high	low	close	volume	split_ratio	dividend	capital_gains
date								
2023-05-26	415.329987	420.769989	415.250000	420.019989	93830000	0.0	0.0	0.0
2023-05-30	422.029999	422.579987	418.739990	420.179993	72216000	0.0	0.0	0.0
2023-05-31	418.279999	419.220001	416.220001	417.850006	110811800	0.0	0.0	0.0
2023-06-01	418.089996	422.920013	416.790009	421.820007	88865000	0.0	0.0	0.0
2023-06-02	424.500000	428.739990	423.950012	427.920013	91366700	0.0	0.0	0.0
...
2024-05-20	529.570007	531.559998	529.169983	530.059998	37764200	0.0	0.0	0.0
2024-05-21	529.280029	531.520020	529.070007	531.359985	33437000	0.0	0.0	0.0
2024-05-22	530.650024	531.380005	527.599976	529.830017	48390000	0.0	0.0	0.0
2024-05-23	532.960022	533.070007	524.719971	525.960022	57211200	0.0	0.0	0.0
2024-05-24	527.849976	530.270020	526.880005	529.440002	41258400	0.0	0.0	0.0

Figure 1.4: Historic price data for SPY

How it works...

The OpenBB Platform follows an easy-to-understand namespace convention. All the methods for acquiring stock price data are methods on `openbb.equity`.

The `historical` method accepts a ticker symbol and returns the open, high, low, close, adjusted close, volume, dividend, and split adjustments in a pandas DataFrame. The additional parameters you can specify are as follows:

- `start_date`: Start date to get data from with
- `interval`: Interval (in minutes) to get data—that is, 1, 5, 15, 30, 60, or 1,440
- `end_date`: End date to get data from with
- `provider`: Source of data extracted

There's more...

An important benefit of using the OpenBB Platform is choosing your data source. By default, the OpenBB Platform will attempt to download data from free sources such as Yahoo! Finance. In most OpenBB Platform calls, you can indicate a different source. To use a source that requires an API key (either free or paid), you can configure it in the OpenBB Hub.

TIP

Check out the OpenBB Platform documentation for the latest functionality: <https://docs.openbb.co>.

Let's look at some more of the functions of the OpenBB Platform.

Comparison of fundamental data

Not only can the OpenBB Platform download fundamental data in an organized and usable way, but it can also concatenate it in a single Pandas DataFrame for further analysis.

We can use the following code to see the balance sheet metrics from **AAPL** and **MSFT**:

```
obb.equity.fundamental.metrics(  
    "AAPL,MSFT",  
    provider="yfinance"  
)
```

The output of the preceding snippet is a pandas DataFrame with fundamental data for each ticker that was passed:

	0	1
symbol	AAPL	MSFT
market_cap	2913325809664.0	3198271619072.0
pe_ratio	29.593458	37.32177
forward_pe	26.27801	32.452488
peg_ratio	2.75	2.25
peg_ratio_ttm	2.2528	2.1289
enterprise_to_ebitda	22.762	25.749
earnings_growth	0.007	0.2
earnings_growth_quarterly	-0.022	0.199
revenue_per_share	24.537	31.834
revenue_growth	-0.043	0.17
enterprise_to_revenue	7.732	13.624
quick_ratio	0.875	1.132
current_ratio	1.037	1.242
debt_to_equity	140.968	41.963
gross_margin	0.45586	0.69894
operating_margin	0.30743	0.44588
ebitda_margin	0.33968	0.52912
profit_margin	0.26306	0.36427
return_on_assets	0.22074	0.15295
return_on_equity	1.4725	0.38488
dividend_yield	0.0053	0.007
dividend_yield_5y_avg	0.0073	0.0094
payout_ratio	0.1493	0.2478
book_value	4.837	34.058
price_to_book	39.27848	12.634917
enterprise_value	2950608977920	3223296671744
overall_risk	1.0	1.0
audit_risk	6.0	3.0
board_risk	1.0	5.0
compensation_risk	2.0	2.0
shareholder_rights_risk	1.0	2.0
beta	1.264	0.893
price_return_1y	0.071517	0.298753
currency	USD	USD

Figure 1.5: Balance sheet data for MSFT and AAPL

Building stock screeners

One of the most powerful features of the OpenBB Platform is the custom stock screener. It uses the **Finviz** stock screener under the hood and surfaces metrics across a range of stocks based on either pre-built or custom criteria. See the documentation for more on how to use the OpenBB screener functions (<https://docs.openbb.co/platform/reference/equity/screener>):

1. Create an overview screener based on a list of stocks using the default view:

```
obb.equity.compare.groups(
    group="industry",
    metric="valuation",
    provider="finviz"
)
```

The output of the preceding snippet is the following pandas DataFrame:

	name	market_cap	performance_1D	pe	forward_pe	...	volume	price_to_sales	price_to_book	price_to_cash	price_to_free_cash_flow
0	Pharmaceutical Retailers	13580000000	-0.0394	NaN	4.99	...	20580000	0.09	0.99	17.30	829.95
1	Airlines	122270000000	-0.0182	10.97	6.83	...	70720000	0.47	2.22	2.31	43.65
2	REIT - Mortgage	53670000000	-0.0068	15.94	6.98	...	40160000	1.69	0.81	3.60	4.73
3	Insurance - Reinsurance	53050000000	-0.0063	7.66	7.53	...	20499999	0.90	1.21	NaN	3.03
4	Insurance - Life	269430000000	-0.0107	12.25	8.11	...	16079999	1.10	1.48	4120.41	4.41
...
140	REIT - Healthcare Facilities	123940000000	-0.0028	94.22	46.31	...	27280000	5.51	1.68	25.46	22.11
141	Uranium	36590000000	0.0248	112.12	46.58	...	36490000	15.51	4.90	28.10	143.86
142	Shell Companies	26350000000	0.0185	44.08	133.74	...	10560000	46.71	2.10	36.15	105.00
143	Infrastructure Operations	34110000000	-0.0041	57.81	146.15	...	1160000	3.40	7.22	6.25	28.37
144	Real Estate - Diversified	6760000000	-0.0141	75.91	NaN	...	549690	4.54	1.74	6.63	58.78

Figure 1.6: Results of a comparison screener between F, GE, and TSLA

2. Create a screener that returns the top gainers from the technology sector based on a preset:

```
obb.equity.compare.groups(
    group="technology",
    metric="performance",
    provider="finviz"
)
```

The output of the preceding snippet is the following pandas DataFrame:

	name	performance_1D	performance_1W	performance_1M	performance_3M	...	performance_YTD	analyst_recommendation	volume	volume_average	volume_relative
0	Pharmaceutical Retailers	-0.0394	-0.1342	-0.1270	-0.2850	...	-0.4065	3.05	20600000	13170000	1.56
1	Gambling	-0.0602	-0.0940	-0.0237	-0.0917	...	0.0263	1.49	55830000	18370000	3.04
2	Tools & Accessories	-0.0169	-0.0683	-0.0322	-0.0660	...	-0.0766	2.59	5910000	5070000	1.16
3	Airports & Air Services	-0.0213	-0.0680	-0.0299	0.0950	...	-0.0233	2.17	5530000	7110000	0.78
4	Utilities - Regulated Water	-0.0171	-0.0601	0.0082	0.0268	...	-0.0489	1.91	5450000	5980000	0.91
...
140	Department Stores	0.0098	0.0397	0.0640	0.0028	...	0.0385	3.24	12020000	16200000	0.74
141	Utilities - Renewable	-0.0010	0.0584	0.1817	0.3068	...	0.3735	1.79	49110000	26700000	1.84
142	Utilities - Independent Power Producers	0.0190	0.1012	0.3138	0.7382	...	0.9717	1.73	12450000	13270000	0.94
143	Semiconductors	0.0380	0.1032	0.1944	0.2518	...	0.5913	1.54	308940000	320980000	0.96
144	Solar	0.0119	0.2505	0.2811	0.2057	...	0.0411	1.85	799280000	155900000	5.13

Figure 1.7: Results of a screener showing the day's top-gaining stocks

3. Create a screener that presents an overview grouped by sector:

```
obb.equity.compare.groups(  
    group="sector",  
    metric="overview",  
    provider="finviz"  
)
```

The output of the preceding snippet is the following pandas DataFrame:

	name	stocks	market_cap	performance_1D	dividend_yield	pe	forward_pe	peg	float_short	volume
0	Gambling	18	78550000000	-0.0602	0.0038	88.00	20.37	3.28	0.0257	55830000
1	Pharmaceutical Retailers	8	13580000000	-0.0394	0.0979	NaN	4.99	NaN	0.0634	20600000
2	Staffing & Employment Services	23	17212000000	-0.0224	0.0230	26.15	22.45	2.46	0.0245	9700000
3	Lumber & Wood Production	6	21290000000	-0.0222	0.0102	25.55	13.97	1.53	0.0249	846080
4	Airports & Air Services	9	29420000000	-0.0213	0.0338	17.29	15.19	4.46	0.1007	5530000
...
140	Other Precious Metals & Mining	18	18040000000	0.0320	0.0052	64.77	23.81	3.51	0.0249	32460000
141	Copper	8	18065000000	0.0338	0.0185	42.89	25.63	1.75	0.0183	27360000
142	Oil & Gas Drilling	11	29580000000	0.0371	0.0190	18.99	9.75	0.37	0.1092	30250000
143	Semiconductors	67	6048100000000	0.0380	0.0065	49.71	26.46	1.63	0.0202	308940000
144	Silver	3	4370000000	0.0488	0.0023	46.11	24.66	1.28	0.0436	10710000

Figure 1.8: Results of a screener grouped by sector

See also

For more on OpenBB and the **Finviz** stock screener, check out the following resources:

- The OpenBB Platform documentation, which contains details on dozens of functions to acquire stock market data for free: <https://docs.openbb.co/platform/reference/equity>
- The **Finviz** home page, which includes free access to the web-based screener <https://finviz.com/?a=2548677>
- The top-selling cohort-based course to help beginners get up and running with Python for quant finance, algorithmic trading, and data analysis: <https://www.pyquantnews.com/getting-started-with-python-for-quant-finance>

Fetching historic futures data with the OpenBB Platform

Traders use continuous futures data for backtesting trading strategies. Futures traders use the roll from one contract to another as a potential opportunity for profit. Some traders simply pick a date before expiration to roll to the next contract, while others use sophisticated techniques involving open interest. This **basis trade** is persistently one of the most popular trading strategies for futures traders. These traders want control over the data that's used to compute the basis trade, so acquiring

individual contract data is important. This recipe will guide you through the process of using the OpenBB Platform to fetch individual futures contract data.

Getting ready...

By now, you should have the OpenBB Platform installed in your virtual environment. If not, go back to the beginning of this chapter and get it set up.

How to do it...

We'll use the futures functionality in the OpenBB Platform to download individual futures data for free:

1. Import pandas and the OpenBB Platform:

```
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download the current futures curve for the VIX futures contract from the **Chicago Board Options Exchange (CBOE)**:

```
data = obb.derivatives.futures.curve(symbol="VX")
```

3. Inspect the resulting DataFrame:

```
print(data)
```

Running the preceding code generates the futures curve for the VIX futures contract:

	expiration	price	symbol
0	2024-05-29	13.1250	VX22/K4
1	2024-06-05	13.2750	VX23/M4
2	2024-06-12	13.4000	VX24/M4
3	2024-06-18	13.7256	VX/M4
4	2024-06-26	14.4000	VX26/M4
...
9	2024-10-16	17.9105	VX/V4
10	2024-11-20	16.9675	VX/X4
11	2024-12-18	17.0162	VX/Z4
12	2025-01-22	17.6250	VX/F5
13	2025-02-19	17.9750	VX/G5

Figure 1.9: Settlement prices for the forward Eurodollar futures contracts

4. Update the DataFrame index to the expiration dates and plot the settlement prices:

```
data.index = pd.to_datetime(data.expiration)
data.plot()
```

By running the proceeding code, we plot the VIX futures curve:

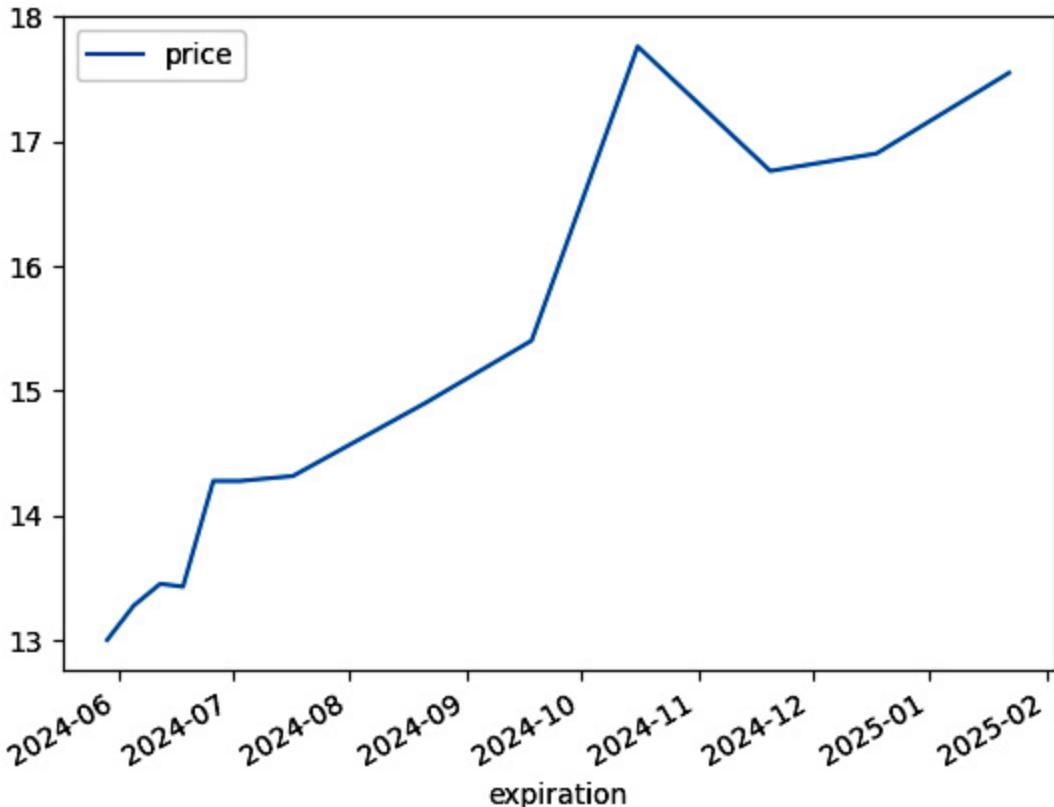


Figure 1.10: VIX futures curve

There's more...

You can use the `obb.derivatives.futures.historical` method to get historical data for an individual expiration. Stitching together data across a range of years can provide insight into the market's expectation of supply and demand of the underlying commodity:

1. First, create a list containing the year and month expirations you're interested in:

```
expirations = [
    "2024-12",
    "2025-12",
    "2026-12",
    "2027-12",
    "2028-12",
    "2029-12",
    "2030-12",
]
```

2. The preceding code creates a Python list of expiration years and dates in string format. Now, loop through each of the expirations to download the data:

```
contracts = []
for expiration in expirations:
    df = (
```

```

    obb
    .derivatives
    .futures
    .historical(
        symbol="CL",
        expiration=expiration,
        start_date="2020-01-01",
        end_date="2022-12-31"
    )
    ).rename(columns={
        "close": expiration
    })
    contracts.append(df[expiration])
)

```

3. For each of the contracts, use the OpenBB Platform to download historical futures data for the CL contract between January 1, 2020, and 31 December 31, 2022. Using the pandas `rename` method, change the column name from "`close`" to the expiration date. Finally, append the newly created pandas DataFrame to a list of DataFrames:

```

historical = (
    pd
    .DataFrame(contracts)
    .transpose()
    .dropna()
)

```

4. Concatenate the DataFrames together, swap the columns and rows using the `transpose` method, and drop any records with no data using the `dropna` method. Inspect the resulting DataFrame:

```
print(historical)
```

By printing the DataFrame, we will see the historical settlement prices:

	2024-12	2025-12	2026-12	2027-12	2028-12	2029-12	2030-12
date							
2020-01-24	50.139999	50.700001	51.560001	51.630001	51.630001	51.630001	51.630001
2020-01-27	50.599998	51.180000	51.049999	51.119999	51.119999	51.119999	51.119999
2020-01-28	50.779999	51.230000	51.549999	51.619999	51.619999	51.619999	51.619999
2020-01-29	50.639999	51.130001	51.599998	51.669998	51.669998	51.669998	51.669998
2020-01-30	50.910000	51.439999	51.490002	51.560001	51.560001	51.560001	51.560001
...
2024-05-20	76.870003	72.300003	69.169998	67.000000	65.430000	64.349998	63.619999
2024-05-21	76.330002	72.010002	69.019997	66.940002	65.470001	64.500000	63.770000
2024-05-22	75.160004	70.980003	68.120003	66.160004	64.769997	63.810001	63.130001
2024-05-23	74.570000	70.589996	67.839996	65.959999	64.660004	63.750000	63.070000
2024-05-24	75.089996	70.849998	67.959999	65.980003	64.629997	63.720001	63.090000

Figure 1.11: Historic settlement prices for the December CL futures contract

The result is the historical data between January 2020 and December 2022 for each of the December expirations between 2023 and 2030:

5. To visualize the market's expectation of the future supply and demand of the December contract, you can plot the last price:

```
historical.iloc[-1].plot()
```

Here's the output:

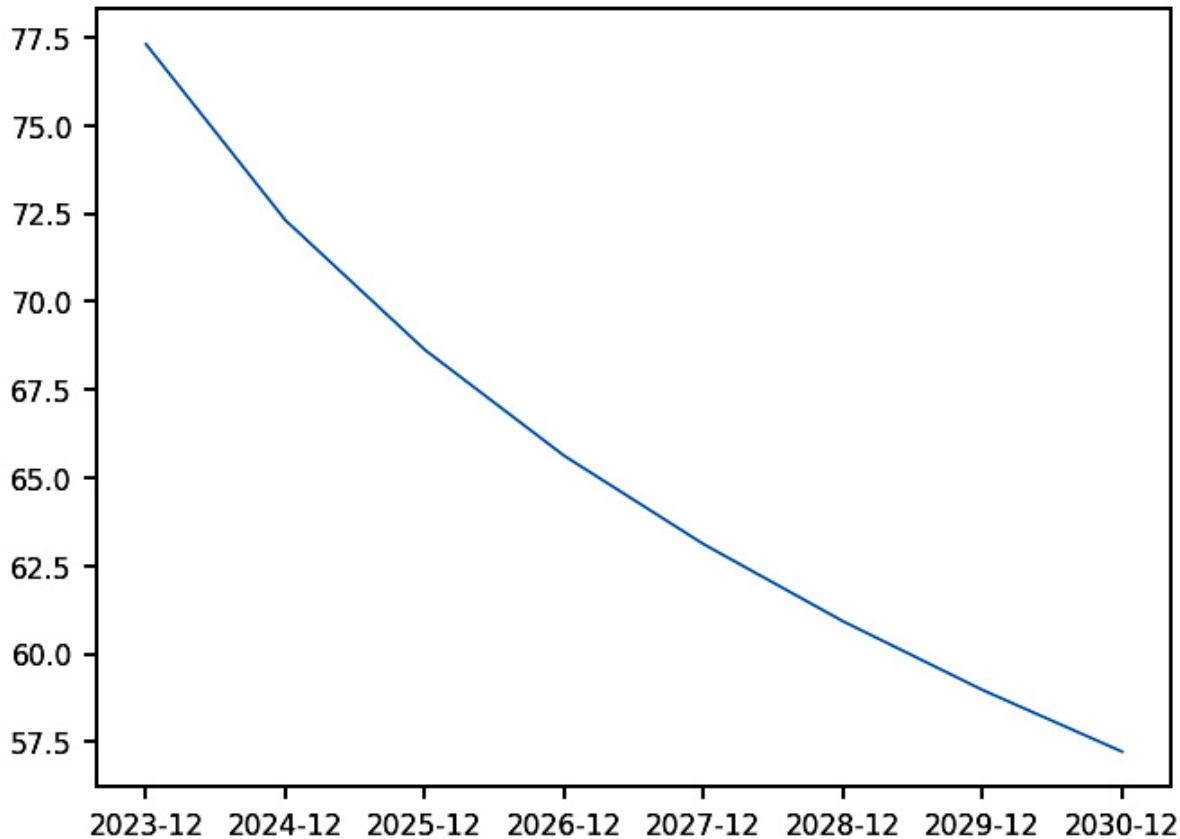


Figure 1.12: Futures curve for the December CL contract

See also

For more on the OpenBB Platform futures functionality, you can browse the following documentation:

- Documentation on the OpenBB Platform's futures **curve** method:
<https://docs.openbb.co/platform/reference/derivatives/futures/curve>
- Documentation on the OpenBB Platform's historical **curve** method:
<https://docs.openbb.co/platform/reference/derivatives/futures/historical>

Navigating options market data with the OpenBB Platform

Options are exchange-listed derivative contracts that convey the right (but not the obligation) to buy or sell the underlying stock at a certain price on or before a certain expiration date. Options are among the most versatile financial instruments in the market. They allow traders to define their risk profiles before entering trades and express market views not only on the direction of the underlying but the volatility. While options offer a high degree of flexibility for trading, this feature complicates data collection for research and backtesting.

A single underlying stock can have an array of options contracts with different combinations of strike prices and expiration dates. Collecting and manipulating this data is a challenge. The combination of options contracts for all strikes and expiration dates is commonly referred to as an options chain. There can be thousands of individual options contracts at a given time for a single underlying stock. Not only does the number of individual contracts pose a challenge, but getting price data has historically been expensive. With the introduction of the OpenBB Platform, it is now only a few lines of Python code to download options chains into a pandas DataFrame. This recipe will walk you through acquiring options data using the OpenBB Platform.

Getting ready...

By now, you should have the OpenBB Platform installed in your virtual environment. If not, go back to the beginning of this chapter and get it set up.

How to do it...

Similar to how we used the OpenBB Platform for futures data, we can use it for options data too:

1. Import the OpenBB Platform and Matplotlib for visualization:

```
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Use the `chains` method to download the entire options chain:

```
chains = obb.derivatives.options.chains(symbol="SPY")
```

3. Inspect the resulting DataFrame:

```
chains.info()
```

By running the preceding code, we'll see the details of the options chains data:

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8518 entries, 0 to 8517
Data columns (total 27 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   contract_symbol    8518 non-null   object  
 1   expiration        8518 non-null   object  
 2   strike            8518 non-null   float64 
 3   option_type       8518 non-null   object  
 4   open_interest     8518 non-null   int64  
 5   volume            8518 non-null   int64  
 6   theoretical_price 8518 non-null   float64 
 7   last_trade_price  8518 non-null   float64 
 8   tick              8518 non-null   object  
 9   bid               8518 non-null   float64 
 10  bid_size          8518 non-null   int64  
 11  ask               8518 non-null   float64 
 12  ask_size          8518 non-null   int64  
 13  open              8518 non-null   float64 
 14  high              8518 non-null   float64 
 15  low               8518 non-null   float64 
 16  prev_close        8518 non-null   float64 
 17  change             8518 non-null   float64 
 18  change_percent    8518 non-null   float64 
 19  implied_volatility 8518 non-null   float64 
 20  delta              8518 non-null   float64 
 21  gamma              8518 non-null   float64 
 22  theta              8518 non-null   float64 
 23  vega               8518 non-null   float64 
 24  rho                8518 non-null   float64 
 25  last_trade_timestamp 7190 non-null   datetime64[ns] 
 26  dte                8518 non-null   int64  
dtypes: datetime64[ns](1), float64(17), int64(5), object(4)
memory usage: 1.8+ MB

```

Figure 1.13: Preview of the data downloaded for the SPY options chains

Note that there are 8,518 options contracts for the **SPY Exchange Traded Fund (ETF)** that can be downloaded from CBOE (for free).

How it works...

The `obb.derivatives.options.chains` method downloads the entire options chain and stores it in a pandas DataFrame. The `obb.derivatives.options.chains` has an additional optional parameter:

- **provider:** The source from which the data should be downloaded. The default is CBOE. You can also select Tradier, Intrinio, or TMX. Note that for Tradier, Intrinio, and TMX, you need to provide your API key, which can be configured in the OpenBB Hub.

There's more...

You can use the OpenBB Platform to download historical options data for a single contract. To do this, you need the option symbol.

We'll use the `obb.equity.price.historical` method to get the historical options data for an SPY call option with a strike price of \$550 expiring on December 20, 2024:

```
data = obb.equity.price.historical(  
    symbol="SPY241220C00550000",  
    provider="yfinance"  
) [["close", "volume"]]
```

The result is a pandas DataFrame with the closing price and volume of the options contract.

	close	volume
date		
2023-05-31	2.920000	16
2023-06-02	3.690000	607
2023-06-05	3.740000	911
2023-06-06	3.750000	1543
2023-06-07	3.780000	0
...
2024-05-21	18.410000	83
2024-05-22	17.940001	127
2024-05-23	15.200000	55
2024-05-24	16.500000	627
2024-05-28	15.900000	21

Figure 1.14: Closing prices and volume of the SPY options contract

Options Greeks

Options Greeks measure how options prices change given a change in one of the inputs to an options pricing model. For example, **delta** measures how an options price changes given a change in the underlying stock price.

Using `obb.derivatives.options.chains`, the OpenBB Platform returns the most used Greeks including Delta, Gamma, Theta, Vega, and Rho.

See also

Options are a fascinating and deep topic that is rich with opportunities for trading. You can learn more about options, volatility, and how to analyze both via the OpenBB Platform:

- Free articles, code, and other resources for options trading: <https://pyquantnews.com/free-python-resources/options-trading-with-python/>
- Learn more about financial options if you're unfamiliar with them: [https://en.wikipedia.org/wiki/Option_\(finance\)](https://en.wikipedia.org/wiki/Option_(finance))
- Documentation on the OpenBB Platform's options methods: <https://docs.openbb.co/platform/reference/derivatives/options>

Harnessing factor data using `pandas_datareader`

Diversification is great until the entire market declines in value. That's because the overall market influences all assets. Factors can offset some of these risks by targeting drivers of return not influenced by the market. Common factors are size (large-cap versus small-cap) and style (value versus growth). If you think small-cap stocks will outperform large-cap stocks, then you might want exposure to small-cap stocks. If you think value stocks will outperform growth stocks, then you might want exposure to value stocks. In either case, you want to measure the risk contribution of the factor. Eugene Fama and Kenneth French built the Fama-French three-factor model in 1992. The three Fama-French factors are constructed using six value-weight portfolios formed on capitalization and book-to-market.

The three factors are as follows:

- **Small Minus Big**, which represents the differential between the average returns of three small-cap portfolios and three large-cap portfolios.
- **High Minus Low**, which quantifies the difference in average returns between two value-oriented portfolios and two growth-oriented portfolios.
- **Rm-Rf**, which denotes the market's excess return over the risk-free rate.

We'll explore how to measure and isolate alpha in [Chapter 5, Build Alpha Factors for Stock Portfolios](#). This recipe will guide you through the process of using `pandas_datareader` to fetch historic factor data for use in your analysis.

Getting ready...

By now, you should have the OpenBB Platform installed in your virtual environment. If not, go back to the beginning of this chapter and get it set up. By installing the OpenBB Platform, `pandas_datareader` will be installed and ready to use.

How to do it...

Using the `pandas_datareader` library, we have access to dozens of investment research factors:

1. Import `pandas_datareader`:

```
import pandas_datareader as pdr
```

2. Download the monthly factor data starting in January 2000:

```
factors = pdr.get_data_famafrance("F-F_Research_Data_Factors")
```

3. Get a description of the research data factors:

```
print(factors["DESCR"])
```

The result is an explanation of the data included in the DataFrame:

```
F-F Research Data Factors
-----
This file was created by CMPT_ME_BEME_RETTS using the 202305 CRSP database. The 1-month TBill return is from Ibbotson and Associates, Inc. Copyright 2023 Kenneth R. French
0 : (281 rows x 4 cols)
1 : Annual Factors: January–December (23 rows x 4 cols)
```

Figure 1.15: Preview of the description that is downloaded with factor data

4. Inspect the monthly factor data:

```
print(factors[0].head())
```

By running the preceding code, we get a DataFrame containing monthly factor data:

	Mkt-RF	SMB	HML	RF
Date				
2018-08	3.44	1.13	-3.98	0.16
2018-09	0.06	-2.28	-1.69	0.15
2018-10	-7.68	-4.77	3.44	0.19
2018-11	1.69	-0.68	0.28	0.18
2018-12	-9.57	-2.37	-1.85	0.20

Figure 1.16: Preview of the monthly data downloaded from the Fama-French Data Library

5. Inspect the annual factor data:

```
print(factors[1].head())
```

By running the preceding code, we get a DataFrame containing annual factor data:

	Mkt-RF	SMB	HML	RF
Date				
2018	-6.95	-3.21	-9.73	1.83
2019	28.28	-6.11	-10.34	2.15
2020	23.66	13.18	-46.56	0.45
2021	23.56	-3.89	25.53	0.04
2022	-21.60	-6.82	25.81	1.43

Figure 1.17: Preview of the annual data downloaded from the Fama-French Data Library

How it works...

Under the hood, `pandas_datareader` fetches data from the Fama-French Data Library by downloading a compressed CSV file, uncompressing it, and creating a pandas DataFrame.

There are 297 different datasets with different factor data available from the Fama-French Data Library. Here are some popular versions of the Fama-French 3-factor model for different regions:

- `Developed_3_Factors`
- `Developed_ex_US_3_Factors`
- `Europe_3_Factors`
- `Japan_3_Factors`
- `Asia_Pacific_ex_Japan_3_Factors`

You can use these in the `get_data_famafrench` method, just like `F-F_Research_Data_Factors`.

Some datasets return a dictionary with more than one DataFrame representing data for different time frames, portfolio weighting methodologies, and aggregate statistics. Data for these portfolios can be accessed using numerical keys. For example, the `5_Industry_Portfolios` dataset returns eight DataFrames in the dictionary. The first can be accessed using the `0` key, the second using the `1` key, and so on. Each dictionary includes a description of the dataset, which can be accessed using the `DESCR` key.

There's more...

`pandas_datareader` can be used to access data from many remote online sources. These include Tiingo, IEX, Alpha Vantage, FRED, Eurostat, and many more. Review the full list of data sources on

the documentation page: https://pandas-datareader.readthedocs.io/en/latest/remote_data.html.

See also

For more details on the factors available in the investment factor research library, take a look at the following resources. For another example of using the Fama-French 3-factor model, see the resources on the PyQuant News website:

- Documentation for all the Fama-French factor data: https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/Data_Library.html
- Details on the Fama-French 3-factor model: https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/Data_Library/f_factors.html
- Code walkthrough for using the Fama-French 3-factor model: <https://www.pyquantnews.com/past-pyquant-newsletter-issues>

Analyze and Transform Financial Market Data with pandas

The pandas library was invented by Wes McKinney while at the investment management firm **AQR Capital Management**, where he researched macro and credit trading strategies. He built pandas to provide flexible, easy-to-use data structures for data analysis. Since it was open sourced in 2009, pandas has become the standard tool to analyze and transform data using Python.

pandas is well-suited for working with tabular data, like that stored in spreadsheets or databases, and it integrates well with many other data analysis libraries in the Python ecosystem. Its capabilities extend to handling missing data, reshaping datasets, and merging and joining datasets, and it also provides robust tools for loading data from flat files, Excel files, databases, and HDF5 file formats. It's widely used in academia, finance, and many areas of business due to its rich features and ease of use.

This chapter will begin by covering recipes to help you build pandas data structures, primarily focusing on `DataFrames` and `Series`, the two primary data structures of pandas. You will learn how these structures allow for a wide variety of operations, such as slicing, indexing, and subsetting large datasets, all of which are crucial in algorithmic trading. From there, you will learn how to inspect and select data from `DataFrames`.

After you have a firm understanding of manipulating data using pandas, we'll cover recipes for analyses that are common in algorithmic trading, including how to compute asset returns and the volatility of a return series. The chapter will teach you how to generate a cumulative return series and resample data to different time frames, providing you with the flexibility to analyze data at various granularities. You will also learn how to handle missing data, a common issue in real-world datasets.

Lastly, this chapter will show you how to apply custom functions to time series data. Throughout this chapter, you will see how pandas integrates with other libraries in the scientific Python ecosystem, such as Matplotlib for data visualization, NumPy for numerical operations, and Scikit-Learn for machine learning.

In this chapter, we'll cover the following recipes:

- Diving into pandas index types
- Building pandas Series and DataFrames
- Manipulating and transforming DataFrames

- Examining and selecting data from DataFrames
- Calculating asset returns using pandas
- Measuring the volatility of a return series
- Generating a cumulative return series
- Resampling data for different time frames
- Addressing missing data issues
- Applying custom functions to analyze time series data

Diving into pandas index types

The Index is an immutable sequence that's used for indexing and alignment that serves as the label or key for rows in the DataFrame or elements in a series. It allows for fast lookup and relational operations and as of pandas version 2, it can contain values of any type, including integers, strings, and even tuples. Indexes in pandas are immutable, which makes them safe to share across multiple DataFrames or Series. They also have several built-in methods for common operations, such as sorting, grouping, and set operations such as union and intersection. pandas supports multiple indexes, allowing for complex, hierarchical data organization. This feature is particularly useful when dealing with high-dimensional data such as option chains. We'll see examples of MultiIndexes later in this chapter.

There are seven types of pandas indexes. The differences are dependent on the type of data used to create the index. For example, an Int64Index is an index that's made up of 64-bit integers. pandas is smart enough to create the right type of index, depending on the data used to instantiate it.

How to do it...

We'll start by creating a simple index with a series of integers:

1. Import pandas as the common alias, `pd`:

```
import pandas as pd
```

2. Instantiate the `Index` class:

```
idx_1 = pd.Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

3. Inspect the index:

```
print(idx_1)
```

Running the preceding code generates the index and prints out its type:

```
Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')
```

Figure 2.1: A pandas Int64Index index with 10 values

How it works...

This is the simplest example of creating a pandas index. You can instantiate the `Index` class with a one-dimensional, array-like data structure containing the values you want in the index.

There's more...

pandas has several `Index` types to support many use cases, including those related to time series analysis. We'll cover examples of the most often-used index types.

DatetimeIndex

A more interesting index type is `DatetimeIndex`. This type of index is extremely useful when dealing with time series data:

```
days = pd.date_range("2016-01-01", periods=6, freq="D")
```

This creates an index with six incremental datetime objects starting from 2016-01-01.

You can use many different frequencies, including `seconds`:

```
seconds = pd.date_range("2016-01-01", periods=100, freq="s")
```

By default, DatetimeIndexes are “timezone naive.” This means they have no time zone information attached. To “localize” `DatetimeIndex`, use `tz_localize`:

```
seconds_utc = seconds.tz_localize("UTC")
```

Localizing `DatetimeIndex` just appends time zone information to the object. It does not adjust the time from one time zone to another. To do that, you can use `tz_convert`:

```
seconds_utc.tz_convert("US/Eastern")
```

PeriodIndex

It's possible to create ranges of periods, such as quarters, using the pandas `period_range` method:

```
prng = pd.period_range("1990Q1", "2000Q4", freq="Q-NOV")
```

`period_range` returns a `PeriodIndex` object that has special labels that represent periods. In this case, the period starts in 1990Q1 and extends through to 2000Q4 at quarterly increments, with the year ending in November.

MultiIndex

`MultiIndex`, also known as a hierarchical index, is a data structure that allows for complex data organization within pandas DataFrames and Series. It allows data spanning multiple dimensions to be represented, even within the inherently two-dimensional structure of a DataFrame, by allowing multiple levels of indices. We'll look at `MultiIndex` in detail later in this book, so don't worry if it's not exactly clear how they work yet.

To create a `MultiIndex` object, pass a list of tuples to the `from_tuples` method:

```
tuples = [
    (pd.Timestamp("2023-07-10"), "WMT"),
    (pd.Timestamp("2023-07-10"), "JPM"),
    (pd.Timestamp("2023-07-10"), "TGT"),
    (pd.Timestamp("2023-07-11"), "WMT"),
    (pd.Timestamp("2023-07-11"), "JPM"),
    (pd.Timestamp("2023-07-11"), "TGT"),
]
midx = pd.MultiIndex.from_tuples(
    tuples,
    names=("date", "symbol")
)
```

There are several ways to create a `MultiIndex` object. In this example, we use a list of tuples. Each tuple contains a pandas timestamp and a ticker symbol. Using the list of tuples, you can call the `from_tuples` method to create the `MultiIndex` object. The result is a two-dimensional index that supports hierarchical DataFrames.

IMPORTANT NOTE

Giving the `names` argument to the `from_tuples` method is done to give names to the index columns for label-based indexing. This is optional.

See also

The pandas documentation is very thorough. To learn more about the index types we covered in this recipe, take a look at the following resources:

- Documentation on pandas indexes: <https://pandas.pydata.org/docs/reference/api/pandas.Index.html>
- Documentation on pandas DatetimeIndexes: <https://pandas.pydata.org/docs/reference/api/pandas.DatetimeIndex.html>
- Documentation on pandas MultiIndexes: <https://pandas.pydata.org/docs/reference/api/pandas.MultiIndex.html>

Building pandas Series and DataFrames

A Series is a one-dimensional labeled array that can hold any data type, including integers, floats, strings, and objects. The axis labels of a Series are collectively referred to as the index, which allows

for easy data manipulation and access. A key feature of the pandas Series is its ability to handle missing data, represented as a NumPy `nan` (Not a Number).

IMPORTANT

NumPy's `nan` is a special floating-point value. It is commonly used as a marker for missing data in numerical datasets. The `nan` value being a float is useful because it can be used in numerical computations and included in arrays of numbers without changing their data type, which aids in maintaining consistent data types in numeric datasets. Unlike other values, `nan` doesn't equal anything, which is why we need to use functions such as `numpy.isnan()` to check for `nan`.

Furthermore, the `Series` object provides a host of methods for operations such as statistical functions, string manipulation, and even plotting.

A DataFrame is a two-dimensional data structure that can be thought of as a spreadsheet with rows and columns. It's essentially a table of data with rows and columns. The data is aligned in a tabular fashion in rows and columns, and it can be of different types (for example, numbers and strings).

DataFrames make manipulating data easy, allowing for operations such as aggregation, slicing, indexing, subsetting, merging, and reshaping. They also handle missing data gracefully and provide convenient methods for filtering out, filling in, or otherwise manipulating null values.

Getting ready

Before building a Series and DataFrame, make sure you completed the previous recipe and have the indexes stored in memory.

How to do it...

Execute the following steps to construct a DataFrame out of several Series:

1. Import NumPy to create an array of normally distributed random numbers:

```
import numpy as np
```

2. Create a function that returns a NumPy array of randomly distributed numbers with the same length as the `DatetimeIndex` object you created earlier:

```
def rnd():
    return np.random.randn(100,)
```

The return value is a NumPy array of length 100. It's filled with random samples from a normal distribution. Creating a function allows you to return a different set of values each time the function is called.

TIP

When working with pandas, it's common to use random numbers to populate Series and DataFrames for testing.

3. Create three pandas Series that you'll use to create the DataFrame:

```
s_1 = pd.Series(rnd(), index=seconds)
s_2 = pd.Series(rnd(), index=seconds)
s_3 = pd.Series(rnd(), index=seconds)
```

We use the same DatetimeIndex object we created in the previous recipe as the index for each of the Series. This lets pandas align each of the columns along the common index.

4. Create the DataFrame using a dictionary:

```
df = pd.DataFrame({"a": s_1, "b": s_2, "c": s_3})
```

The result is a DataFrame with a DatetimeIndex object of second resolution and three columns all with samples from a normal distribution:

	a	b	c
2016-01-01 00:00:00	-1.075846	2.547374	0.415277
2016-01-01 00:00:01	-0.457549	0.498616	-0.574556
2016-01-01 00:00:02	-0.644713	1.843763	0.388706
2016-01-01 00:00:03	-0.268644	1.186620	0.857448
2016-01-01 00:00:04	-0.590639	-0.421629	-1.244135
...
2016-01-01 00:01:35	-1.996332	-0.625641	-0.173174
2016-01-01 00:01:36	-1.502497	1.172084	-0.238044
2016-01-01 00:01:37	-0.572685	0.106536	0.499070
2016-01-01 00:01:38	-0.905668	0.382154	-0.221563
2016-01-01 00:01:39	0.548253	-1.149373	-0.257776

Figure 2.2: DataFrame with a DatetimeIndex object and three columns of random data

How it works...

pandas Series and DataFrames have several ways of being constructed. Series can take any array-like, iterable (for example, list or tuple), dictionary, or scalar value to be created. In this example, you created a NumPy array-like object called an array. We passed in the `DatetimeIndex` object created in the previous recipe, which is the same length as the array that was used to create the Series.

DataFrames accept any multidimensional structured or homogenous objects, be they iterable, dictionaries, or other DataFrames. In this example, you passed in a Python dictionary, which is a list of key-value pairs. Each key represents the column name and each value represents the data for that column. DataFrames also accept an `index` argument in case the input data is not structured with an index. DataFrames can also accept a `columns` argument, which names the columns. Since the Series included indexes, pandas automatically used them to create the DataFrame index and aligned the values appropriately. If there were missing data for an index, pandas would still include the index but include `nan` as the value.

There's more...

A `MultiIndex` object is a multidimensional index that provides added flexibility to DataFrames. You can create a `MultiIndex` DataFrame from scratch or “reindex” an existing DataFrame to make it a `MultiIndex` DataFrame.

Building a MultiIndex DataFrame from scratch

Use the same `MultiIndex` object you created in the previous recipe:

```
tuples = [
    (pd.Timestamp("2023-07-10"), "WMT"),
    (pd.Timestamp("2023-07-10"), "JPM"),
    (pd.Timestamp("2023-07-10"), "TGT"),
    (pd.Timestamp("2023-07-11"), "WMT"),
    (pd.Timestamp("2023-07-11"), "JPM"),
    (pd.Timestamp("2023-07-11"), "TGT"),
]
midx = pd.MultiIndex.from_tuples(
    tuples,
    names=("date", "symbol")
)
```

Now that we have the index, we can create the DataFrame:

```
df_2 = pd.DataFrame(
{
    "close": [158.11, 144.64, 132.55, 158.20, 146.61, 134.86],
    "factor_1": [0.31, 0.24, 0.67, 0.29, 0.23, 0.71],
},
index=midx,
```

We create the DataFrame using a dictionary. The keys are the column names and the values are the data for the columns. Note that we use the `index` argument while passing in the `MultiIndex` object. The result is a DataFrame that contains records for each of the three symbols for each date:

close factor_1			
	date	symbol	
2023-07-10	WMT	158.11	0.31
	JPM	144.64	0.24
	TGT	132.55	0.67
2023-07-11	WMT	158.20	0.29
	JPM	146.61	0.23
	TGT	134.86	0.71

Figure 2.3: MultiIndex DataFrame

Reindexing an existing DataFrame with a MultiIndex object

It's common to add a `MultiIndex` object to a DataFrame. Let's consider an example of reindexing options data for a `MultiIndex` object:

1. Import the OpenBB Platform:

```
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download option chains using OpenBB:

```
chains = obb.derivatives.options.chains(
    "SPY",
    provider=".cboe"
)
```

The result is a DataFrame with a `RangeIndex` object starting at 0:

	contract_symbol	expiration	strike	option_type	open_interest	...	theta	vega	rho	last_trade_timestamp	dte
0	SPY240703C00445000	2024-07-03	445.0	call	2	...	0.0000	0.000	0.0000	2024-07-01 14:02:04	0
1	SPY240703P00445000	2024-07-03	445.0	put	5	...	0.0000	0.000	0.0000	2024-07-02 09:50:35	0
2	SPY240703C00450000	2024-07-03	450.0	call	5	...	0.0000	0.000	0.0000	2024-07-03 12:43:35	0
3	SPY240703P00450000	2024-07-03	450.0	put	30	...	0.0000	0.000	0.0000	2024-07-02 11:59:56	0
4	SPY240703C00455000	2024-07-03	455.0	call	0	...	0.0000	0.000	0.0000	NaT	0
...
8641	SPY261218P00810000	2026-12-18	810.0	put	0	...	-0.0692	0.000	-0.0001	NaT	898
8642	SPY261218C00815000	2026-12-18	815.0	call	0	...	-0.0077	1.309	0.9418	NaT	898
8643	SPY261218P00815000	2026-12-18	815.0	put	0	...	-0.0692	0.000	-0.0001	NaT	898
8644	SPY261218C00820000	2026-12-18	820.0	call	8	...	-0.0074	1.262	0.8956	2024-07-02 13:57:35	898
8645	SPY261218P00820000	2026-12-18	820.0	put	0	...	-0.0692	0.000	-0.0001	NaT	898

Figure 2.4: DataFrame with SPY options data

3. Options are derivatives that are frequently grouped by expiration date, strike price, option type, and any combination of those three. Using the `set_index` method takes the arguments in the list and uses those columns as indexes, converting `RangeIndex` into a `MultiIndex` object. In this example, we use the expiration date, strike price, and option type as the three indexes:

```
df_3 = chains.set_index(["expiration", "strike",
                       "option_type"])
```

The result is a hierarchical DataFrame with a three-dimensional `MultiIndex`:

			contract_symbol	open_interest	...	last_trade_timestamp	dte	
expiration	strike	option_type						
2024-05-28	444.0	call	SPY240528C00444000	1	...	2024-05-24 12:00:01	0	
		put	SPY240528P00444000	498	...	2024-05-24 15:59:58	0	
	445.0	call	SPY240528C00445000	3	...	2024-05-28 14:21:58	0	
		put	SPY240528P00445000	201	...	2024-05-23 15:44:44	0	
	446.0	call	SPY240528C00446000	0	...	NaT	0	
		
	2026-12-18	785.0	put	SPY261218P00785000	0	...	2024-04-03 09:36:19	934
		790.0	call	SPY261218C00790000	0	...	2024-05-28 13:07:31	934
			put	SPY261218P00790000	0	...	NaT	934
		795.0	call	SPY261218C00795000	4	...	2024-05-23 11:42:10	934
			put	SPY261218P00795000	0	...	NaT	934

Figure 2.5: A DataFrame with a three-dimensional MultiIndex

You can inspect the index using dot notation:

```
df_3.index
```

As you can see, the `MultiIndex` object is a list of tuples. Each tuple contains the elements of the `MultiIndex` object. We also have a property called `names`, which includes a list of the `MultiIndex` level names:

```
MultiIndex([(2023-07-12, 372.0, 'call'),
            (2023-07-12, 372.0, 'put'),
            (2023-07-12, 373.0, 'call'),
            (2023-07-12, 373.0, 'put'),
            (2023-07-12, 374.0, 'call'),
            (2023-07-12, 374.0, 'put'),
            (2023-07-12, 375.0, 'call'),
            (2023-07-12, 375.0, 'put'),
            (2023-07-12, 376.0, 'call'),
            (2023-07-12, 376.0, 'put'),
            ...
            (2025-12-19, 645.0, 'call'),
            (2025-12-19, 645.0, 'put'),
            (2025-12-19, 650.0, 'call'),
            (2025-12-19, 650.0, 'put'),
            (2025-12-19, 655.0, 'call'),
            (2025-12-19, 655.0, 'put'),
            (2025-12-19, 660.0, 'call'),
            (2025-12-19, 660.0, 'put'),
            (2025-12-19, 665.0, 'call'),
            (2025-12-19, 665.0, 'put')],
           names=['expiration', 'strike', 'optionType'], length=7306)
```

Figure 2.6: Inspecting the MultiIndex object

See also

To learn more about the Series and DataFrame objects, take a look at the following resources:

- Documentation on pandas Series: <https://pandas.pydata.org/docs/reference/api/pandas.Series.html>
- Documentation on pandas DataFrames: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html>

Manipulating and transforming DataFrames

Before moving on to more advanced recipes, it's important to understand the fundamentals of working with data. DataFrames are the most common pandas data structures you'll work with. Despite the existence of hundreds of methods for DataFrame manipulation, only a subset of these are regularly used.

In this recipe, we will show you how to manipulate a DataFrame using the following common methods:

- Creating new columns using aggregates, Booleans, and strings
- Concatenating two DataFrames together
- Pivoting a DataFrame such as Excel

- Grouping data on a key or index and applying an aggregate
- Joining options data together to create straddle prices

Getting ready...

We'll start by importing the necessary libraries and downloading market price data:

1. Start by importing NumPy, pandas, and the OpenBB Platform:

```
import numpy as np
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Load some stock price data to work with:

```
asset = obb.equity.price.historical(
    "AAPL",
    provider="yfinance"
)
benchmark = obb.equity.price.historical(
    "SPY",
    provider="yfinance"
)
```

How to do it...

Creating new columns is a common operation when dealing with DataFrames. Let's learn how to do this.

Creating new columns using aggregates, Booleans, and strings

Here's how to create new columns using aggregates, Booleans, and strings:

1. Start by renaming the columns so that they follow Python conventions:

```
columns = [
    "open",
    "high",
    "low",
    "close",
    "volume",
    "dividends",
    "splits",
]
asset.columns = columns
benchmark.columns = columns + ["capital_gain"]
```

2. Add a new column with values from an aggregate:

```
asset["price_diff"] = asset.close.diff()
benchmark["price_diff"] = benchmark.close.diff()
```

TIP

When adding a new column to a DataFrame, the data you provide is automatically aligned by the DataFrame's index. This means that if you add a Series as a new column, pandas matches each value to its corresponding row in the DataFrame using the Series's index. If any index values in the DataFrame don't exist in the new Series, pandas will fill them with `nan` values, signifying missing data.

3. Add a new column with a Boolean:

```
asset["gain"] = asset.price_diff > 0  
benchmark["gain"] = benchmark.price_diff > 0
```

4. Add a new column with a string value:

```
asset["symbol"] = "AAPL"  
benchmark["symbol"] = "SPY"
```

Three new columns of data are added to each DataFrame – `returns`, `gain`, and `symbol`:

	open	high	low	close	adj_close	volume	dividends	splits	price_diff	gain	symbol
date											
2020-08-17	114.001870	114.026427	111.939155	112.572701	112.572701	119561600	0.0	0.0	NaN	False	AAPL
2020-08-18	112.322219	113.940467	111.983343	113.510735	113.510735	105633600	0.0	0.0	0.938034	True	AAPL
2020-08-19	113.923294	115.082344	113.557410	113.653175	113.653175	145538000	0.0	0.0	0.142441	True	AAPL
2020-08-20	113.694908	116.290493	113.677717	116.175079	116.175079	126907200	0.0	0.0	2.521904	True	AAPL
2020-08-21	117.145048	122.650538	117.132773	122.161873	122.161873	338054800	0.0	0.0	5.986794	True	AAPL
...
2023-08-16	177.130005	178.539993	176.500000	176.570007	176.570007	46964900	0.0	0.0	-0.879990	False	AAPL
2023-08-17	177.139999	177.509995	173.479996	174.000000	174.000000	66062900	0.0	0.0	-2.570007	False	AAPL
2023-08-18	172.300003	175.100006	171.960007	174.490005	174.490005	61114200	0.0	0.0	0.490005	True	AAPL
2023-08-21	175.070007	176.130005	173.740005	175.839996	175.839996	46311900	0.0	0.0	1.349991	True	AAPL
2023-08-22	177.059998	177.677795	176.250000	177.229996	177.229996	41363946	0.0	0.0	1.389999	True	AAPL

Figure 2.7: Result of adding new columns to the asset DataFrame

5. Set a single value based on the aggregate of values:

```
asset_2 = asset.copy()  
asset_2.at[  
    asset_2.index[10],  
    "volume"  
] = asset_2.volume.mean()
```

You can inspect the value by using the `iat` method on the `asset_2` DataFrame:

```
asset_2.iat[10, 5]
```

The result is a scalar value representing the mean volume between indexes 5 and 10.

Concatenating two DataFrames together

In the simplest case, concatenating two DataFrames together either stacks one on top of each other (row-wise concatenation) or lines them up next to each other (column-wise concatenation). You can control this through the `axis` argument.

Call the pandas `concat` method, leaving `axis` as the default (0 for row-wise concatenation):

```
pd.concat([asset, asset_2]).drop_duplicates()
```

Here are the concatenated DataFrames:

	open	high	low	close	adj_close	volume	dividends	splits	price_diff	gain	symbol
date											
2020-08-17	114.001870	114.026427	111.939155	112.572701	112.572701	119561600	0.0	0.0	NaN	False	AAPL
2020-08-18	112.322219	113.940467	111.983343	113.510735	113.510735	105633600	0.0	0.0	0.938034	True	AAPL
2020-08-19	113.923294	115.082344	113.557410	113.653175	113.653175	145538000	0.0	0.0	0.142441	True	AAPL
2020-08-20	113.694908	116.290493	113.677717	116.175079	116.175079	126907200	0.0	0.0	2.521904	True	AAPL
2020-08-21	117.145048	122.650538	117.132773	122.161873	122.161873	338054800	0.0	0.0	5.986794	True	AAPL
...
2023-08-17	177.139999	177.509995	173.479996	174.000000	174.000000	66062900	0.0	0.0	-2.570007	False	AAPL
2023-08-18	172.300003	175.100006	171.960007	174.490005	174.490005	61114200	0.0	0.0	0.490005	True	AAPL
2023-08-21	175.070007	176.130005	173.740005	175.839996	175.839996	46311900	0.0	0.0	1.349991	True	AAPL
2023-08-22	177.059998	177.677795	176.250000	177.229996	177.229996	41363946	0.0	0.0	1.389999	True	AAPL
2020-08-31	125.314885	128.674163	123.762935	126.748955	126.748955	212727600	0.0	4.0	4.157341	True	AAPL

Figure 2.8: Two DataFrames concatenated row-wise (stacked on top of each other)

The method takes a list of DataFrames to concatenate. In this example, we concatenated two identical DataFrames before dropping the duplicate values using the `drop_duplicates` method.

Pivoting a DataFrame such as Excel

Pivot tables are ubiquitous in Excel. They're used to aggregate data along a defined set of columns. They work the same way in pandas. In this example, we'll aggregate the returns using three methods – `sum`, `mean`, and `std`:

Call the `pivot_table` method:

```
pd.pivot_table(
    data=asset,
    values="price_diff",
    columns="gain",
    aggfunc=["sum", "mean", "std"]
)
```

The result is a pivoted DataFrame with `MultiIndex` column labels:

	sum		mean		std	
gain	False	True	False	True	False	True
returns	-745.20977	809.867065	-2.025027	2.076582	1.832028	1.773361

Figure 2.9: A pivoted DataFrame

Grouping data on a key or index and applying an aggregate

Grouping lets you aggregate different sections of your data. This is useful when you're working with market data and a single DataFrame contains prices for multiple assets:

1. Concatenate asset data and benchmark data into the same DataFrame:

```
concated = pd.concat([asset, benchmark])
```

2. Group the resulting DataFrame by the `symbol` column, return the `adj_close` column, and apply the `ohlc` aggregate:

```
concated.groupby("symbol").close.ohlc()
```

The result is a DataFrame with `open`, `high`, `low`, and `close` properties for the asset symbol (**AAPL**) and the benchmark symbol (**SPY**):

symbol	open	high	low	close
AAPL	112.572701	196.185074	104.943108	177.229996
SPY	323.008636	466.563324	309.646606	438.149994

Figure 2.10: Grouped DataFrame

TIP

The `ohlc` aggregate is a pandas Resampler. `open` is the first value of the group, `high` is the maximum value of the group, `low` is the minimum value of the group, and `close` is the last value of the group.

Joining options data together to create straddle prices

DataFrame joins are similar to SQL joins. It combines two DataFrames based on a matching key. A great use case for joins is combining two DataFrames containing options chains to price straddles. A straddle is a complex options position made up of a long call and long put with the same strike and expiration:

1. Download option chains data using the OpenBB Platform:

```
chains = obb.derivatives.options.chains(
    "AAPL", provider=".cboe")
```

2. Filter out the call options and the put options for a specific expiration:

```

expirations = chains.expiration.unique()
calls = chains[
    (chains.optionType == "call")
    & (chains.expiration == expirations[5])
]
puts = chains[
    (chains.optionType == "put")
    & (chains.expiration == expirations[5])
]

```

TIP

The pandas `unique` method returns an array of unique values from the Series. In this case, it returns a unique set of expiration dates. From there, we select the expiration at index 5, which is sufficiently into the future and is where there will be trading activity.

- Set the index to the strike prices:

```

calls_strike = calls.set_index("strike")
puts_strike = puts.set_index("strike")

```

- Then, join the call DataFrame on the put DataFrame using a left join:

```

joined = calls_strike.join(
    puts_strike,
    how="left",
    lsuffix="_call",
    rsuffix="_put"
)

```

Since both DataFrames have a column named `lastPrice`, we add the `lsuffix` and `rsuffix` arguments to distinguish the `lastPrice` columns from each other.

- Use only the price columns from the joined DataFrame:

```

prices = joined[["last_trade_price_call",
                 "last_trade_price_put"]]

```

- Sum up the call and put prices by using the `axis` argument in the `sum` method:

```

prices["straddle_price"] = prices.sum(axis=1)

```

The result is a DataFrame with the strike prices as the index, and the call, put, and straddle prices as columns for each strike:

strike	last_trade_price_call	last_trade_price_put	straddle_price
100.0	0.0	0.0	0.0
105.0	0.0	0.0	0.0
110.0	0.0	0.0	0.0
115.0	0.0	0.0	0.0
120.0	0.0	0.0	0.0
...
245.0	0.0	0.0	0.0
250.0	0.0	0.0	0.0
255.0	0.0	0.0	0.0
260.0	0.0	0.0	0.0
265.0	0.0	0.0	0.0

Figure 2.11: DataFrame containing call, put, and straddle prices

How it works...

In this recipe, we covered four ways of manipulating DataFrames.

DataFrames supports column addition via the `df["new_col"] = value` assignment syntax. For aggregates, methods such as `groupby` and `agg` compute summary metrics, which can be joined to the original DataFrame. By using Boolean indexing, such as `df["new_col"] = df["existing_col"] > threshold`, we can create new columns based on conditions.

The `pivot_table` method reshapes data, providing a multidimensional analysis tool similar to pivot tables in Excel. By specifying the `index` and `columns` parameters, we can set row and column keys, respectively. The `values` parameter denotes which DataFrame columns to aggregate, and `aggfunc` defines the aggregation function, such as `sum` or `mean`. This method then constructs a new DataFrame, where rows and columns are determined by unique combinations of key values.

The `groupby` method enables data segmentation based on column values. By passing one or multiple column names to `groupby`, we create a `GroupBy` object, segmenting the data. This object allows aggregation, transformation, or filtering of these groups using methods such as `sum`, `mean`, or `apply`.

The result is a DataFrame with an index based on the grouping columns and aggregated or transformed values.

Finally, we covered the join method, which combines two DataFrames based on their indexes. Specifying the `on` parameter can alter the default behavior, using a particular column as the join key. The `how` parameter determines the type of join: `left`, `right`, `inner`, or `outer`. This method appends columns from the joining DataFrame to the calling DataFrame based on matching index or column values.

There's more...

The pandas `groupby` method is one of the most used transforms since it lets you aggregate and perform analysis on different chunks of data in the same DataFrame. Here are a few more ways to use it.

Grouping by multiple columns

`groupby` can be used with multiple columns, providing a detailed view of your data. This is useful when you want to group data based on a combination of values from different columns. For example, let's say you want to aggregate the open interest for each contract:

```
(  
    chains  
    .groupby(  
        ["option_type", "strike", "expiration"]  
    )  
    .open_interest  
    .sum()  
)
```

The result is a Series with a `MultiIndex` object composed of `optionType`, `strike`, and `expiration`. The values are the sum of the `openInterest` aggregate for each group:

```
option_type  strike  expiration  
call         5.0     2024-03-15      2.0  
                  2024-06-21      341.0  
                  2024-09-20     128.0  
          10.0     2024-03-15      0.0  
                  2024-06-21     12.0  
                  ...  
put          330.0   2024-08-16      0.0  
          340.0   2024-08-16      0.0  
          350.0   2024-08-16      0.0  
          360.0   2024-08-16      0.0  
          370.0   2024-08-16      0.0  
Name: open_interest, Length: 2086, dtype: float64
```

Figure 2.12: Option chains grouped by option type, strike, and expiration with the openInterest aggregate

Applying different methods to different columns

You can use `agg` to apply specific methods to specific columns after grouping. For example, let's say you want to find the maximum `lastPrice` and the total `openInterest` for each `optionType`:

```
(  
    chains  
    .groupby(  
        ["option_type", "strike", "expiration"]  
    )  
    .agg({  
        "last_trade_price": "max",  
        "open_interest": "sum"  
    })  
)
```

This provides a DataFrame with `option_type`, `strike`, and `expiration` as the indexes and two columns: one with the maximum `last_price` and the other with the sum of `open_interest` for each option type. It offers a snapshot of the highest option price and total open interest for the call and put options:

option_type	strike	expiration	last_trade_price	open_interest
call	5.0	2024-06-21	185.05	31
		2024-07-19	164.00	0
		2024-08-16	182.23	1
		2024-09-20	179.30	2
		2024-10-18	172.06	0
...		
put	370.0	2024-08-16	0.00	0
		2024-10-18	0.00	0
		2024-11-15	187.00	0
		2025-03-21	0.00	0
		380.0	2025-03-21	189.89

Figure 2.13: Maximum price and aggregate open interest for each combination of option type, strike, and expiration

Applying custom functions

You can apply your own functions using the `apply` method. For example, let's say you want to calculate the average spread (difference between `ask` and `bid`) for each group of `optionType`:

```
(  
    chains  
    .groupby(["option_type"])  
    .apply(lambda x: (x["ask"] - x["bid"]).mean(),           include_groups=False)  
)
```

The result is a Series with `option_type` as the index, where the values represent the average spread for each option type:

```
option_type  
call      1.274094  
put       1.260757  
dtype: float64
```

Figure 2.14: Average spread for calls and puts

TIP

A *lambda function* can't be defined using the `def` keyword. It can accept any number of input arguments and returns any number of outputs. Lambda functions must exist on a single line and are useful for logic that can be written concisely.

Grouping and transforming data

The `transform` method can be used after `groupby` to return a Series with the same shape as the original, where each group is replaced with the value from a method applied to that group. For example, let's say you want to compute the z-score of `lastPrice` within each `expiration` date:

```
(  
    chains  
    .groupby("expiration")  
    .last_trade_price  
    .transform(lambda x: (x - x.mean()) / x.std())  
)
```

This results in a Series of the same length as `chains`, where `lastPrice` is replaced with its z-score value within each `expiration` date. The z-score can be useful for comparisons as it puts all the prices on the same scale relative to their specific expiration date group:

```
0      3.552798
1     -0.646571
2      3.226523
3     -0.646871
4      3.142709
...
1855    1.456281
1856   -0.917640
1857    1.613997
1858   -0.942438
1859   -1.048326
Name: lastPrice, Length: 1860, dtype: float64
```

Figure 2.15: Creating a standardized price within each expiration

See also

For more on option straddles, Investopedia offers a high-quality description. To learn more about the methods that were used in this recipe, take a look at the following documentation:

- Understanding option staddles: <https://www.investopedia.com/terms/s/straddle.asp>
- Documentation on concatenating pandas data structures: <https://pandas.pydata.org/docs/reference/api/pandas.concat.html>
- Documentation on pivoting pandas DataFrames: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot.html>
- Documentation on grouping pandas DataFrames: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html>
- Documentation on joining pandas DataFrames: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.join.html>

Examining and selecting data from DataFrames

Once you've loaded, manipulated, and transformed data in DataFrames, the next step is retrieving the data from DataFrames. This is where indexing and selecting data come into play. This functionality allows you to access data using methods such as `iloc` and `loc` and techniques such as Boolean indexing or query functions. These methods can target data based on its position, labels, or condition based on whether you're after a specific row, column, or combination. Inspection enables potential issues to be identified, such as missing values, outliers, or inconsistencies, that can affect analysis and modeling. Additionally, an initial inspection provides insights into the nature of data, helping determine appropriate preprocessing steps and analysis methods.

How to do it...

Let's start by downloading stock price data:

1. Start by importing pandas and the OpenBB Platform:

```
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Then, load some data:

```
df = obb.equity.price.historical(
    "AAPL",
    start_date="2021-01-01",
    provider="yfinance"
)
```

3. Display the first 5 records:

```
df.head(5)
```

4. Display the last 5 records:

```
df.tail(5)
```

5. Return the DataFrame as a NumPy array:

```
df.values
```

6. Get descriptive statistics:

```
df.describe()
```

7. Rename the columns so that they follow Python conventions:

```
df.columns = [
    "open",
    "high",
    "low",
    "close",
    "volume",
    "dividends",
    "splits",
]
```

8. Select one column of data:

```
df["close"]
```

IMPORTANT NOTE

When selecting one column from a DataFrame, the return type is a Series. If you try to apply a DataFrame method on a Series, this results in an exception. To select one column while returning a DataFrame instead of a Series, slice the DataFrame using a list by using df[["close"]].

9. Select rows by index:

```
df[0:3]
```

IMPORTANT NOTE

When slicing Python objects using indexes, the result is not inclusive of the last index. For example, when slicing using indexes 0:3, the values at index location 0, 1, and 2 are returned, but not the value at index location 3.

10. pandas is smart enough to turn strings into **DatetimeIndex** values and use them to return a range of data between two dates:

```
df.index = pd.to_datetime(df.index)
df["2021-01-02":"2021-01-11"]
```

IMPORTANT NOTE

When slicing Python objects using labels, the result is inclusive of the last label. For example, when slicing using labels "2021-01-02":"2021-01-11", the value at label location "2021-01-11" is included in the return value.

Selection by label using loc

The **loc** method is a method that's used for data selection based on label-based indexing. It allows data to be selected by row, column, or both, using labels of rows or list-like objects. This method is useful when the index of a DataFrame is a label other than a numerical series:

1. Selecting a single row transposes the Series and puts the DataFrame columns as row labels:

```
df.loc[df.index[0]]
```

2. Selecting a single row and single column returns a scalar:

```
df.loc[df.index[0], "close"]
```

3. Selecting a range of rows and a list of columns returns a subsection of the data:

```
df.loc[df.index[0:6], ["high", "low"]]
```

4. Selecting a range of labels and a list of columns returns a subsection of the data:

```
df.loc["2021-01-02":"2021-01-11", ["high", "low"]]
```

Selection by position using iloc

The pandas **iloc** method is a data selection method that's used for indexing based on integer locations. **iloc** works independently of the DataFrame's index labels:

1. Selecting a single row by index location transposes the Series and puts the DataFrame columns as row labels:

```
df.iloc[3]
```

2. Selecting a range of rows by index and a range of columns by index returns a subset of the data:

```
df.iloc[3:5, 0:2]
```

3. Selecting a combination of specific rows and columns by index works too:

```
df.iloc[[1, 2, 4], [0, 2]]
```

Selection by Boolean indexing

Boolean indexing involves generating a Boolean Series that corresponds to the rows in the DataFrame, where `true` indicates that the row meets the condition and `false` denotes it does not. By passing this Boolean Series to the DataFrame, it returns only the rows where the condition is `true`, thereby allowing for conditional selection and manipulation of data:

1. Inspect where a condition is True for each row:

```
df.close > df.close.mean()
```

2. Using single-column values to select data:

```
df[df.close > df.close.mean()]
```

3. Use the result to return all the rows and the data at column index location 0:

```
df[df.close > df.close.mean()].iloc[:, 0]
```

4. Use label-based indexing with Boolean indexing to create query-like slicing:

```
df.loc[  
    (df.close > df.close.mean())  
    & (df.close.mean() > 100)  
    & (df.volume > df.volume.mean())  
]
```

How it works...

The `loc` method is for label-based data selection within a DataFrame or Series. We can reference rows and columns using their labels, defining both the row index and column name in `df.loc[row, column]` format. It supports slicing, as we saw previously, enabling multiple rows and columns to be selected based on label ranges.

The `iloc` method is for integer-location-based indexing. We use the index positions referencing rows and columns in `df.iloc[row, column]` format. It also supports slicing, allowing multiple rows and columns to be selected based on integer ranges. Unlike `loc`, `iloc` disregards index or column labels and strictly operates on integer positions.

Boolean indexing allows for data filtering based on condition evaluations. Within a DataFrame or Series, a condition that's applied to a column or the entire structure returns a Boolean array, where `true` values indicate rows meeting the condition. By placing this Boolean array within the DataFrame's square brackets – for example, `df[boolean_array]` – only the rows with `true` values are extracted. Multiple conditions can be combined using the `&` (and), `|` (or), and `~` (not) operators. Additionally, the `query` method can be used as an alternative to achieve the same result in a more readable syntax. Boolean indexing can return results from full DataFrames considerably faster than looping or enumerating through DataFrame rows.

There's more...

Indexing is one of the most important aspects of working with pandas. Let's look at some advanced examples of how to slice data.

Partial string indexing

You can select data using a `DatetimeIndex` object based on a partial string:

```
df["2023"]
```

The preceding code will return a DataFrame containing all records from 2023. The resulting DataFrame maintains the same columns as the original DataFrame but only includes rows where the index falls within 2023.

You can also include the month:

```
df["2023-07"]
```

The preceding code will return a DataFrame with all records from July 2023.

Using `at` for fast access to a scalar

If we want to quickly access the value of a specific cell, we can use the `at` accessor. For example, to get the `close` price on July 12, 2023, we can run the following command:

```
df.at["2023-07-12", "close"]
```

The result is the closing price on July 12, 2023. The `at` accessor provides a fast way to access a single value at a specific row-column pair.

Using `nsmallest` and `nlargest`

Pandas provides handy methods to get rows with the smallest or largest values in a column. For instance, to get the five rows with the highest volume, we can use the following command:

```
df.nlargest(5, "volume")
```

The result is a DataFrame with the five rows with the highest `volume` values.

Using the `query` method to query DataFrames

The `query` method allows you to query a DataFrame in a more readable way. For instance, to get all rows where the `close` price is higher than the `open` price, you can use the following command:

```
df.query("close > open")
```

The resulting DataFrame will contain all rows where the `close` price was higher than the `open` price.

See also

Indexing can be a complex topic that takes practice to master. This recipe covers many use cases that are common in algorithmic trading. For more details, take a look at the following documentation on indexing and slicing pandas DataFrames: https://pandas.pydata.org/docs/user_guide/indexing.html.

Calculating asset returns using pandas

Returns are integral to understanding the performance of a portfolio. There are two types: simple returns and compound (or log) returns.

Simple returns, which are calculated as the difference in price from one period to the next divided by the price at the beginning of the period, are beneficial in certain circumstances. They aggregate across assets, meaning the simple return of a portfolio is the aggregate of the returns of the individual assets, weighted according to their proportions. This trait makes simple returns practical for comparing assets and evaluating portfolio performance over short-term intervals.

Simple returns are defined as follows:

$$R_t = P_t - P_{t-1} / P_{t-1} = P_t / P_{t-1} - 1$$

On the other hand, **compound returns**, which are calculated using the natural logarithm of the price-relative change, are additive over time. This quality makes them suitable for multi-period analyses as the compound return for a given period is the sum of the log returns within that period.

Compound returns are defined as follows:

$$r_t = \log(P_t / P_{t-1}) = \log(P_t) - \log(P_{t-1})$$

Compound returns are generally preferred in practice since they have mathematical properties that make them easier to use in analytics. They're less influenced by extreme values, which might skew the analysis. A major price change in a single period will have a large effect on simple returns but a moderated effect on log returns.

The additive nature of compound returns across time is mathematically convenient. If you need to calculate the total return over a sequence of periods, you simply sum up the log returns, whereas for simple returns, you would need to calculate the product of (1 plus each period's return) and then subtract one.

Given a daily compounded return, r_t , it is straightforward to solve back for the corresponding simple net return, R_t :

$$R_t = e^{r_t} - 1$$

The difference between simple and log returns becomes evident when we look into the statistical properties. If we assume that asset prices follow a log-normal distribution, which may not always be the case, the log returns would be normally distributed. This assumption is beneficial because normal distribution is a cornerstone in statistical modeling. For daily or intraday data, the difference between simple and log returns is usually minimal, with log returns generally being slightly smaller.

How to do it...

Let's start building returns using stock data from the OpenBB Platform:

1. Import pandas, NumPy, and the OpenBB Platform:

```
import numpy as np
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download stock price data for AAPL:

```
data = obb.equity.price.historical(
    "AAPL",
    provider="yfinance"
)
```

3. Select the close data:

```
df = data.loc[:, ["close"]]
```

4. Add a new column with simple returns and compound returns using the adjusted closing price:

```
df["simple"] = df["close"].pct_change()
df["compound"] = np.log(
    df["close"] / df["close"].shift()
)
```

The resulting DataFrame now has two additional columns:

	close	simple	compound
date			
2023-05-30	177.300003	NaN	NaN
2023-05-31	177.250000	-0.000282	-0.000282
2023-06-01	180.089996	0.016023	0.015896
2023-06-02	180.949997	0.004775	0.004764
2023-06-05	179.580002	-0.007571	-0.007600
...
2024-05-21	192.350006	0.006857	0.006834
2024-05-22	190.899994	-0.007538	-0.007567
2024-05-23	186.880005	-0.021058	-0.021283
2024-05-24	189.979996	0.016588	0.016452
2024-05-28	189.990005	0.000053	0.000053

Figure 2.16: The resulting DataFrame with daily simple and compound returns

How it works...

First, we imported NumPy for numerical operations, pandas for data manipulation, and the OpenBB Platform for financial data retrieval.

Then, we used the OpenBB Platform's `stocks.load` method to fetch the stock price data for AAPL. The data was stored in a DataFrame.

Next, we created a new DataFrame containing only the `close` column. We did this using the `loc` function, a label-based data selector.

After, we used the `pct_change` method to calculate simple returns, which represent the percentage change in the adjusted close price from the previous day. The result was stored in a new column called `simple`.

For compound returns, we calculated the ratio of the current day's adjusted close price to the previous day's price using the `shift` method. This shifted the DataFrame index by one period. Then, we took the natural logarithm of these ratios with `log`, reflecting the formula for compound returns. The results were stored in another new column called `compound`.

There's more...

The `pct_change` method accepts a `periods` argument that will shift the input data by that number before computing the simple return. This is useful for computing returns that are multiple periods, such as 3 days:

```
df["close"].pct_change(periods=3)
```

The result is a new Series, where each element represents a three-period simple return. Note that the first three elements of the resulting Series will be `nan` because there aren't enough prior periods for the calculation. (We'll learn how to fill in missing data in the *Addressing Missing Data Issues* recipe in this chapter.)

You can also do the same with compound returns by passing an argument to the `shift` method:

```
np.log(df["close"] / df["close"].shift(3))
```

The `pct_change` method also accepts a `freq` parameter, which lets you resample the return to an aligned period. For example, instead of passing in 22 to the `periods` argument (since not all months have 22 trading days), you can pass in `ME`:

```
df.index = pd.to_datetime(df.index)
df["close"].pct_change(freq="ME").dropna()
```

The result is a DataFrame with monthly returns:

```
date
2020-07-31    0.109736
2020-08-31    0.216569
2020-09-30   -0.102526
2020-11-30    0.096400
2020-12-31    0.114574
2021-03-31   -0.044135
2021-04-30    0.076218
2021-06-30    0.102028
2021-08-31    0.044925
2021-09-30   -0.068037
2021-11-30    0.111313
2021-12-31    0.074228
2022-01-31   -0.015712
2022-02-28   -0.054066
2022-03-31    0.057473
2022-05-31   -0.056352
2022-06-30   -0.081430
2022-08-31   -0.025210
2022-09-30   -0.120977
2022-10-31    0.109551
2022-11-30   -0.033027
2023-01-31    0.153674
2023-02-28    0.023183
2023-03-31    0.118649
2023-05-31    0.046613
2023-06-30    0.094330
Name: Adj Close, dtype: float64
```

Figure 2.17: Series with simple returns resampled to a monthly frequency

See also

For more on the `pct_change` method, see the following documentation:

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pct_change.html.

Measuring the volatility of a return series

Volatility plays an integral role in finance, serving as a key indicator of risk linked to a particular asset. A higher degree of volatility suggests a greater risk associated with the asset as it indicates more significant price changes and, therefore, a less predictable investment outcome.

Standard deviation is widely used as the measure of asset return volatility. It statistically quantifies the dispersion of asset returns from their mean, thus providing an effective metric for risk. When asset returns exhibit a larger standard deviation, it signifies more pronounced volatility, pointing to a higher risk level. Conversely, a lower standard deviation implies that the asset returns are more stable and less likely to deviate significantly from their average, indicating a lower risk.

The standard deviation's value as a risk measure extends beyond its ability to quantify risk alone. It is a common component in calculating risk-adjusted returns that provides a more nuanced evaluation of investment performance by considering the risk taken to achieve the returns. The Sharpe ratio, for instance, is a popular risk-adjusted return measure that divides the excess return of an investment (over the risk-free rate) by its standard deviation.

How to do it...

We'll use AAPL's historic stock price to compute the volatility. Here's how it's done:

1. Import the necessary libraries:

```
import numpy as np
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Load stock price data from the OpenBB Platform:

```
df = obb.equity.price.historical(
    "AAPL",
    start_date="2020-01-01",
    provider="yfinance"
)
```

3. Grab the series of adjusted close prices:

```
close = df["close"]
```

4. Compute the daily simple return:

```
returns = close.pct_change()
```

5. Calculate the daily standard deviation of returns:

```
std_dev = returns.dropna().std()
```

6. Annualize the standard deviation assuming 252 trading days in a year:

```
annualized_std_dev = std_dev * np.sqrt(252)
```

The result is a float that represents the annualized volatility over the entire period.

How it works...

The multiplication that's done by the square root of 252 in the calculation of standard deviation is based on the concept of scaling in statistics. Scaling is used when transferring a data point from one timescale to another. The number 252 is used because there are typically 252 trading days in a year. In statistical terms, variance, which is the square of standard deviation, is additive over time. This means that if you want to calculate the variance over a certain period, you can add up the variances for each sub-period. However, the standard deviation is not additive, but its square (that is, variance) is. To convert daily standard deviation into annual standard deviation, we must square it to get the variance, multiply this variance by 252 (the typical number of trading days in a year) to annualize, and then take the square root to revert to standard deviation. This process ensures we respect the additive property of variance while transitioning from daily to annual measure. By taking the square root of 252, we're essentially projecting the daily volatility over a yearly timeframe under the assumption of independent daily returns.

There's more...

Depending on the frequency of returns, we can annualize the volatility measure to match. For example, if you're looking over a long period, you may be dealing with monthly or quarterly returns. Alternatively, you may be working with daily data and want to resample to a monthly or quarterly period:

```
close.index = pd.to_datetime(close.index)
(
    close
    .pct_change(freq="ME")
    .dropna()
    .std()
```

```
        * np.sqrt(12)
    )
```

You can pass the `freq` argument to the `pct_change` method to generate monthly returns. Then, instead of multiplying the standard deviation by the square root of 252, you can multiply it by the square root of 12.

Similarly, for quarters, you can pass the `freq` argument and multiply the standard deviation by the square root of 4:

```
(  
    close  
    .pct_change(freq="QE")  
    .dropna()  
    .std()  
    * np.sqrt(4)  
)
```

In both cases, the result is a float that represents the annualized volatility.

Looking at historical price changes, rather than just one moment, is important to get a broader picture of how much and how often the price has moved in the past. We'll use the pandas `rolling` method to plot the annualized volatility through time:

```
(  
    close  
    .pct_change()  
    .rolling(window=22)  
    .std()  
    * np.sqrt(252)  
)  
.plot()
```

This results in the following plot:

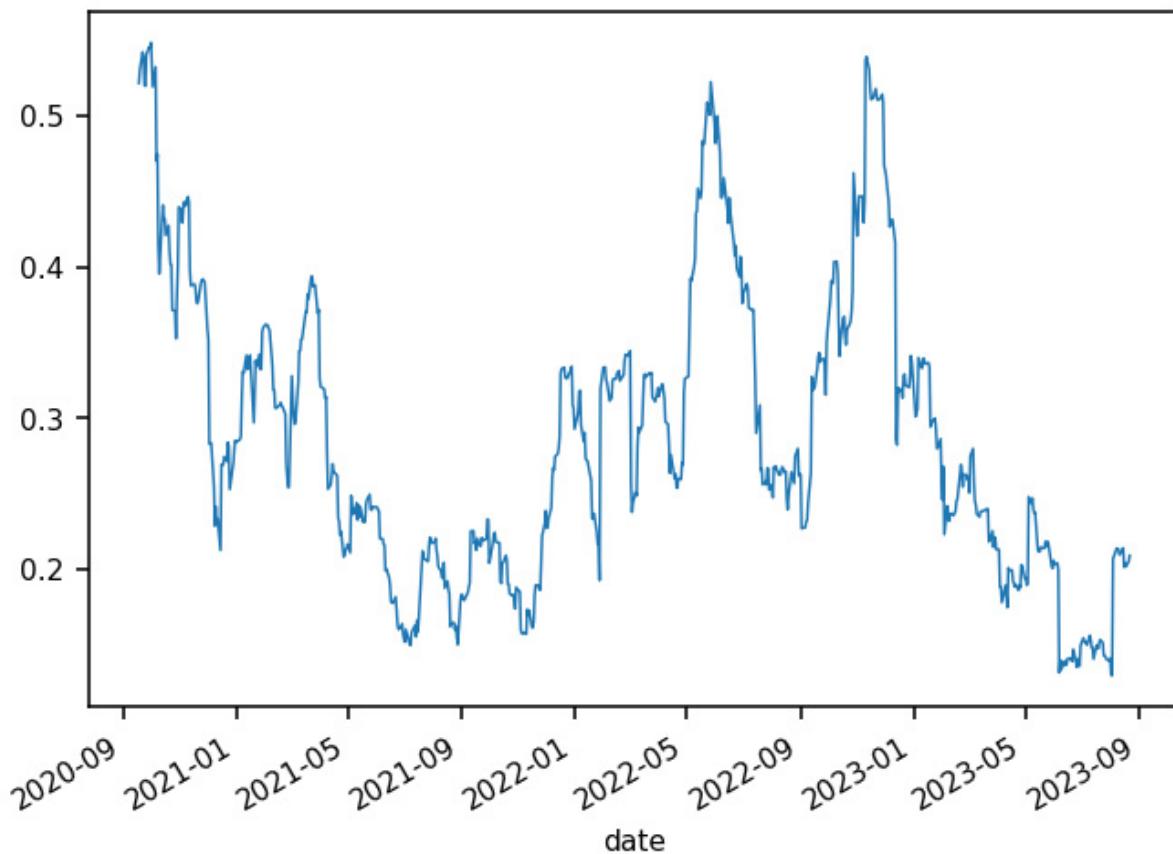


Figure 2.18: Rolling annualized volatility based on a 22-day lookback window

TIP

Rolling a method in pandas involves taking a range of data, applying a function or method that returns a single value, moving that range of data forward one step, applying the function, and repeating this. The result is a method being applied over a window of data that extends through the entire range of data.

See also

Volatility is a critical part of algorithmic trading. To learn more about volatility, take a look at the following article, which describes how investors think about volatility for risk management:
<https://www.investopedia.com/terms/v/volatility.asp>.

Generating a cumulative return series

Cumulative returns quantify the total change in the value of an investment over a specific period. To compute the cumulative return of a series of simple single-period returns, you need to add 1 to each

return, then multiply these results together, and finally subtract 1 from the product. Recall the formula for the simple return:

$$R_t = P_t - P_{t-1} / P_{t-1} = P_t / P_{t-1} - 1$$

Upon writing $(P_t / P_{t-1}) = (P_t / P_{t-1})(P_{t-1} / P_{t-2})$, the two-period return can be expressed as follows:

$$\begin{aligned} R(2) &= (P_t / P_{t-1})(P_{t-1} / P_{t-2}) - 1 \\ &= (1 + R_t)(1 + R_{t-1}) - 1 \end{aligned}$$

To compute the cumulative return of a series of continuously compounded single-period returns, you need to sum up all the single-period returns. This is because, in the case of continuously compounded returns, the logarithmic returns are additive. To illustrate this, consider a two-period compound return, which is defined as follows:

$$r(2) = \ln(1 + R(2)) = \ln(P_t / P_{t-1}) = \ln(P_t) - \ln(P_{t-1}) = p_t - p_{t-1}$$

Taking the exponent of both sides and rearranging the preceding formula gives us the following output:

$$p_t = p_{t-2} e^{r(2)}$$

Here, $r(2)$ is the compounded growth rate of prices between periods $t - 2$ and t . Using $P_t / P_{t-2} = (P_t / P_{t-1})(P_{t-1} / P_{t-2})$ and the fact that $\ln(xy) = \ln(x) + \ln(y)$, we get the following:

$$\begin{aligned} r(2) &= \ln((P_t / P_{t-1})(P_{t-1} / P_{t-2})) \\ &= \ln(P_t / P_{t-1}) + \ln(P_{t-1} / P_{t-2}) \\ &= r_t + r_{t-1} \end{aligned}$$

Getting ready...

We assume that you've followed the instructions from the previous recipes and have a Series called `close` with a `DatetimeIndex` object as the index.

How to do it...

We'll use pandas to compute the cumulative sum of simple and compound returns.

The cumulative sum of simple returns

We'll start with the cumulative sum of simple returns:

1. Compute the single period daily simple returns:

```
returns = close.pct_change()
```

2. Replace all **nan** values with zeros by using NumPy's **isnan** method as a mask:

```
returns[np.isnan(returns)] = 0
```

Note that this is equivalent to using pandas:

```
returns.fillna(0.0, inplace=True)
```

3. Now, add **1** to each of the daily simple returns:

```
returns += 1
```

4. Use the **cumprod** method to build the cumulative product of returns and subtract **1**:

```
cumulative_returns = returns.cumprod() - 1
```

5. Plot the result:

```
cumulative_returns.plot()
```

The result is the following plot:

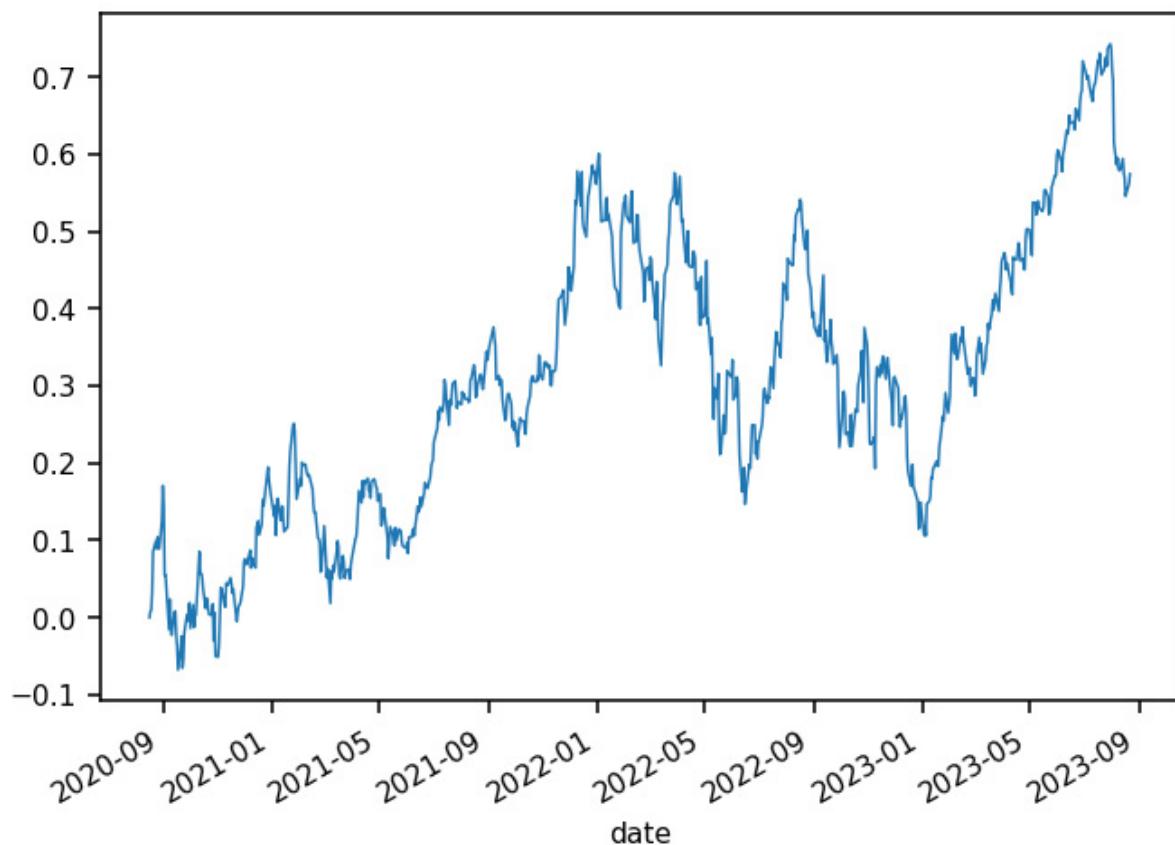


Figure 2.19: Cumulative simple returns of AAPL

The cumulative sum of compound returns

Next, we'll build the cumulative sum of compound returns:

1. Compute the single-period compound returns:

```
log_returns = np.log(close / close.shift())
```

2. Use the pandas `cumsum` method to build the cumulative product of returns:

```
cumulative_log_returns = log_returns.cumsum()
```

3. Plot the result:

```
cumulative_log_returns.plot()
```

The result is the following plot:

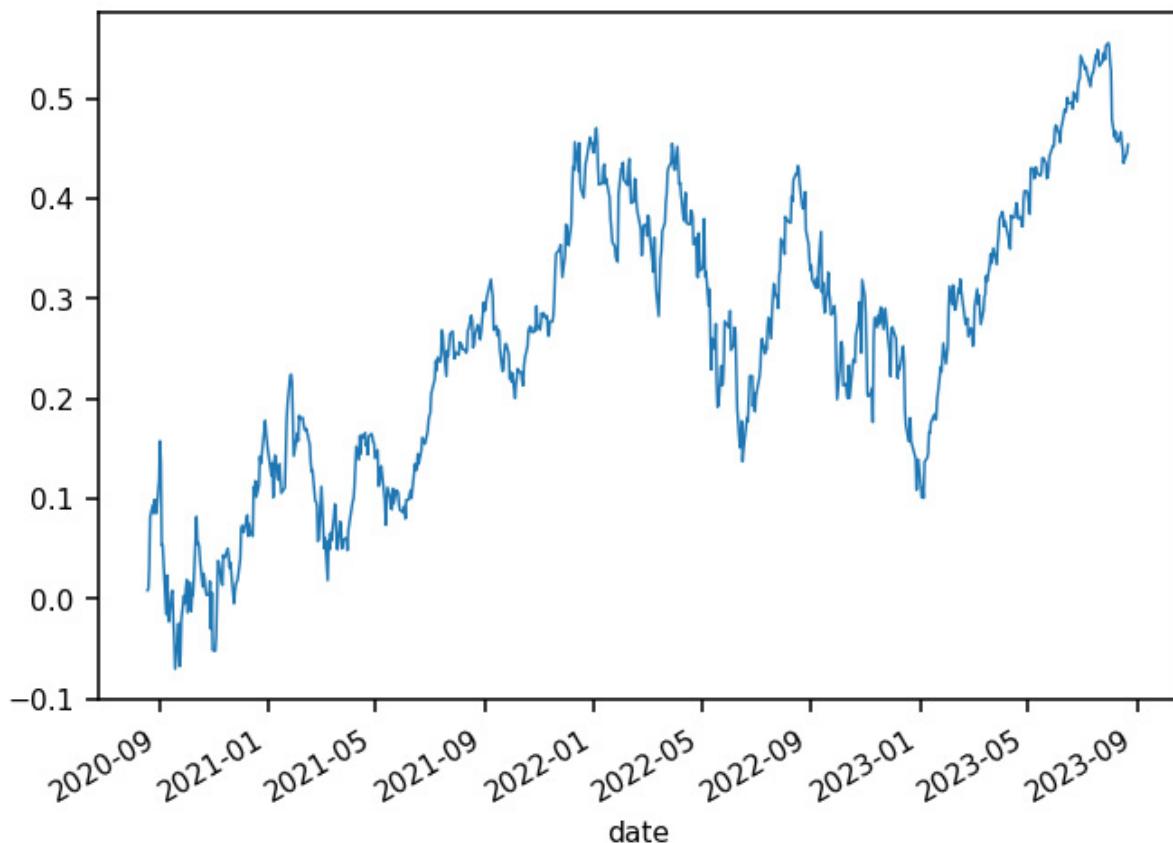


Figure 2.20: Cumulative compound returns of AAPL

Note that simple returns and compound returns will not be equal. Compound returns take into consideration returns earned on prior returns, while simple returns do not.

How it works...

`isnan` checks each value in the `returns` Series to see if it is a `nan` value. `returns[np.isnan(returns)] = 0` takes all the positions in the `returns` Series where the value is `nan` and replaces those `nan` values with `0`. To compute cumulative simple returns, we first need to add `1` to every single period return. This can be accomplished by incrementing every value in the `returns` Series by `1` with `returns += 1`. Finally, we generate the cumulative product using the `cumprod`. It calculates the cumulative product of the Series, meaning it multiplies the values in the Series together and keeps track of the result. For example, if you have a Series with the numbers `1, 2, 3, and 4`, the `cumprod` method will create a new Series with the values `1, 2 (1 x 2), 6 (1 x 2 x 3), and 24 (1 x 2 x 3 x 4)`. Finally, we subtract `1` from each value in the cumulative series, compute the compound returns, and call `cumsum` to generate the cumulative sum of the values. Instead of multiplying the values together via `cumprod`, `cumsum` adds them.

See also

While computing returns may seem trivial, it's important to understand the difference between simple returns and compound returns. You can learn more about them, along with the methods we used to compute them, by taking a look at the following documentation:

- A deep dive into the mathematics behind simple and compound returns (and a lot more):
<https://bookdown.org/compfinzbook/introcompfinr/AssetReturnCalculations.html#continuously-compounded-returns>
- Documentation on the pandas `cumsum` method: <https://pandas.pydata.org/docs/reference/api/pandas.Series.cumsum.html>
- Documentation on the pandas `cumprod` method: <https://pandas.pydata.org/docs/reference/api/pandas.Series.cumprod.html>

Resampling data for different time frames

Two types of resampling are upsampling, where data is converted into a higher frequency (such as daily data to hourly data), and downsampling, where data is converted into a lower frequency (such as daily data to monthly data). In financial data analysis, resampling can help in various ways. For instance, if you have daily stock prices, you can resample this data to calculate monthly or yearly average prices, which can be useful for long-term trend analysis. A common use case is when aligning trade and quote data. There are a lot more quotes than trades – often an order of magnitude more – and we may need to align the open, high, low, and closing quote prices to the open, high, low, and closing trade data. Since the quotes and trades will have different timestamps, resampling to a 1-second resolution is a great way to align these disparate data sources.

How to do it...

We'll work on resampling stock price data from one period to another:

1. Import the necessary libraries:

```
import numpy as np
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Use the OpenBB Platform to download intraday data at 1-minute intervals:

```
df = obb.equity.price.historical(
    "AAPL",
    interval="1m",
    provider="yfinance"
)
```

3. Resample the 1-minute resolution adjusted close data to hourly:

```
resampled = df.resample(rule="h") ["close"]
```

4. Grab the first value within the resampled time interval:

```
resampled.first()
```

Here's the output:

```
date
2023-07-12 09:00:00    189.910004
2023-07-12 10:00:00    190.619995
2023-07-12 11:00:00    189.716003
2023-07-12 12:00:00    188.897293
2023-07-12 13:00:00    189.429993
...
2023-07-18 11:00:00    193.489899
2023-07-18 12:00:00    193.130005
2023-07-18 13:00:00    192.660004
2023-07-18 14:00:00    192.985001
2023-07-18 15:00:00    193.669907
Freq: H, Name: Adj Close, Length: 151, dtype: float64
```

Figure 2.21: Resampled Series with the first value in the interval

5. Now, grab the last value:

```
resampled.last()
```

Here's the output:

```

date
2023-07-12 09:00:00    190.710007
2023-07-12 10:00:00    189.669907
2023-07-12 11:00:00    188.639893
2023-07-12 12:00:00    189.360992
2023-07-12 13:00:00    189.919998
...
2023-07-18 11:00:00    193.199997
2023-07-18 12:00:00    192.520004
2023-07-18 13:00:00    192.985001
2023-07-18 14:00:00    193.789902
2023-07-18 15:00:00    193.729996
Freq: H, Name: Adj Close, Length: 151, dtype: float64

```

Figure 2.22: Resampled Series with the last value in the interval

- Compute the mean value within the resampled time interval:

```
resampled.mean()
```

Here's the output:

```

date
2023-07-12 09:00:00    190.771767
2023-07-12 10:00:00    190.389116
2023-07-12 11:00:00    189.314387
2023-07-12 12:00:00    189.224929
2023-07-12 13:00:00    189.812174
...
2023-07-18 11:00:00    193.254420
2023-07-18 12:00:00    192.732713
2023-07-18 13:00:00    192.905636
2023-07-18 14:00:00    193.604533
2023-07-18 15:00:00    193.984174
Freq: H, Name: Adj Close, Length: 151, dtype: float64

```

Figure 2.23: Resampled Series with the mean value in the interval

- Generate the open, high, low, and closing values for the resampled time interval:

```
resampled.ohlc()
```

Here's the output:

	open	high	low	close
date				
2023-07-12 09:00:00	189.910004	191.625000	189.570007	190.710007
2023-07-12 10:00:00	190.619995	190.914993	189.662003	189.669907
2023-07-12 11:00:00	189.716003	189.889999	188.489899	188.639893
2023-07-12 12:00:00	188.897293	189.759995	188.580002	189.360992
2023-07-12 13:00:00	189.429993	190.009995	189.379898	189.919998
...
2023-07-18 11:00:00	193.489899	193.637802	192.825302	193.199997
2023-07-18 12:00:00	193.130005	193.130005	192.434998	192.520004
2023-07-18 13:00:00	192.660004	193.160004	192.585007	192.985001
2023-07-18 14:00:00	192.985001	193.949905	192.985001	193.789902
2023-07-18 15:00:00	193.669907	194.229996	193.570007	193.729996

Figure 2.24: Resampled Series with the open, high, low, and closing values in the interval

How it works...

The `resample` method is used for time series data resampling and alters the frequency. By specifying a time frequency string, such as `D` for daily or `M` for monthly, we can decide on the new sampling rate. This method creates a `Resampler` object, upon which aggregation or transformation functions such as `mean` or `sum` can be applied. The resulting DataFrame or Series reflects the data that's consolidated or redistributed to the desired frequency. There's also a choice in how the boundaries of the bins are defined. The `closed` parameter can dictate which side of the bin interval is closed, either `right` or `left`. Moreover, the `label` parameter specifies which bin edge label to use for labeling the aggregation result.

The `resample` method has several useful arguments:

- `rule`: This is the offset string or object representing target conversion.
- `axis`: This specifies the axis to be resampled. By default, it's `0` (index).
- `closed`: This specifies which side of the bin interval is closed. The options are `right` or `left`.
- `label`: This specifies which bin edge label to label the bucket with. The options are `right` or `left`.
- `convention`: This is used when resampling period data (a time series with `PeriodIndex`) with the `start` or `end` option. The default is `end`.

- **kind**: This is used when upsampling from low to high frequency. The options are `timestamp` or `period`. By default, `timestamp` is used.
- **loffset**: This adjusts the resampled time labels.
- **base**: This is used for adjusting the start of the bins to a different timestamp.
- **on**: This is used for a DataFrame, to resample based on the time of a particular column rather than the DataFrame index.
- **level**: This is used for a `MultiIndex` DataFrame, to resample based on the time of a particular level of the `MultiIndex` object.
- **origin**: This defines the origin of the adjusted timestamps.
- **offset**: This adjusts the start of the bins based on this time delta.

There's more...

The `resample` and `asfreq` methods in pandas are both used to change the frequency of time series data, but they serve slightly different purposes and can be used in different scenarios. The `resample` method is primarily used for downsampling, where it provides different ways to aggregate the data for the new frequency (such as taking the mean, sum, maximum, minimum, and so on). When upsampling, `resample` can also interpolate the missing values by passing an arbitrary function to perform binning over a Series or DataFrame object in bins of arbitrary size.

The `asfreq` method is used when you want to convert a time series into a specified frequency. It does not provide any aggregation or transformation – it simply changes the frequency of the data. If you're upsampling (increasing the frequency), `asfreq` will introduce `nan` values for the newly created data points. If you're downsampling (decreasing the frequency), `asfreq` will drop the data points that don't fit into the new frequency.

pandas offers over 40 offsets that we can use with both `resample` and `asfreq`. They allow for even more flexibility than passing in `D` or `H` (for day or minute). You can see a full list by running

```
pd.offsets._all_
```

Downsample the minute data to daily and adjust the labels in the index so that they match the frequency.

```
df.asfreq("D").to_period()
```

The result is a DataFrame. Note that the time portion of the index is dropped. Also, note the `nan` values on the days when there is no market data:

	Open	High	Low	Close	Adj Close	Volume
date						
2023-08-16	177.130005	177.639999	177.000000	177.494995	177.494995	3608898.0
2023-08-17	177.139999	177.505402	177.070007	177.339996	177.339996	2969302.0
2023-08-18	172.300003	173.089996	171.960007	172.945007	172.945007	4147338.0
2023-08-19	NaN	NaN	NaN	NaN	NaN	NaN
2023-08-20	NaN	NaN	NaN	NaN	NaN	NaN
2023-08-21	175.070007	175.119995	174.660004	174.731995	174.731995	1938994.0
2023-08-22	177.059998	177.085007	176.580002	176.602997	176.602997	1789325.0

Figure 2.25: DataFrame with frequency changed from 1 minute to calendar days

The `asfreq` method incorporates a `method` argument, directing pandas on how to handle `nan` values. This argument permits either `backfill` to populate `nan` values using the preceding valid observation or `pad` to populate with the subsequent valid observation. Additionally, the `fill_value` parameter lets us specify a custom value for filling in missing entries during the upsampling process.

If we want to avoid including days with no market prices, we can use the business day offset:

```
df.asfreq(pd.offsets.BDay())
```

The result is a DataFrame that only includes valid business days in the index:

	Open	High	Low	Close	Adj Close	Volume
date						
2023-08-16 09:30:00	177.130005	177.639999	177.000000	177.494995	177.494995	3608898
2023-08-17 09:30:00	177.139999	177.505402	177.070007	177.339996	177.339996	2969302
2023-08-18 09:30:00	172.300003	173.089996	171.960007	172.945007	172.945007	4147338
2023-08-21 09:30:00	175.070007	175.119995	174.660004	174.731995	174.731995	1938994
2023-08-22 09:30:00	177.059998	177.085007	176.580002	176.602997	176.602997	1789325

Figure 2.26: DataFrame with frequency changed from 1 minute to business days

IMPORTANT

`resample` applies an aggregate to the data within the selected interval (for example, `ohlc`), whereas `asfreq` changes the index and does not aggregate the data. This is apparent in Figure 2.23, where the time portion of the index is displayed. The values that match the given index (for example, `2023-07-12 09:30:00`) are selected for inclusion in the

returned DataFrame. We can see this by selecting data from the first row using `df.loc["2023-07-12 09:30:00"]`. The result is a Series with values that match those in the first row.

See also

See the documentation for more about the differences between `resample` and `asfreq`:

- Documentation on the pandas `resample` method: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.resample.html>
- Documentation on the pandas `asfreq` method: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.asfreq.html>

Addressing missing data issues

pandas is well-suited to handling missing data in time series data. In the context of financial market data, missing data can occur for various reasons:

- **Market closures:** Most financial markets aren't open 24/7. They operate on specific days and hours, closing for weekends, public holidays, or special events. If a data source tries to retrieve data when the market is closed, it might represent this as a missing value.
- **Data availability:** Not all historical data is available for every market or every security. Some markets may only have data available from a certain date, or some data may be missing due to technological issues, glitches, or errors during data recording and transmission.
- **Delisting of securities:** If a security gets delisted from a market (for example, a company going out of business), no new data is produced for that security. If your timeframe extends beyond the delisting date, missing data will be encountered.
- **Data granularity:** The level of detail in the dataset might also influence the existence of missing data. For example, if you're looking for minute-by-minute data but your source only provides hourly data, you'll have missing data for each minute that isn't the start of a new hour.

pandas has ways of handling missing data. This recipe presents those most common for algorithmic trading.

Getting ready...

We'll demonstrate several ways of filling in missing data. We'll start with AAPL's stock price data:

1. Import the necessary libraries:

```
import numpy as np
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download the stock price data:

```
df = obb.equity.price.historical(
    "AAPL",
    start_date="2020-07-01",
```

```

        end_date="2023-07-06",
        provider="yfinance",
    )

```

3. `df` only contains the trading days for AAPL, so we'll reindex the DataFrame so that it includes all calendar days between the start and end date of the time series. First, create a `DatetimeIndex` object with calendar days:

```

calendar_dates = pd.date_range(
    start=df.index.min(),
    end=df.index.max(),
    freq="D"
)

```

4. Then, reindex the DataFrame:

```
calendar_prices = df.reindex(calendar_dates)
```

5. The resulting DataFrame is populated with `nan` values for dates where there is no data:

	Open	High	Low	Close	Adj Close	Volume	Dividends	Stock Splits
2020-07-01	89.498016	90.047081	89.201423	89.250443	89.250443	110737200.0	0.0	0.0
2020-07-02	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
2020-07-03	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2020-07-04	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2020-07-05	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...
2023-07-02	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2023-07-03	193.518682	193.618553	191.501402	192.200470	192.200470	31458200.0	0.0	0.0
2023-07-04	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2023-07-05	191.311658	192.719744	190.362927	191.071976	191.071976	46920300.0	0.0	0.0
2023-07-06	189.583986	191.761054	188.944850	191.551331	191.551331	45094300.0	0.0	0.0

Figure 2.27: A DataFrame of stock prices with missing data on non-trading days

How to do it...

We'll demonstrate several examples of filling in the missing values:

1. Use the pandas `fillna` method with the `method` argument set to `bfill` to replace `nan` values with the next valid observation:

```
calendar_prices.bfill()
```

The result is a filled DataFrame:

	Open	High	Low	Close	Adj Close	Volume	Dividends	Stock Splits
2020-07-01	89.498016	90.047081	89.201423	89.250443	89.250443	110737200.0	0.0	0.0
2020-07-02	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
2020-07-03	90.694182	92.110972	90.662316	91.637894	91.637894	118655600.0	0.0	0.0
2020-07-04	90.694182	92.110972	90.662316	91.637894	91.637894	118655600.0	0.0	0.0
2020-07-05	90.694182	92.110972	90.662316	91.637894	91.637894	118655600.0	0.0	0.0
...
2023-07-02	193.518682	193.618553	191.501402	192.200470	192.200470	31458200.0	0.0	0.0
2023-07-03	193.518682	193.618553	191.501402	192.200470	192.200470	31458200.0	0.0	0.0
2023-07-04	191.311658	192.719744	190.362927	191.071976	191.071976	46920300.0	0.0	0.0
2023-07-05	191.311658	192.719744	190.362927	191.071976	191.071976	46920300.0	0.0	0.0
2023-07-06	189.583986	191.761054	188.944850	191.551331	191.551331	45094300.0	0.0	0.0

Figure 2.28: A DataFrame with missing data backfilled to the previous NaN

2. Use `ffill` to propagate the last valid observation forward:

```
calendar_prices.ffill()
```

The result is a filled DataFrame:

	Open	High	Low	Close	Adj Close	Volume	Dividends	Stock Splits
2020-07-01	89.498016	90.047081	89.201423	89.250443	89.250443	110737200.0	0.0	0.0
2020-07-02	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
2020-07-03	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
2020-07-04	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
2020-07-05	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
...
2023-07-02	191.371579	194.217727	191.002068	193.708420	193.708420	85069600.0	0.0	0.0
2023-07-03	193.518682	193.618553	191.501402	192.200470	192.200470	31458200.0	0.0	0.0
2023-07-04	193.518682	193.618553	191.501402	192.200470	192.200470	31458200.0	0.0	0.0
2023-07-05	191.311658	192.719744	190.362927	191.071976	191.071976	46920300.0	0.0	0.0
2023-07-06	189.583986	191.761054	188.944850	191.551331	191.551331	45094300.0	0.0	0.0

Figure 2.29: A DataFrame with missing data backfilled to the next NaN

How it works...

The `fillna` method in pandas has several arguments that provide different ways to handle missing values:

- **value**: This specifies the value to use to fill NA/NaN values. It could be a scalar, dictionary, or Series.
- **axis**: This determines whether to fill missing values along rows (**0** or **index**) or columns (**1** or **columns**).
- **inplace**: If set to **True**, the operation modifies the data in place. The default is **False**.
- **limit**: Limits the number of consecutive forward/backward filled values.
- **downcast**: A dictionary of **item->dtype** that specifies the type of data for each item.

There's more...

Sometimes, simple forward-filling or back-filling techniques may not suffice, especially in more complex scenarios, such as generating implied volatility surfaces for derivatives pricing. In these cases, we need a more sophisticated method of dealing with missing data. The pandas **interpolate** method uses various interpolation techniques to fill missing values, including linear, polynomial, time, and spatial interpolations. The choice of interpolation method can be adapted based on the characteristics of the data, allowing for more accurate handling of missing values in complex structures such as implied volatility surfaces.

Here's an example of using linear interpolation to fill in the missing values in the DataFrame:

```
calendar_prices.interpolate(method="linear")
```

Here's an example of using cubic spline interpolation to fill in the missing values in the DataFrame:

```
calendar_prices.interpolate(method="cubicspline")
```

In both cases, the result is a DataFrame with filled values.

See also

For more about the **fillna** method and interpolation, take a look at the following documentation:

- Documentation on the pandas **fillna** method: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html>
- Documentation on the pandas **interpolate** method:
<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html>
- More on interpolation: <https://en.wikipedia.org/wiki/Interpolation>

Applying custom functions to analyze time series data

Custom functions let us apply transformations and computations to data beyond the standard pandas methods. This flexibility becomes important when dealing with unique or complex analytical tasks.

The `apply` function accepts a custom or built-in function as an argument and applies this function across the DataFrame or Series, returning an output containing the results.

IMPORTANT

The `apply` function is notoriously slow since it operates on every row or column in a loop. Use this with caution when dealing with large DataFrames!

Getting ready...

We assume that you've followed the instructions from the previous recipes and have a DataFrame called `df` with a `DatetimeIndex` object as the index.

How to do it...

We'll use the range as our custom function for this recipe:

1. Determine the range (high price minus low price) of the stock prices using an anonymous lambda function:

```
df.apply(lambda x: x["high"] - x["low"], axis=1)
```

2. Apply a user-defined function that does the same thing:

```
def fcn(row):
    return row["high"] - row["low"]
df.apply(fcn, axis=1)
```

The result in both cases is a Series with the high minus low price:

```
date
2020-07-01    0.846799
2020-07-02    1.676423
2020-07-06    1.450613
2020-07-07    1.568424
2020-07-08    1.261619
...
2023-06-29    1.130005
2023-06-30    3.220001
2023-07-03    2.120010
2023-07-05    2.360001
2023-07-06    2.807999
Length: 758, dtype: float64
```

Figure 2.30: Series with the range of AAPL stock

3. Test the validity of the data by flagging where the closing price is less than the low price or greater than the high price:

```
df["valid"] = df.apply(
```

```

lambda x: x["low"] <= x["close"] <= x["high"], axis=1)
df[df.valid == False]

```

The result is an empty DataFrame since no values were invalid:

	Open	High	Low	Close	Adj Close	Volume	Dividends	Stock Splits	range	valid
date										

Figure 2.31: Empty DataFrame showing now records where the closing price is outside the high and low of the day

How it works...

The pandas `apply` function can accept an anonymous lambda function or a user-defined function. The code for computing the range of stock prices using a lambda is equivalent to the code using a user-defined function. We pass a 1 to the `axis` argument to apply the calculation along each row.

You can expand the use of the `apply` function with its other arguments:

- `func`: The function to apply to each row or column of the DataFrame
- `axis`: Whether to apply the function to each row (0 or `index`) or column (1 or `columns`)
- `broadcast`: Whether to broadcast the output of the function back onto the DataFrame
- `raw`: If `True`, the function receives arrays instead of Series objects or DataFrames
- `reduce`: If `True`, try to apply reduction procedures
- `result_type`: Controls the type of output: `expand`, `reduce`, `broadcast`, or `None`
- `args`: Additional positional arguments to pass to `func`

There's more...

The function that's passed to the `apply` method may accept additional arguments. Let's say we're interested in calculating the price range between two columns but we want to ignore the price ranges that are below a certain threshold. Our user-defined function needs to accept the two columns for which we compute the range and the threshold value.

First, we'll define the function that calculates the price range. This function will take four arguments: a row from the DataFrame, a `high` column, a `low` column, and a threshold for the minimum range:

```

def calculate_range(row, high_col, low_col, threshold):
    range = row[high_col] - row[low_col]
    return range if range > threshold else np.nan

```

IMPORTANT NOTE

When the pandas `apply` function is invoked with `axis=1`, it applies the specified function to each row of the DataFrame. Each row is treated as a pandas Series object, with the index of the Series being the column names of the DataFrame. This Series object, representing the entire row, is passed as an argument to the function you've defined.

This function computes the difference between the high and low prices. If this difference is greater than the threshold, it returns the difference; otherwise, it returns `nan`.

Next, we can use the `apply` method to apply this function to the DataFrame. The `args` parameter of `apply` allows us to pass extra arguments to the function:

```
threshold = 1.5
df["range"] = df.apply(
    calculate_range,
    args=("high", "low", threshold),
    axis=1
)
df.range
```

The result of this operation is a new column in the DataFrame that contains the computed price range if it's above the threshold, or `nan` otherwise:

```
date
2020-07-17      NaN
2020-07-20      2.437500
2020-07-21      2.507500
2020-07-22      NaN
2020-07-23      5.067497
...
2023-07-17      2.510010
2023-07-18      1.910004
2023-07-19      5.580002
2023-07-20      3.970001
2023-07-21      3.740005
Name: range, Length: 758, dtype: float64
```

Figure 2.32: Series with `NaN` values where the difference between the high and low price does not exceed a threshold

See also

See the following documentation for more on the `apply` method:

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pct_change.html

3

Visualize Financial Market Data with Matplotlib, Seaborn, and Plotly Dash

The first step when working with data is to visualize and explore it. This is especially true when dealing with financial market data we rely on for trading. This chapter sets the stage by introducing five powerful data visualization techniques: pandas, Matplotlib, Seaborn, Plotly, and Plotly Dash.

Each tool has pros and cons and should be selected depending on the use case. pandas has built-in plotting functionality using both Matplotlib and Plotly to render the charts. Matplotlib offers advanced functionality for building 3-dimensional surfaces and animated charts. Seaborn offers an array of statistical data visualizations. Plotly works with JavaScript for interactive charting. Plotly Dash is a framework for building interactive web apps with Python.

By the end of the chapter, you'll have a wide range of tools and chart types to visually inspect the financial market data required to research and build algorithmic trading applications.

In this chapter, we present the following recipes:

- Quickly visualizing data using pandas
- Animating the evolution of the yield curve with Matplotlib
- Plotting options implied volatility surfaces with Matplotlib
- Visualizing statistical relationships with Seaborn
- Creating an interactive PCA analytics dashboard with Plotly Dash

Quickly visualizing data using pandas

pandas is an all-purpose data manipulation library. Not only can you use it for data acquisition and manipulation as we saw in [*Chapter 1, Acquire Free Financial Market Data with Cutting-edge Python Libraries*](#) and [*Chapter 2, Analyze and Transform Financial Market Data with pandas*](#), but you can use it for plotting too. pandas offers various “backends” that are used while plotting through a common method. In this recipe, you'll learn how to use the default backend, Matplotlib, to quickly plot financial market data using a line plot, bar chart, histogram, and others.

How to do it...

You can use the Matplotlib plots through pandas by importing them.

1. Import the libraries:

```
import matplotlib as plt
import pandas as pd
from openbb import obb
from pandas.plotting import bootstrap_plot, scatter_matrix
obb.user.preferences.output_type = "dataframe"
```

2. Download stock price data:

```
df = obb.equity.price.historical("AAPL", provider="yfinance")
```

3. Create a line chart that plots the closing price:

```
df.close.plot()
```

The result is the following chart:

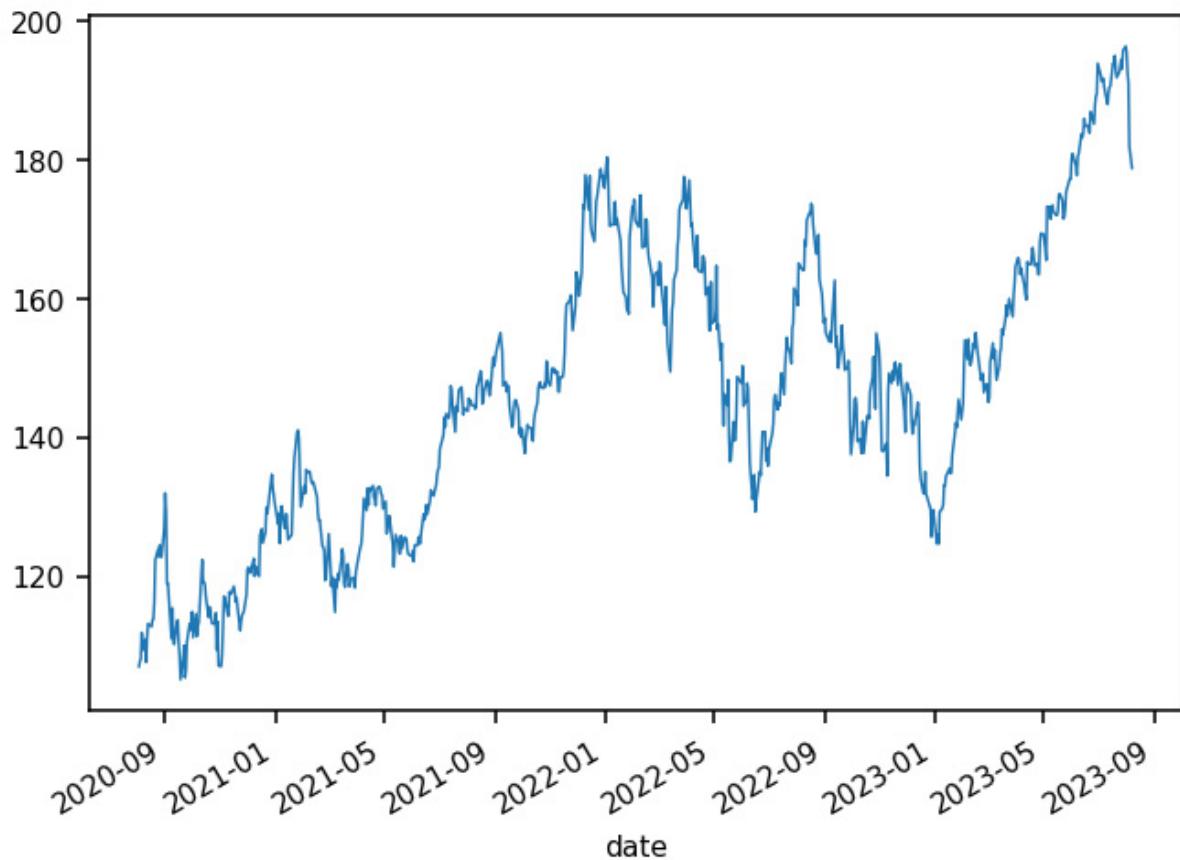


Figure 3.1: Line chart plotting AAPL's unadjusted closing price between 2020 and 2023.

4. Plot the daily returns as a bar chart using additional options to style the chart:

```
returns = df.close.pct_change()
returns.name = "return"
returns.plot.bar(
    title="AAPL returns",
    grid=False,
```

```
    legend=True,  
    xticks=[]  
)
```

The result is the following chart:

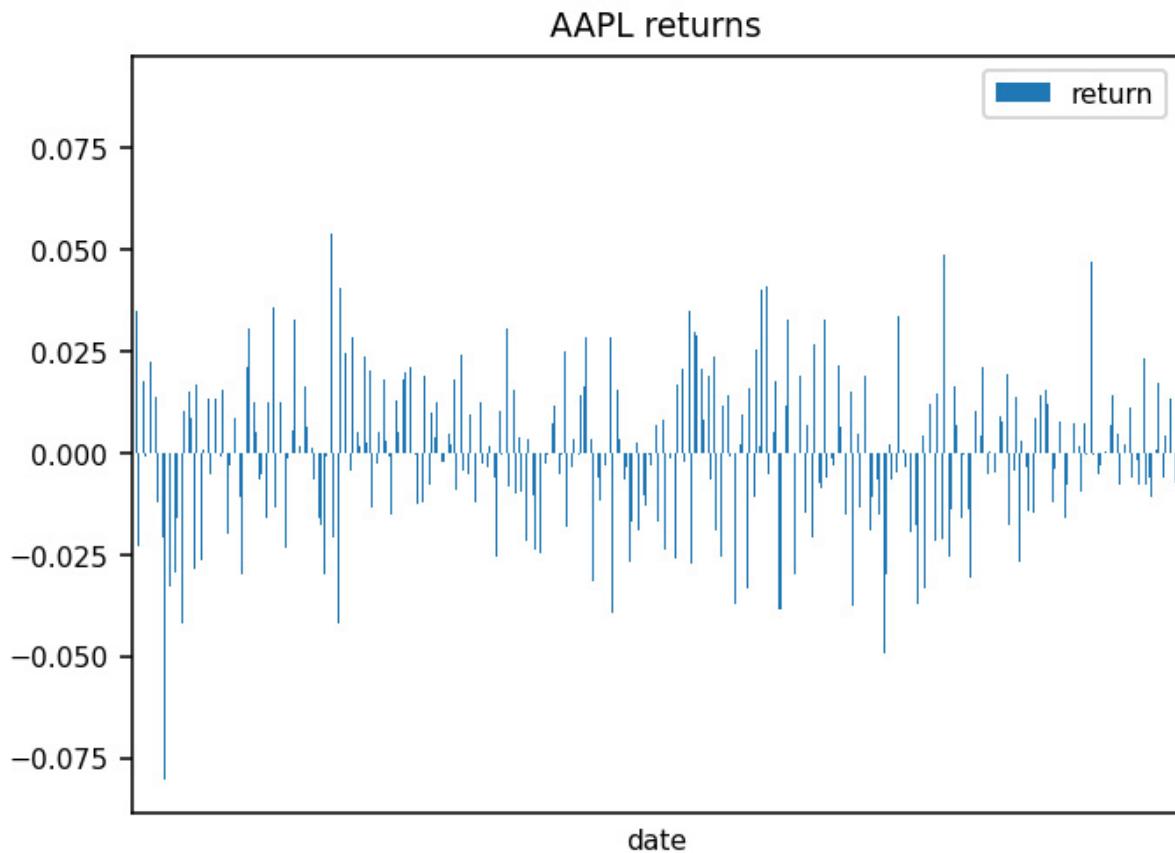


Figure 3.2: Bar chart plotting AAPL's daily returns without the x-axis labels, grid, or legend.

5. Create a histogram of returns with 50 bins:

```
returns.plot.hist(bins=50)
```

The result is the following chart:

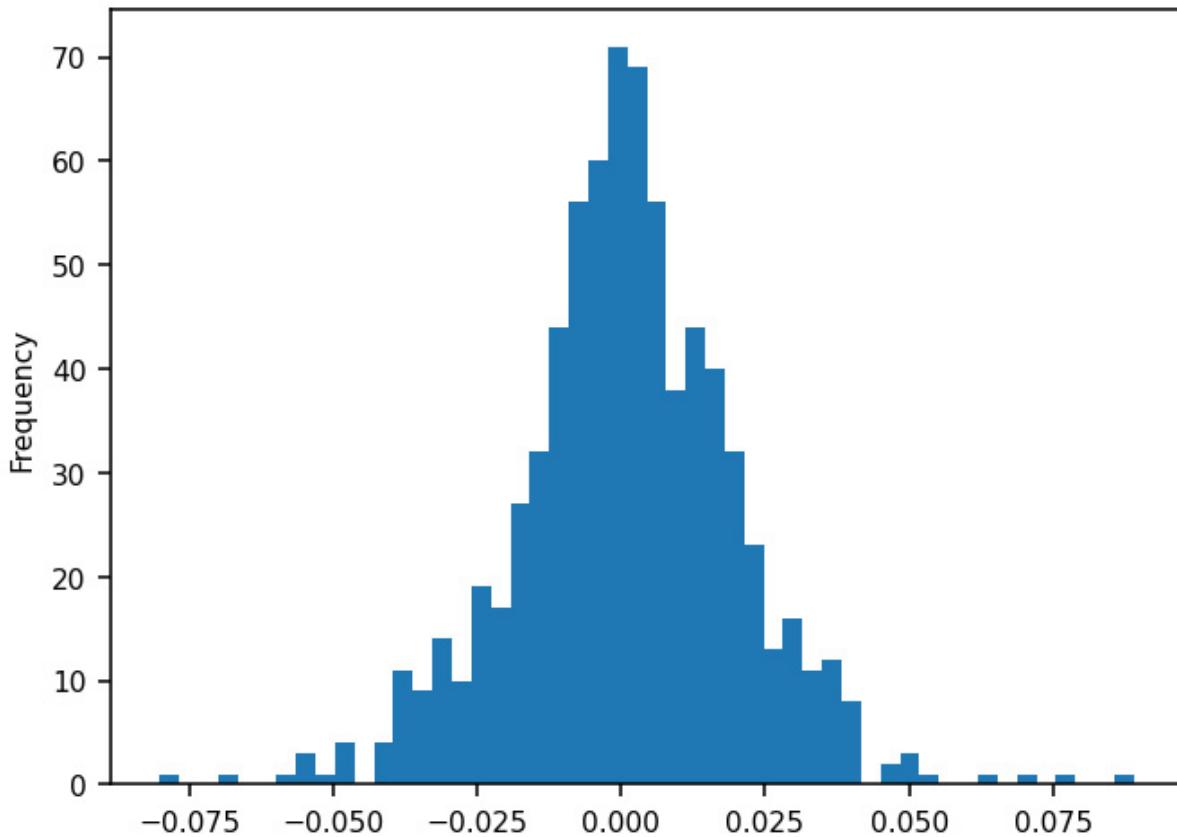


Figure 3.3: Histogram of AAPL daily returns with 50 bins.

6. Create a box-and-whisker plot:

```
returns.plot.box()
```

The result is the following chart:

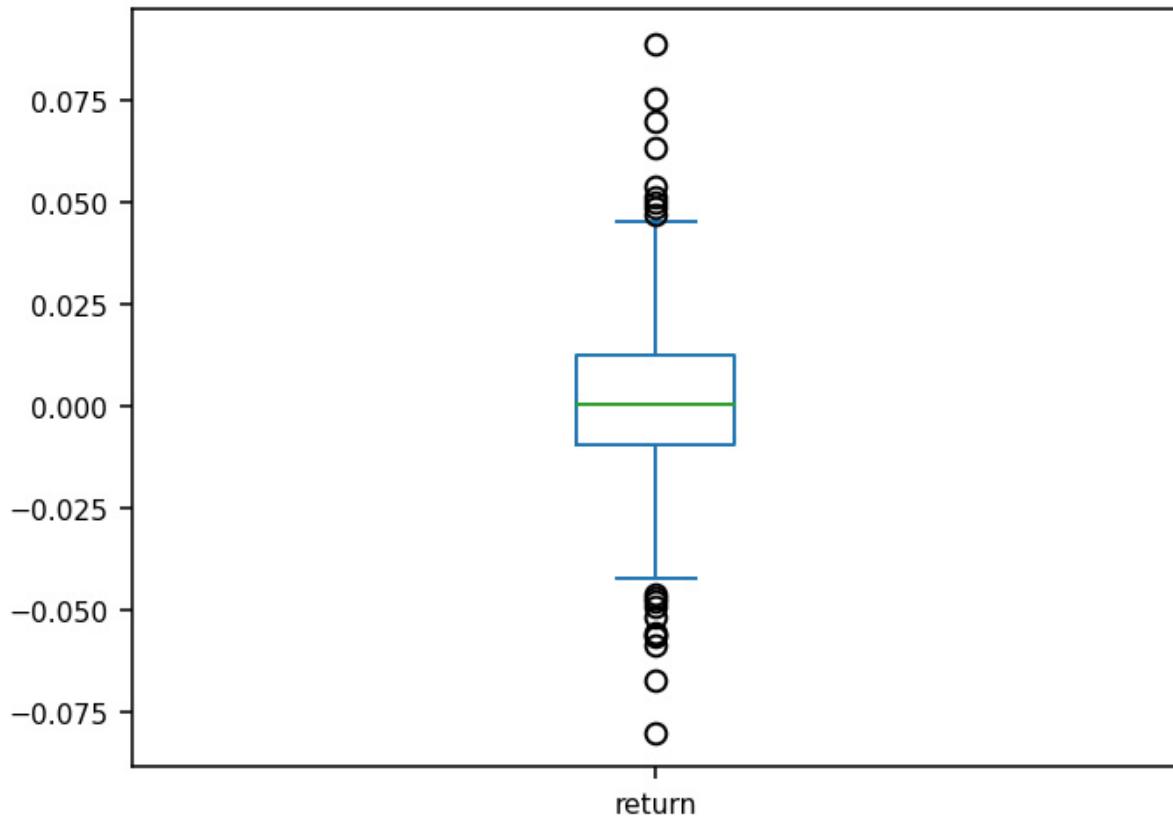


Figure 3.4: Box-and-whisker plot showing the median, quartiles, and outliers of AAPL daily returns.

IMPORTANT

A box plot serves as a graphical representation for displaying numerical data distributions via their quartiles. The box itself spans from the first quartile (Q1) to the third quartile (Q3), with a line indicating the median (Q2). Whiskers extend from the box boundaries to illustrate the data's range, and their default position is determined by 1.5 times the interquartile range (IQR), where IQR equals Q3 minus Q1. Data points beyond the whiskers are considered outliers.

How it works...

The pandas `plot` method lets you create various types of plots using DataFrames and Series. Under the hood, pandas use the defined backend (Matplotlib by default) to generate these visualizations. To use this method, you first create a DataFrame or Series, then call the `plot` method on the object, optionally specifying the type of plot and other parameters that control the plot's appearance such as color, size, title, and axes labels.

IMPORTANT

Using the `plot` method, we can generate bar plots, density plots, scatter plots, and many others. We can define the type of plot using the `kind` parameter. Not all backends support all plot types. For example, the plot type `hexbin` does not work with the Plotly backend. See the pandas documentation for details.

There's more...

A common step in quantitative portfolio construction and risk management is analyzing the relationship between two or more assets. Scatter plots are a type of visual representation that can be used to explore the relationship between two stocks. Each dot on the scatter plot represents a pair of values for the two stocks at a specific point in time. The x-coordinate of the dot represents the value of one stock, and the y-coordinate represents the value of the other stock.

If the dots form a pattern that slants upwards from left to right, it suggests a positive correlation between the stocks, meaning as one stock's price increases, the other's tends to as well. If the plot forms a pattern slanting downwards, it suggests a negative correlation, meaning as one stock's price increases, the other's tends to decrease. If the dots appear randomly scattered with no discernible pattern, it indicates no or a weak correlation.

Compare AAPL with the Nasdaq tracking ETF, QQQ.

```
qqq = obb.equity.price.historical("QQQ", provider="yfinance")
qqq_returns = qqq.close.pct_change()
asset_bench = pd.concat([returns, qqq_returns], axis=1)
asset_bench.columns = ["AAPL", "QQQ"]
asset_bench.plot.scatter(x="QQQ", y="AAPL", s=0.25)
```

The result is a scatter plot that shows the seemingly positive relationship between AAPL returns and QQQ returns.

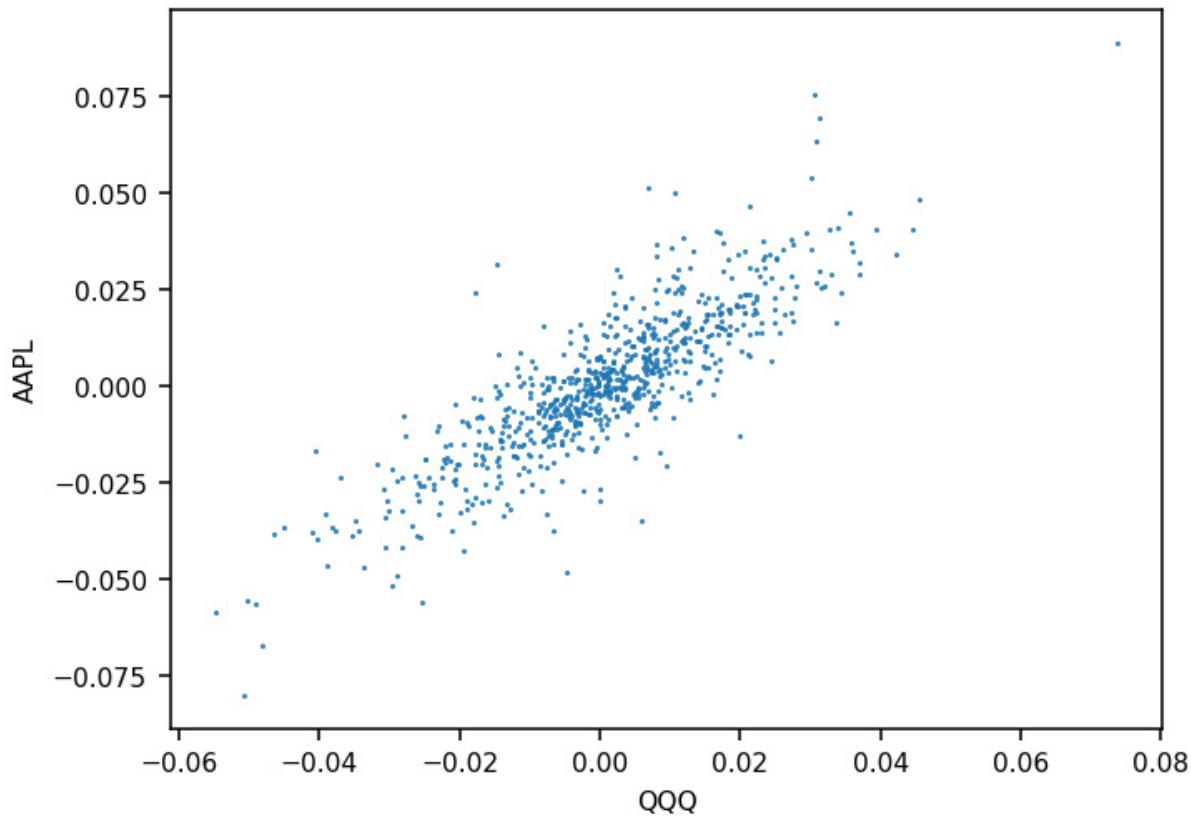


Figure 3.5: Scatter plot of AAPL daily returns and QQQ daily returns showing a positive linear relationship.

The pandas `scatter_matrix` function visualizes pairwise relationships. It generates a matrix of scatter plots, each plotting a pair of columns against each other. This allows for a quick visual inspection of potential correlations or patterns within your data. Additionally, the main diagonal (from top left to bottom right) shows the histogram of each column, which helps to visualize data distribution.

```
scatter_matrix(asset_bench)
```

The result is the scatter matrix.

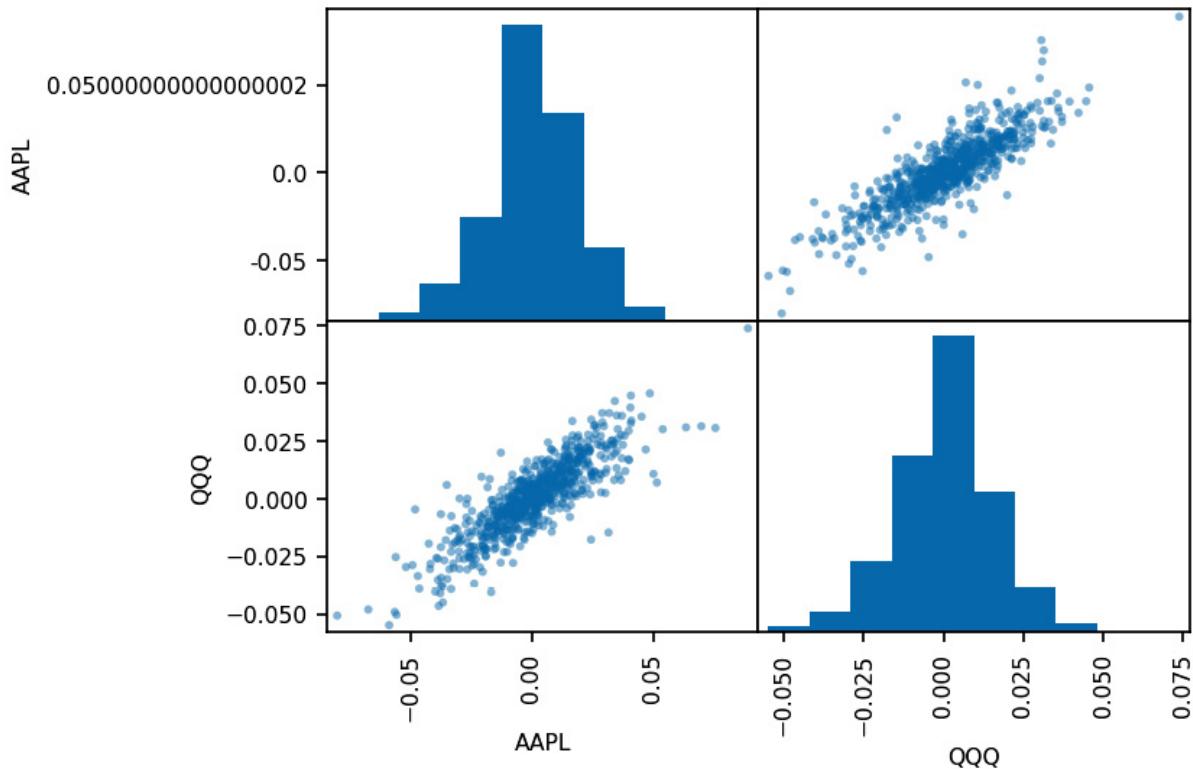


Figure 3.6: A scatter matrix plot summarizing AAPL and QQQ returns in one chart.

Bootstrap plots serve to graphically evaluate the variability associated with a particular statistic, including the mean, median, and midrange. A designated subset size is randomly sampled from the data set, and the target statistic is calculated for this subset. This procedure is iteratively performed a predetermined number of times. The plots and histograms collectively form the bootstrap plot.

```
bootstrap_plot(returns)
```

By default, `bootstrap_plot` selects 50 data points to consider during each sampling. The result is a `bootstrap_plot`.

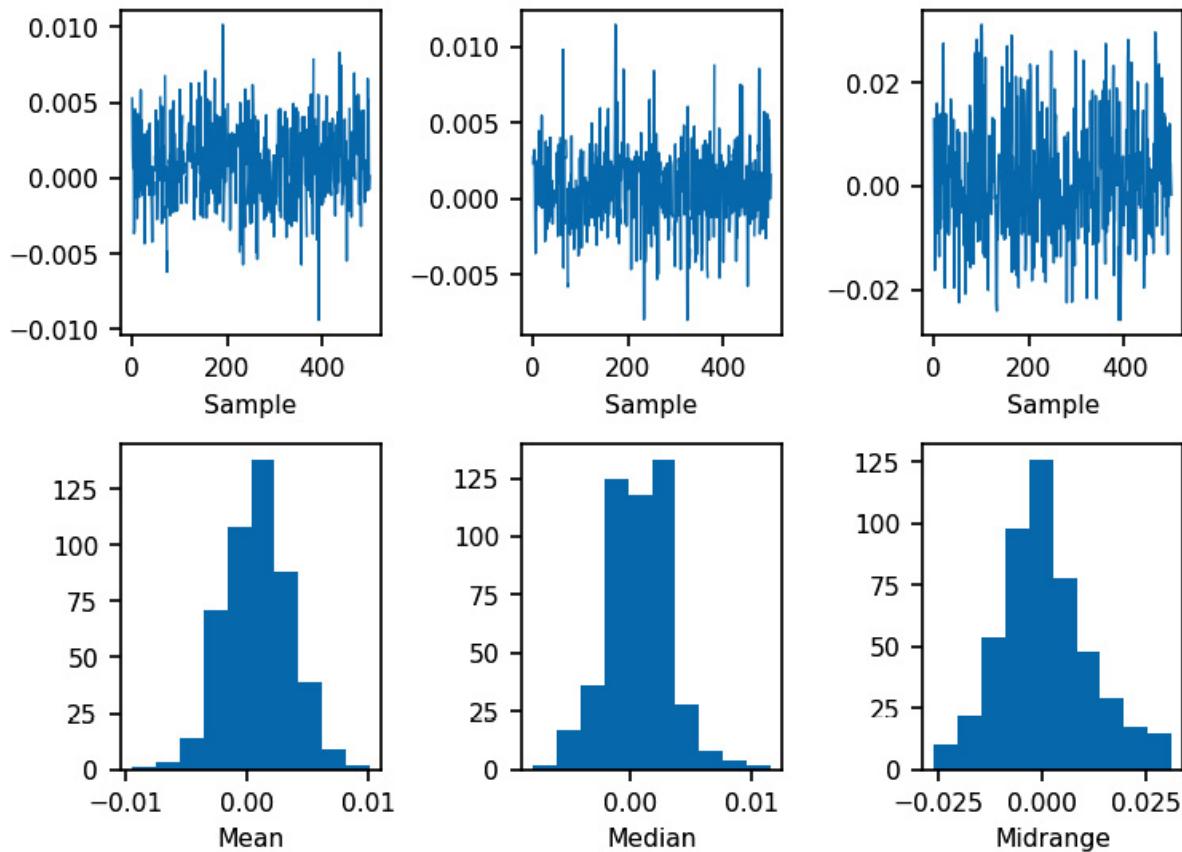


Figure 3.7: A bootstrap plot that depicts random subsets of data.

See also

pandas has detailed documentation on its plots. You can read more here:

- pandas plot documentation: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.html>.
- pandas `scatter_matrix` documentation: https://pandas.pydata.org/docs/reference/api/pandas.plotting.scatter_matrix.html.
- pandas `bootstrap_plot` documentation: https://pandas.pydata.org/docs/reference/api/pandas.plotting.bootstrap_plot.html.

To see more practical examples of using pandas plotting, you can check out past issues of the PyQuant Newsletter here: <https://www.pyquantnews.com/past-pyquant-newsletter-issues>.

Animating the evolution of the yield curve with Matplotlib

Even though pandas use Matplotlib as its backend, there are times you need to go deeper. One such case is when you want to visualize the change in data through time—like when analyzing the

evolution of the yield curve. The yield curve, which charts the yields of bonds of the same quality across different maturities, typically slopes upward. This means that longer-term bonds have higher yields than shorter-term bonds, which makes sense given the additional risks associated with holding a bond for a longer time (e.g., inflation, higher interest rate volatility). However, there are times when the yield curve inverts, meaning that shorter-term bonds yield more than longer-term ones. Many traders and economists view an inverted yield curve as a precursor to a recession.

An inverted yield curve has historically preceded U.S. recessions, suggesting traders' anticipation of lower future interest rates and a coming economic slowdown. The inversion can constrict bank profitability, leading to reduced lending and a potential economic deceleration. Additionally, the expectation of a recession can become a self-fulfilling prophecy as businesses and consumers curtail spending. Investors may also shift towards safer assets, limiting funding for riskier ventures.

How to do it...

Creating an animated plot requires a specialized function called `animation` which is imported from Matplotlib.

1. Import the libraries:

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import animation
from mpl_toolkits.mplot3d import Axes3D
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Create a list of maturities and download the data:

```
maturities = ["3m", "6m", "1y", "2y", "3y", "5y", "7y", "10y",
              "30y"]
data = obb.fixedincome.government.treasury_rates(
    start_date="1985-01-01",
    provider="federal_reserve",
).dropna(how="all").drop(columns=["month_1", "year_20"])
data.columns = maturities
```

3. Use boolean indexing to mark where the yield curve is inverted:

```
data["inverted"] = data["30y"] < data["3m"]
```

4. Initialize the figure:

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
line, = ax.plot([], [])
```

5. Set the range of ticks

```
ax.set_xlim(0, 8)
ax.set_ylim(0, 20)
```

6. Set the tick locations:

```
ax.set_xticks(range(9))
ax.set_yticks([2, 4, 6, 8, 10, 12, 14, 16, 18])
```

7. Set the axis labels:

```
ax.set_xticklabels(maturities)
ax.set_yticklabels([2, 4, 6, 8, 10, 12, 14, 16, 18])
```

8. Force the y-axis labels to the left:

```
ax.yaxis.set_label_position("left")
ax.yaxis.tick_left()
```

9. Add the axis labels:

```
plt.ylabel("Yield (%)")
plt.xlabel("Time to maturity")
plt.title("U.S. Treasury Bond Yield Curve")
```

10. Create the function that is run when the animation is initialized:

```
def init_func():
    line.set_data([], [])
    return line,
```

11. Create the function that runs at each iteration through the data:

```
def animate(i):
    x = range(0, len(maturities))
    y = data[maturities].iloc[i]
    dt_ = data.index[i].strftime("%Y-%m-%d")
    if data.inverted.iloc[i]:
        line.set_color("r")
    else:
        line.set_color("y")
    line.set_data(x, y)
    plt.title(f"U.S. Treasury Bond Yield Curve ({dt_})")
    return line,
```

12. Generate the animation that brings it all together:

```
ani = animation.FuncAnimation(
    fig,
    animate,
    init_func=init_func,
    frames=len(data.index),
    interval=250,
    blit=True
)
```

13. Persist the plot to give the script time to update and display the chart:

```
plt.show()
```

The result is an animated, interactive chart that displays the shape of the yield curve through time.

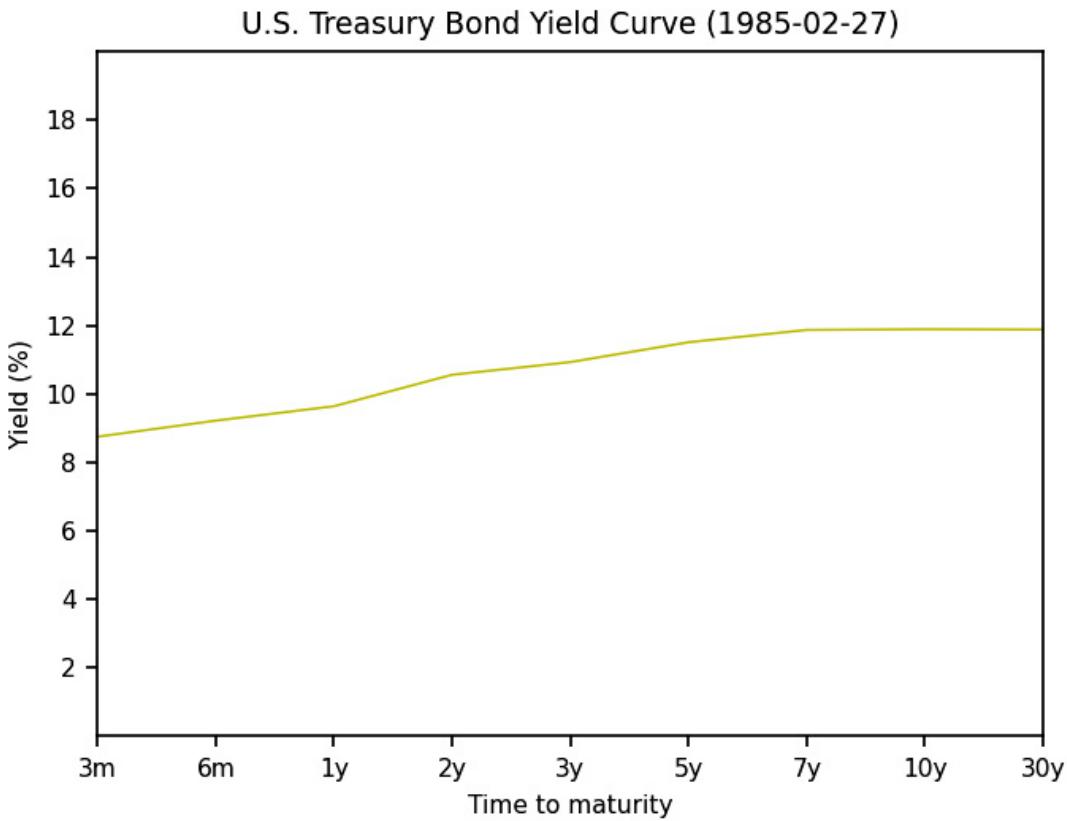


Figure 3.8: One frame from the animated plot of the US yield curve. This frame depicts the shape of the yield curve on 27 February 1985.

How it works...

A plot is prepared with maturity times on the horizontal axis and yield values on the vertical axis. A line plot is then set up to display the yield data.

The initialization function (`init_func`) establishes the starting frame of the animation. The main animation function (`animate`) is where the data gets updated. It redraws the line with the yields for each bond maturity at different time points. If the data indicates an inverted yield curve on a specific date, the line color changes to red, otherwise, it remains yellow.

The actual animation is put together using the `FuncAnimation` class. The class calls the `animate` function at regular intervals of 250 milliseconds. Using blitting helps make the animation more efficient by only redrawing the parts of the plot which change. The `show` command at the end displays the animation.

There's more...

`FuncAnimation` is a versatile class. Apart from the arguments used above, it also accepts the following:

1. `repeat`: A boolean value indicating if the animation should repeat once all frames have been displayed.
2. `repeat_delay`: The delay, in milliseconds, between consecutive repetitions of the animation.
3. `fargs`: Any additional arguments to pass to the `func` (i.e., the animation function).
4. `save_count`: The number of values from frames to cache, to improve performance when saving the animation. If `None`, then the entirety of `frames` is cached.
5. `cache_frame_data`: If `True`, the return values of the `animate` function are cached, which can be useful for speeding up the saving of long animations. If you set it to `False`, the frames will be recreated via the animation function during the save process.
6. `event_source`: An instance of `EventSource` or `None`. If `None`, a new instance of `TimerBase` will be created. This is useful when you want to synchronize multiple animations.

These are some of the main arguments you can use with `FuncAnimation`. There are also some internal and base class arguments available, but they're less commonly used in most applications.

See also

The animation API is very rich offering many animation options. You can read more here:

- https://matplotlib.org/stable/api/animation_api.html — Documentation for Matplotlib's animation API.

Plotting options implied volatility surfaces with Matplotlib

Traders use Matplotlib to visualize complex data, such as **options implied volatility surfaces**. These visuals help understand how implied volatility of options changes with different expiration dates and strike prices. Implied volatility surfaces are important for traders for information on the market's expectations of future volatility.

These surfaces show two main features: skew and term structure. Skew refers to how implied volatility varies at different strike prices for the same expiration date. It can indicate the market's expectation of significant price shifts. Term structure shows how implied volatility changes for options with the same strike price but different expiration dates. Term structure shows how volatility is expected to evolve over time.

Although a detailed explanation of skew and term structure is beyond the scope of this book, it's important to note these aspects of the volatility surface are important for making informed trading

decisions. They also play a significant role in selecting the right options for hedging and in fine-tuning the pricing and risk assessment models for more complex financial instruments.

Getting ready...

We'll start with a fresh Jupyter Notebook to avoid plotting the surface plot on the same axis as the animated plot.

How to do it...

As with the animated charts, to create three dimensional surfaces, we need to import the `Axes3D` class from the `mpl_toolkits` module.

1. Import the libraries:

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download options data using the OpenBB SDK:

```
chains = obb.derivatives.options.chains(
    "AAPL",
    provider=".cboe",
)
```

3. Filter the calls with days to expiration under three months and with a strike price greater than 100. Then drop any duplicates:

```
calls = calls[
    (calls.dte < 100)
    & (calls.strike >= 100)
]
calls.drop_duplicates(subset=["strike", "dte"], keep=False,
    inplace=True)
```

4. Pivot the DataFrame to put strikes in the index, days to expiration along the columns, and the implied volatility within the cells.

Then drop rows where all values in a column is nan:

```
vol_surface = (
    calls
    .pivot(
        index="strike",
        columns="dte",
        values="implied_volatility"
    )
    .dropna(how="all", axis=1)
)
```

5. Use NumPy's meshgrid method to create a two-dimensional grid using the strike price and days to expiration for use in the plot:

```
strike, dte = np.meshgrid(
```

```
    vol_surface.columns,  
    vol_surface.index  
)
```

6. Finally, plot the surface:

```
fig = plt.figure(figsize=(15, 15))  
ax = fig.add_subplot(111, projection='3d')  
ax.set_xlabel("Days to Expiration")  
ax.set_ylabel("Strike Price")  
ax.set_zlabel("Implied Volatility")  
ax.plot_surface(  
    strike,  
    dte,  
    vol_surface.values,  
    cmap="viridis"  
)
```

The result is a three-dimensional surface plot with the strike price on the x-axis, the days to expiration on the y-axis, and the implied volatility on the z-axis:

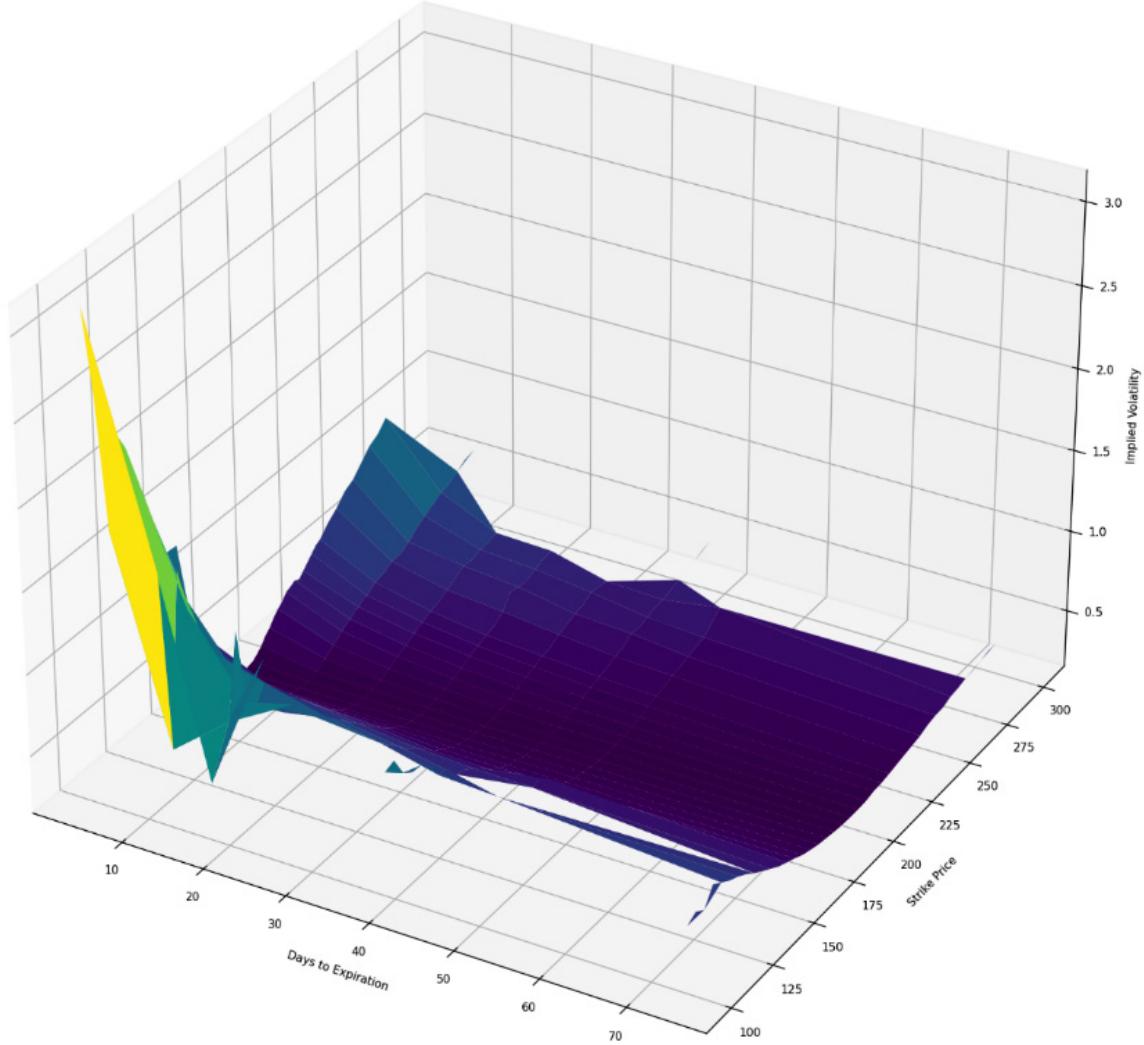


Figure 3.9: Options implied volatility surface showing for AAPL call options.

How it works...

We download options data and narrow down the dataset to options expiring in less than 90 days with a strike price of \$100 or more. It then cleans the data by removing any entries with identical strike prices and expiration periods to ensure uniqueness. The core of the process involves reorganizing the data into a DataFrame where each row and column represent different strike prices and expiration times. We then use the NumPy `meshgrid` method to build two-dimensional grids corresponding to the strike prices and expiration times. The grid is the same structure as the `vol_surface` DataFrame. The culmination of this process is the generation of a 3D plot, where the axes denote the strike price, days to expiration, and implied volatility, respectively.

There's more...

The `plot_surface` method of Matplotlib's 3D axes provides an easy way to create three-dimensional surface plots. In addition to the arguments we used in the preceding section, there are several other arguments commonly used in algorithmic trading:

1. **`rstride` and `cstride`**: These arguments control the stride (step size) used to create the surface plot. The `rstride` and `cstride` parameters set the stride size for row and column data, respectively.
2. **`color`**: A single color that can be used to color the entire surface.
3. **`facecolors`**: A matrix of the same size as Z that provides the colors for each face of the surface plot.
4. **`linewidth`**: The linewidth for the wireframe is drawn on the surface plot. The default is 0, meaning no wireframe.
5. **`antialiased`**: If set to `True`, the surface will be antialiased (smoothed).
6. **`shade`**: If `True`, the surface plot will be shaded, giving it a gradient effect based on light source and orientation.
7. **`vmin` and `vmax`**: The `colorbar` range. If either is `None`, it will be computed using `min(Z)` or `max(Z)` respectively.
8. **`facecolors`**: The face colors of the individual patches of the surface plot.

See also

To learn more about plotting , yield curves, and implied volatility surfaces, see the below resources:

- Documentation for Matplotlib's `plot_surface` method:
https://matplotlib.org/stable/api/_as_gen/mpl_toolkits.mplot3d.axes3d.Axes3D.plot_surface.html.
- More about the history of inverted yield curves: <https://www.investopedia.com/terms/i/invertedyieldcurve.asp>.
- More about how the implied volatility surface is used in practice: <https://www.investopedia.com/articles/stock-analysis/081916/volatility-surface-explained.asp>.

Visualizing statistical relationships with Seaborn

A major part of algorithmic trading is engineering factors. Factor engineering involves creating predictors for algorithmic trading models, often called **alpha factors**. These factors represent patterns or anomalies in the market. They aim to predict future price movements based on historical and real-time data. Factor engineering is increasingly reliant on machine learning and statistical methods. These tools help in automatically extracting patterns from vast datasets. Machine learning models, such as neural networks, can identify non-linear relationships missed by traditional methods. Feature selection techniques aid in determining the most relevant predictors. Regularization techniques prevent overfitting, ensuring model robustness. Clustering and dimensionality reduction help manage complex datasets.

The `seaborn` library is tailor-made for visualizing statistical relationships, making it an important tool for factor engineering. Seaborn is built on Matplotlib and integrates with pandas making it familiar to users of those tools (which we now are).

How to do it...

We'll import Seaborn for the statistical plots.

1. Import the libraries:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download stock price history using the OpenBB SDK:

```
data = obb.equity.price.historical(
    ["AAPL", "SPY"],
    start_date="2020-01-01",
    provider="yfinance"
).pivot(columns="symbol", values="close")
```

3. Compute the daily returns:

```
returns = (
    data
    .pct_change(fill_method=None)
    .dropna()
)
```

4. Reshape the data to a long format using the pandas melt method:

```
returns = returns.reset_index()
melted = pd.melt(
    returns,
```

```

        id_vars=["date"],
        value_vars=["AAPL"],
        var_name="stock",
        value_name="returns",
    )
)

```

5. Add a new column for the month:

```
melted["month"] = melted["date"].dt.to_period("M")
```

6. Generate the box plot:

```

g = sns.boxplot(
    x="month",
    y="returns",
    hue="stock",
    data=melted
)
g.set_xticklabels(
    melted["month"].unique(),
    rotation=45
)

```

The result is a box plot summarizing the monthly returns:

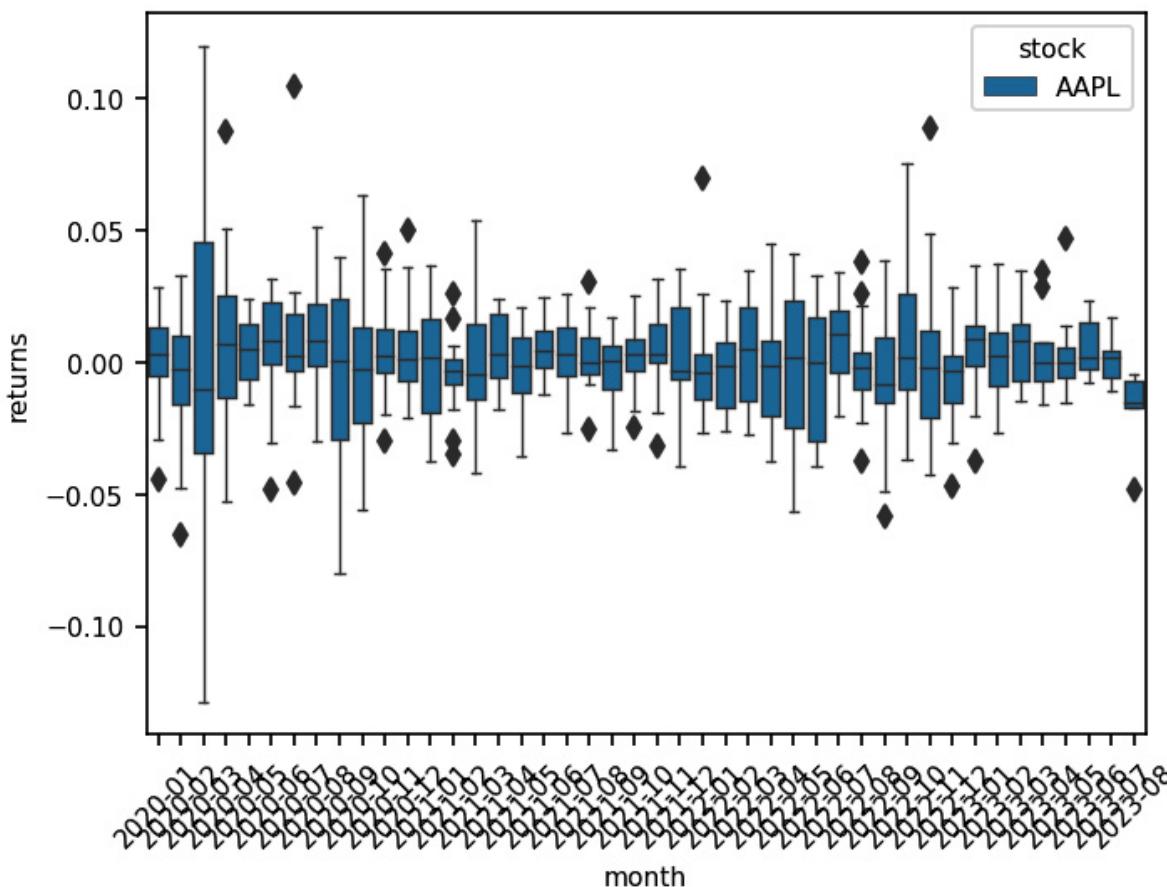


Figure 3.10: Box plot showing monthly summary statistics for AAPL returns.

How it works...

The code computes daily percentage changes in AAPL's closing prices and removes any `nan` values. This Series of returns is then transformed into a DataFrame with two columns: the date and the respective return for that date. Using `melt`, the DataFrame is reshaped to be compatible with Seaborn's boxplot function. It also extracts the month from each date and adds it as a new column. Lastly, the `boxplot` method creates a box plot with months on the x-axis and returns on the y-axis. The x-axis labels (months) are set with a 45-degree rotation to help with readability. The final visualization shows the distribution of AAPL's daily returns for each month in the data.

There's more...

Another popular chart type is the `jointplot` which is a combination of a scatter plot and histograms for each variable along the margins. Traders use joint plots to visually assess the correlation between assets, aiding in diversification, pairs trading, and risk management. The plots help in identifying linear relationships, distribution patterns, and potential outliers.

```
g = sns.jointplot(  
    x="SPY",  
    y="AAPL",  
    data=returns,  
    kind="reg",  
    truncate=False,  
)
```

The result is a joint plot.

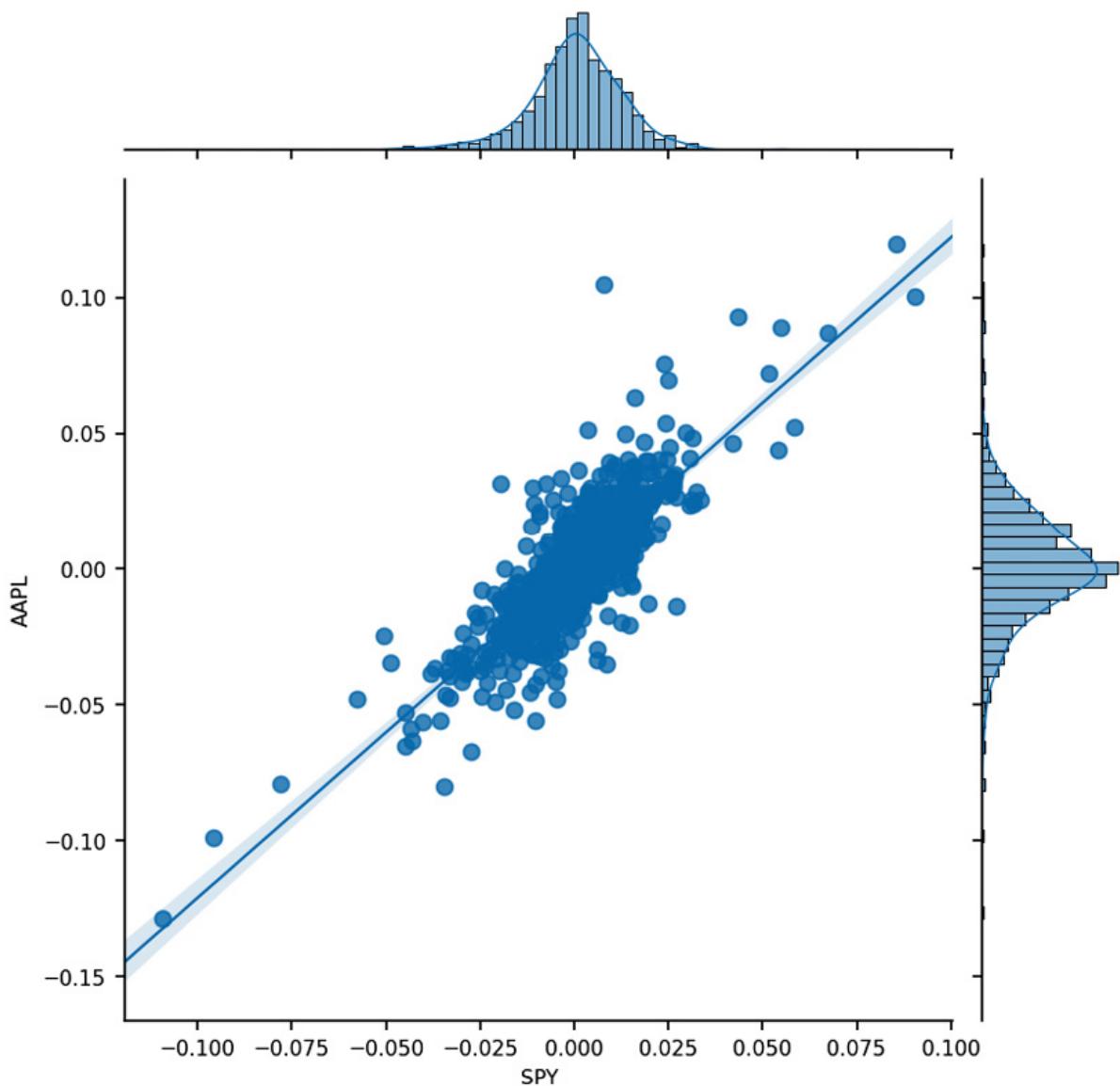


Figure 3.11: Joint plot demonstrating a positive linear relationship between AAPL and SPY returns.

TIP

Note the `reg` argument passed to the `kind` parameter. This indicates the type of plot to draw. The value “`reg`” stands for regression, meaning Seaborn will not only plot the scatter points of SPY vs. AAPL returns but also compute and display a linear regression fit to the data.

Traders use correlation matrices to understand the linear relationships between multiple assets simultaneously. Let’s build a correlation matrix for the stocks in the Dow Jones Industrial Average.

1. The first step is to grab the list of companies and their ticker symbols from the DJIA Wikipedia page.

```
dji = pd.read_html(
    "https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average"
) [1]
```

The `read_html` method is a convenient function in pandas that extracts tables from a given webpage. We're interested in the second table (indexed as `[1]`) which has details about the companies in the DJIA.

2. Now that we have the ticker symbols, let's use them to get the historical stock price data and calculate the daily returns.

```
dji_data = (
    obb.equity.price.historical(
        dji.Symbol, start_date="2020-01-01", provider="yfinance"
    )
).pivot(columns="symbol", values="close")
dji_returns = dji_data.pct_change(fill_method=None).dropna()
```

3. To understand the relationship between different stocks, we compute the pairwise correlation between all of them using the pandas `corr` method.

```
corr = dji_returns.corr()
```

4. Before visualizing the correlation, we'll make some tweaks to make our heatmap more intuitive. First, we'll create a mask to hide the upper triangle of the correlation matrix, as it mirrors the lower triangle.

```
mask = np.triu(
    np.ones_like(corr, dtype=bool)
)
```

5. Next, we'll generate a color palette that will help visually distinguish positive from negative correlations in our heatmap.

```
cmap = sns.diverging_palette(230, 20, as_cmap=True)
```

6. Set the font size to 4 points to include more of the labels and plot the heatmap.

```
plt.rcParams["font.size"] = 4
sns.heatmap(
    corr,
    mask=mask,
    cmap=cmap,
    vmin=-1.0,
    vmax=1.0,
    center=0,
    square=True,
    linewidths=0.5,
)
```

The result is a correlation heatmap.

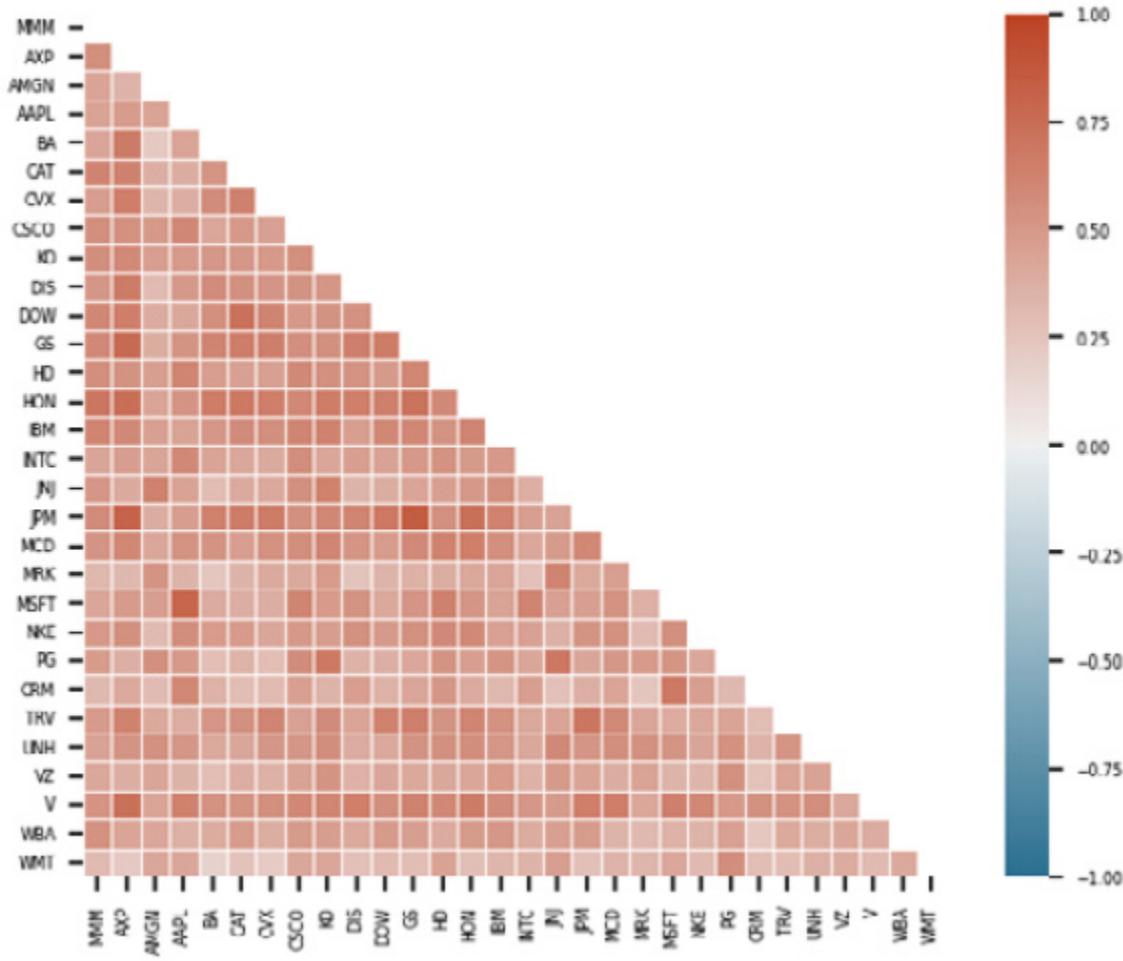


Figure 3.12: Heatmap shaded with the correlations between the constituents of the Dow Jones Industrial Average.

See also

Seaborn is a popular charting library with many unique visualization types. Read more here:

- <https://seaborn.pydata.org/generated/seaborn.boxplot.html> — Documentation for the Seaborn boxplot.
- <https://seaborn.pydata.org/generated/seaborn.jointplot.html> — documentation for the Seaborn joint plot.
- <https://seaborn.pydata.org/generated/seaborn.heatmap.html> — documentation for the Seaborn heatmap.

Creating an interactive PCA analytics dashboard with Plotly Dash

Principal component analysis is used widely in data science. It's a way to reduce the number of dimensions in a data set. In a stock portfolio, a dimension might be a column of returns for one of the

stocks. In a portfolio of 100 stocks, there are 100 dimensions. PCA converts those 100 dimensions into the few that explain the most variance in the data.

PCA isolates the statistical return drivers of a portfolio. These drivers are called “alpha factors” (or just factors) because they create returns that are not explained by a benchmark. Quants use factors in trading strategies. First, they isolate the components. Then they buy the stocks with the largest exposure to a factor and sell the stocks with the smallest exposure to a factor. We’ll look at PCA in a later recipe. For now, we’ll use it to create a Plotly Dash app.

Instead of changing the dates, number of components, and ticker symbols in code, we can do it in an interactive web app. In this recipe, we’ll create a Plotly Dash app that accepts a list of ticker symbols, identifies the principal components of their returns, and generates plots to visualize the top factors.

Getting ready...

So far, we’ve been writing code in Jupyter Notebook. For this recipe, you’ll create a Python script called `app.py` and run it from the command line. Our aim is to highlight some of the intriguing features of Plotly Dash, even if they aren’t essential for a basic app for conducting principal component analysis on stock returns.

You’ll need to install a few new libraries. Make sure your virtual environment is active and run the following command:

```
pip install dash plotly dash-bootstrap-components scikit-learn
```

Plotly and `scikit-learn` may already be installed through the OpenBB Platform installation.

How to do it...

All the code below should be written in the `app.py` script file.

1. Begin by importing all necessary libraries to create the web application:

```
import datetime
import numpy as np
import pandas as pd
import dash
from dash import dcc, html
import dash_bootstrap_components as dbc
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import plotly.io as pio
from sklearn.decomposition import PCA
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Set the default styling, chart templates, and initialize the app:

```
pio.templates.default = "plotly"
app = dash.Dash(__name__, external_stylesheets=[
    dbc.themes.BOOTSTRAP])
```

3. Construct the components of the user interface starting with the text field to enter the list of ticker symbols, the dropdown to select the number of components, the date picker to select the range of data, and the submit button to run the app:

```
ticker_field = [
    html.Label("Enter Ticker Symbols:"),
    dcc.Input(
        id="ticker-input",
        type="text",
        placeholder="Enter Tickers separated by commas (
            e.g. AAPL,MSFT)",
        style={"width": "50%"}
    ),
]
components_field = [
    html.Label("Select Number of Components:"),
    dcc.Dropdown(
        id="component-dropdown",
        options=[{"label": i, "value": i} for i in range(1,6)],
        value=3,
        style={"width": "50%"}
    ),
]
date_picker_field = [
    html.Label("Select Date Range:"), # Label for date picker
    dcc.DatePickerRange(
        id="date-picker",
        start_date=datetime.datetime.now() - datetime.timedelta(
            365 * 3),
        end_date=datetime.datetime.now(),
        # Default to today's date
        display_format="YYYY-MM-DD",
    ),
]
submit = [
    html.Button("Submit", id="submit-button"),
]
```

4. Combine the form elements and placeholders for visualizations to form the app layout:

```
app.layout = dbc.Container(
    [
        html.H1("PCA on Stock Returns"),
        dbc.Row([dbc.Col(ticker_field)]),
        dbc.Row([dbc.Col(components_field)]),
        dbc.Row([dbc.Col(date_picker_field)]),
        dbc.Row([dbc.Col(submit)]),
        dbc.Row(
            [
                dbc.Col([dcc.Graph(id="bar-chart")], width=4),
                dbc.Col([dcc.Graph(id="line-chart")],
                    width=4),
                dbc.Col([dcc.Graph(id="scatter-plot")],
                    width=4),
            ]
        ),
    ],
)
```

```
    ]  
)
```

5. Implement the function that updates the charts upon user input:

IMPORTANT

All the code between steps 5 and 13 belongs to the same function. Make sure you properly indent the code after the definition.

```
@app.callback(  
    [  
        Output("bar-chart", "figure"),  
        Output("line-chart", "figure"),  
        Output("scatter-plot", "figure"),  
    ],  
    [Input("submit-button", "n_clicks")],  
    [  
        dash.dependencies.State("ticker-input", "value"),  
        dash.dependencies.State("component-dropdown", "value"),  
        dash.dependencies.State("date-picker", "start_date"),  
        dash.dependencies.State("date-picker", "end_date"),  
    ],  
)  
def update_graphs(n_clicks, tickers, n_components, start_date, end_date):  
    if not tickers:  
        return {}, {}, {}
```

6. Parse inputs from the user:

```
tickers = tickers.split(",")  
start_date = datetime.datetime.strptime(start_date,  
    "%Y-%m-%dT%H:%M:%S.%f").date()  
end_date = datetime.datetime.strptime(end_date,  
    "%Y-%m-%dT%H:%M:%S.%f").date()
```

7. Download stock data:

```
data = obb.equity.price.historical(  
    tickers,  
    start_date=start_date,  
    end_date=end_date,  
    provider="yfinance"  
).pivot(columns="symbol", values="close")  
daily_returns = data.pct_change().dropna()
```

8. Fit the principal component model:

```
pca = PCA(n_components=n_components)  
pca.fit(daily_returns)  
explained_var_ratio = pca.explained_variance_
```

9. Generate the bar chart for individual explained variance:

```
bar_chart = go.Figure(  
    data=[  
        go.Bar(  
            x=["PC" + str(i + 1) for i in range(  
                n_components)],  
            y=explained_var_ratio,  
        )
```

```

        ],
        layout=go.Layout(
            title="Explained Variance by Component",
            xaxis=dict(title="Principal Component"),
            yaxis=dict(title="Explained Variance"),
        ),
    )
)

```

10. Generate the line chart for cumulative explained variance:

```

cumulative_var_ratio = np.cumsum(explained_var_ratio)
line_chart = go.Figure(
    data=[
        go.Scatter(
            x=["PC" + str(i + 1) for i in range(
                n_components)],
            y=cumulative_var_ratio,
            mode="lines+markers",
        )
    ],
    layout=go.Layout(
        title="Cumulative Explained Variance",
        xaxis=dict(title="Principal Component"),
        yaxis=dict(title="Cumulative Explained Variance"),
    ),
)

```

11. Compute factor exposures:

```

X = np.asarray(daily_returns)
factor_returns = pd.DataFrame(
    columns=["f" + str(i + 1) for i in range(
        n_components)],
    index=daily_returns.index,
    data=X.dot(pca.components_.T),
)
factor_exposures = pd.DataFrame(
    index=[("f" + str(i + 1)) for i in range(n_components)],
    columns=daily_returns.columns,
    data=pca.components_,
).T
labels = factor_exposures.index
data = factor_exposures.values

```

12. Generate the chart for factor exposures:

```

scatter_plot = go.Figure(
    data=[
        go.Scatter(
            x=factor_exposures["f1"],
            y=factor_exposures["f2"],
            mode="markers+text",
            text=labels,
            textposition="top center",
        )
    ],
    layout=go.Layout(
        title="Scatter Plot of First Two Factors",
        xaxis=dict(title="Factor 1"),
        yaxis=dict(title="Factor 2"),
    ),
)

```

13. Return the charts to the app:

```
return bar_chart, line_chart, scatter_plot
```

To run the app, open the terminal, navigate to the directory where the `app.py` script is located, and run the following command:

```
python app.py
```

Running the code starts the app and prints the URL where it is running. Navigate to the URL, enter a list of tickers and press submit. The result will look something like this:

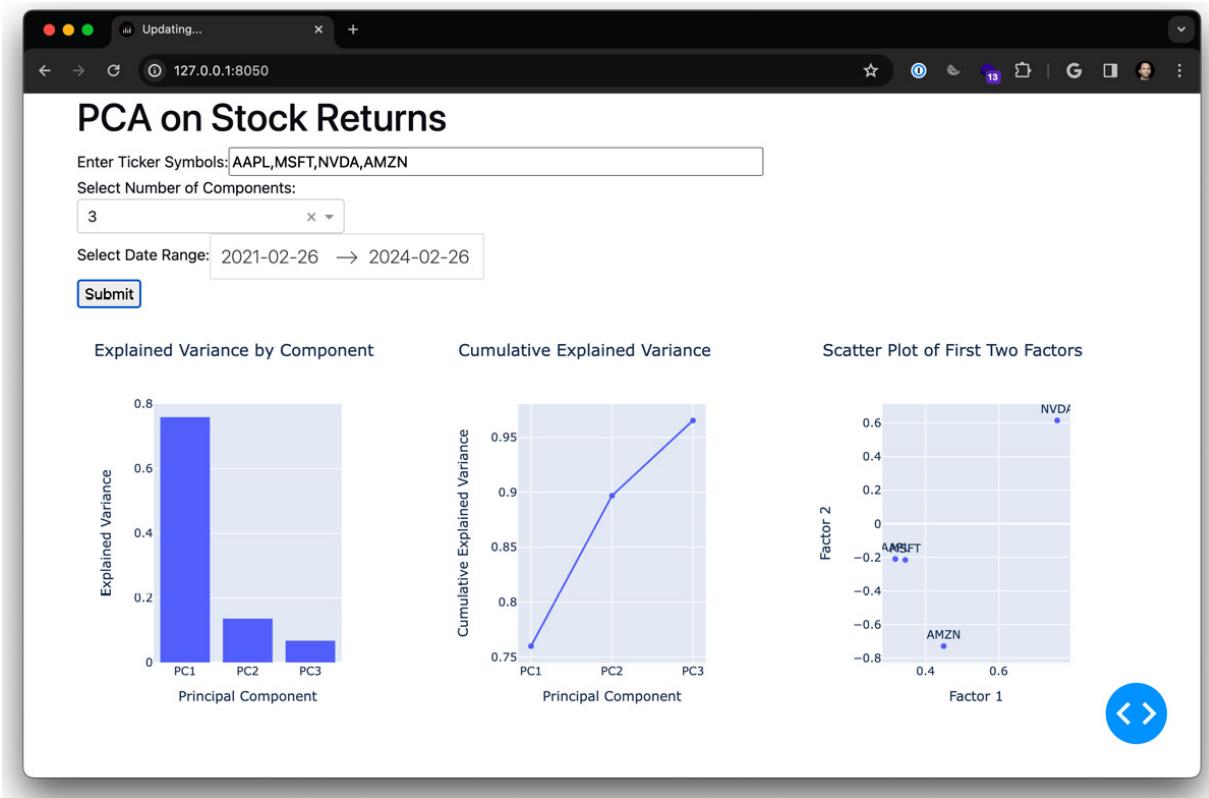


Figure 3.13: Screenshot of our new PCA analytics app

How it works...

Callbacks in Plotly Dash define the logic that connects the components in the application. These callbacks are what enable user interactions. A callback refers to a Python function that gets automatically executed by Dash whenever an input component's property changes. In our app, the callback is defined as `update_graphs`.

First we import the necessary libraries we need for the analysis and web app. We import `datetime` for handling dates, NumPy for numerical operations, and pandas for data manipulation. Dash, its

extensions, and other related packages provide the backbone for our web application. Specifically, `dash_bootstrap_components` assists in styling, and OpenBB fetches the stock data. Lastly, PCA from `sklearn` will help us perform the principal component analysis.

We then choose a default template for styling the charts and initialize our Dash app with a Bootstrap theme. Next we construct the components the user will interact with like a text field for entering ticker symbols, a dropdown to choose the number of principal components, a date picker for specifying the data range, and a submit button to start the analysis. Next we organize our UI components into a layout.

Next, we implement the callback function, `update_graphs`, which ties our user inputs to the visualization outputs. Every time the user presses the submit button, this function fetches the data, performs the PCA, and updates the visualizations. We define a callback by using the `callback` decorator provided by Dash. This decorator specifies which component properties to watch (`input`) and which to update (`output`). In our app, clicking the submit button triggers the callback function. The decorator takes three inputs:

- `Output`: Specifies which components will be updated once the callback function is executed.
- `Input`: Specifies which components the callback should listen to.
- `State`: Specifies which components the callback should read the current value from without triggering the function.

When the submit button is clicked, `update_graphs` gets called, the current values of the specified components are passed as arguments to the function, and the logic inside `update_graphs` executes. The function returns the visualizations, and these are rendered in the specified `output` components in the app.

There's more...

Plotting with different Python libraries has distinct advantages depending on the context. `pandas`, while inherently great for data manipulation, offer basic plotting capabilities that are mainly suited for quick and simple visualizations directly from `DataFrames`. `Matplotlib`, one of the most widely used plotting libraries for Python, provides a greater degree of flexibility and customization but generally lacks interactive features. `Seaborn`, which is built on top of `Matplotlib`, enhances visual aesthetics and has functions tailored for statistical visualizations, making it more intuitive for certain analyses. `Plotly Express`, on the other hand, is a more modern library designed for interactivity from the ground up, making it well-suited for dynamic environments like Plotly Dash apps.

See also

PCA is common in all data sciences and especially useful in algorithmic trading. Read more about the technical details of PCA at Wikipedia.

- Description of PCA: https://en.wikipedia.org/wiki/Principal_component_analysis .
- Plotly Dash user guide: <https://dash.plotly.com>.

4

Store Financial Market Data on Your Computer

If there's one thing algorithmic traders cannot get enough of, it's data. The data that fuels our strategies is more than just numbers—it's the lifeblood of our decision-making processes. And having data available locally—or at least within your control—is a big part of that. Speed of access and reliability are important reasons why you might want to store data locally. Local data is insulated from internet outages, ensuring that data-driven processes remain uninterrupted. Further, if you need to update a bad price, you can persist the update through time.

In terms of price considerations, local storage offers cost-efficiency benefits over recurring cloud expenses. Storing a few terabytes of data in a cloud-based database can cost several hundred dollars per month. The flexibility of data manipulation, ease of integration with research workflows, and speeding up of backtests are other advantages.

In this chapter, we'll explore several ways to store financial market data. We'll start with storing in a CSV file, which can easily be written and read by pandas. Then we'll explore ways to store data in a simple, on-disk SQL database format called **SQLite**. We'll increase the complexity and install a **PostgreSQL** database server on your computer to store data. Finally, we'll use the highly efficient, ultra-fast **HDF5** format to store data.

For recipes using **SQLite** and **PostgreSQL**, we'll develop a script that can be run automatically using a task manager to acquire market data after the market closes.

In this chapter, we present the following recipes:

- Storing data on disk in CSV format
- Storing data on disk with SQLite
- Storing data in a networked Postgres database
- Storing data in ultra-fast HDF5 format

Storing data on disk in CSV format

The **Comma-Separated Values (CSV)** format is one of the most universally recognized and utilized methods for storing data. Its simplicity makes it a favored choice for traders and analysts looking to store tabular data without the overhead of more complex systems. Algorithmic traders often gravitate toward CSV when dealing with data that requires straightforward import and export operations,

especially given the ease with which Python and its libraries, such as pandas, handle CSV files. Further, data in CSV format can be used with other analytics tools such as Tableau, PowerBI, or proprietary systems. Manually inspecting CSV files is also possible using a text editor or Excel. CSV does not have the same speed or sophistication as other storage methods, but its ease of use makes it important in all trading environments.

How to do it...

Since pandas supports writing data to CSV, there are no special libraries required:

1. Import the libraries:

```
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Implement a function to download data, manipulate the results, and return a pandas DataFrame:

```
def get_stock_data(symbol, start_date=None, end_date=None):
    data = obb.equity.price.historical(
        symbol,
        start_date=start_date,
        end_date=end_date,
        provider="yfinance",
    )
    data.reset_index(inplace=True)
    data['symbol'] = symbol
    return data
```

3. Implement a function to save a range of data as a CSV file:

```
def save_data_range(symbol, start_date=None, end_date=None):
    data = get_stock_data(symbol, start_date, end_date)
    data.to_csv(
        f"{symbol}.gz",
        compression="gzip",
        index=False
    )
```

4. Implement a function that reads a CSV file and returns a DataFrame:

```
def get_data(symbol):
    return pd.read_csv(
        f'{symbol}.gz',
        compression='gzip',
        index_col='date',
        usecols=[
            'date',
            'open',
            'high',
            'low',
            'close',
            'volume',
            'symbol'
        ]
    )
```

5. Save the data as a CSV file:

```
save_data_range("PLTR")
```

How it works...

The script uses the `pandas` library and the OpenBB Platform to download and manipulate stock market data. The `get_stock_data` function fetches stock data using a given symbol and date range, then preprocesses this data by resetting its index and standardizing column names. It also appends a stock symbol column for reference.

`save_data_range` uses the `pandas to_csv` method to store the data to disk using GZIP compression in the CSV format. The naming convention for saved files is the stock symbol followed by the `.gz` file extension.

The `get_data` function retrieves and decompresses the stored CSV files, reconstructing them into a DataFrame. During this process, only selected columns such as `date`, `opening price`, `high`, `low`, `closing price`, `volume`, and `stock symbol` are loaded.

The end of the script uses these functions to save stock data for the symbol PLTR. After running this code, you'll have a file on your computer called `PLTR.gz`, which is a compressed CSV file.

There's more...

The `pandas to_csv` method has many options for efficiently saving data to disk in CSV format. Here are some of the most useful:

- **Sep**: Specifies the delimiter to use between fields, defaulting to a comma (,). For a tab-separated file, use `sep='\t'`.
- **header**: A Boolean value determining whether to write out column names. Set to `False` to exclude column headers from the CSV.
- **na_rep**: Sets the string representation for missing (`nan`) values. The default is empty strings, but can be changed to placeholders such as `NULL`.
- **date_format**: Dictates the format for datetime objects. For example, `date_format='%Y-%m-%d %H:%M:%S'` formats datetime objects as `2023-08-10 15:20:30`.
- **float_format**: Controls the format for floating point numbers. For instance, `float_format='%.2f'` rounds all float columns to two decimal places.

Similarly, the `read_csv` method used in the `get_data` method has additional arguments for added flexibility in fetching data from disk:

- **delimiter** or **sep**: Specifies the character that separates fields, defaulting to a comma (,). For tab-separated files, use `delimiter='\t'`.

- **nrows**: Determines the number of rows of the file to read, which can be useful for reading in just a subset of a large file.
- **parse_dates**: A list of column names to parse as dates. If `['date']` is passed, the `date` column will be parsed into a `datetime64` type.
- **dtype**: Provides a dictionary of column names and data types to use for each column. For example, `dtype={'volume': 'int32'}` would ensure the `volume` column is read as a 32-bit integer.
- **skiprows**: Specifies a number or a list of row numbers to skip while reading the file, useful for omitting specific rows.

See also...

Writing data to CSV is a very common operation when dealing with market data. It's important to become comfortable with the different ways you can read and write data from and to CSVs:

- Documentation for the pandas `to_csv` method: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html
- Documentation for the pandas `from_csv` method: https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

Storing data on disk with SQLite

SQLite offers a bridge between the simplicity of flat files and the robustness of relational databases. As a serverless, self-contained database, SQLite provides algorithmic traders with a lightweight yet powerful tool to store and query data with SQL but without the complexity of setting up a full-scale database system. Its integration with Python is seamless, and its compact nature makes it an excellent choice for applications where portability and minimal configuration are priorities. For traders who require more structure than CSVs, or prefer to use SQL, but without the overhead of larger database systems, SQLite is the optimal choice.

Getting ready...

We'll build a script that can be set to run automatically using a CRON job (Mac, Linux, Unix) or Task Scheduler (Windows). For this recipe, we will create a Python script called `market_data_sqlite.py` and run it from the command line. We'll also introduce the `exchange_calendars` Python package, which is a library for defining and querying calendars for trading days and times for over 50 exchanges. You can install it with `pip`:

```
pip install exchange_calendars
```

How to do it...

All the following code should be written in the `market_data_sqlite.py` script file:

1. Begin by importing all necessary libraries to fetch and save data:

```
from sys import argv
import sqlite3
import pandas as pd
import exchange_calendars as xcals
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Reuse the `get_stock_data` function from the previous recipe:

```
def get_stock_data(symbol, start_date=None, end_date=None):
    data = obb.equity.price.historical(
        symbol,
        start_date=start_date,
        end_date=end_date,
        provider="yfinance",
    )
    data.reset_index(inplace=True)
    data['symbol'] = symbol
    return data
```

3. Modify the `save_data_range` function to use the pandas `to_sql` method:

```
def save_data_range(symbol, conn, start_date,
                    end_date):
    data = get_stock_data(symbol, start_date,
                          end_date)
    data.to_sql(
        "stock_data",
        conn,
        if_exists="replace",
        index=False
    )
```

4. Create a function that grabs data from the last trading day based on the exchange's calendar:

```
def save_last_trading_session(symbol, conn, today):
    data = get_stock_data(symbol, today, today)
    data.to_sql(
        "stock_data",
        conn,
        if_exists="append",
        index=False
    )
```

5. Create the script's main execution code, which allows the user to pass in a stock symbol, start, and end date to kick off the data acquisition and storage process:

```
if __name__ == "__main__":
    conn = sqlite3.connect("market_data.sqlite")
    if argv[1] == "bulk":
        symbol = argv[2]
        start_date = argv[3]
        start_date = argv[4]
        save_data_range(symbol, conn, start_date=None,
                        end_date=None)
        print(f"{symbol} saved between {start_date} and {end_date}")
    elif argv[1] == "last":
        symbol = argv[2]
```

```

calendar = argv[3]
cal = xcals.get_calendar(calendar)
today = pd.Timestamp.today().date()
if cal.is_session(today):
    save_last_trading_session(symbol, conn, today)
    print(f"{symbol} saved")
else:
    print(f"{today} is not a trading day. Doing nothing.")
else:
    print("Enter bulk or last")

```

6. To save a range of data, run the following command from your terminal:

```
python market_data_sqlite.py bulk SYMBOL START_DATE END_DATE
```

Where `SYMBOL` is the ticker symbol, `START_DATE` is the first date you want to download data for, and `END_DATE` is the last date you want to download data for. Here's an example:

```
python market_data_sqlite.py bulk SPY 2022-01-01 2022-10-20
```

This downloads and saves data for the SPY symbol between 2022-01-01 and 2022-10-20.

IMPORTANT

The `if_exists` argument is set to `replace` in the `to_sql` method in the `save_data_range` function. That means every time you call the function, the table will be dropped and replaced if it exists. Depending on your use case, you may want to set the value to `append` if you are adding new data for different stocks at different times.

Here's how you'd download data for the last trading day:

```
python market_data_sqlite.py last SYMBOL XNYS
```

Here, `SYMBOL` is the ticker symbol.

TIP

To get a list of supported calendars from the `exchange_calendars` package, you can run `xcals.get_calendar_names(include_aliases=False)`.

How it works...

We use the same function for fetching a range of stock price data as the last recipe. In this recipe, we modify the `save_data_range` function by including an argument to accept a connection, and instead of saving to CSV, we use the pandas `to_sql` method to save the data in the DataFrame to an SQLite table. The table exists inside a file called `market_data.sqlite`, which we defined in the `connect` method. By calling this method, the Python `sqlite3` package will create the file if it does not exist, or connect to it if it does.

The `save_last_trading_session` method takes a connection and the date for which data is downloaded. We call the `get_stock_data` function we created with the current date as the start and

end date. This returns one row of data for the current date. After the data is downloaded, it is appended to the SQLite table.

The code under the `if` statement runs when called from the command line. `argv` is a list provided by the `sys` module that captures command-line arguments passed to a script. The first item in the list (`argv[0]`) represents the script's name itself, and subsequent items contain the arguments in the order they were provided. We use `argv` to determine whether we should download a range of data or data for the last trading session. Depending on whether the user enters `bulk` or `last`, we capture the symbol, start, and end date and call the appropriate function. If the user enters `last`, we use the pandas `Timestamp` class to determine the current date, then use `exchange_calendars` to test whether the current date is a trading day for the given exchange. If it is, we append the last day's data. If it's not, we print a message and do nothing.

There's more...

Automating data retrieval ensures consistent and timely inputs for your trading workflows. Here's how to automate the script you just created to run at 1:00 p.m. EST daily.

Windows

Windows users can create a batch file to run the Python script:

1. Create a batch file:

- Create a new `.bat` file (for example, `run_script.bat`).
- Inside, add the following:

```
@echo off
CALL conda activate quant-stack
python path_to_your_script\market_data_sqlite.py %1 %2
```

2. Open Windows Task Scheduler:

- Press `Windows + R`, type `taskschd.msc`, and hit `Enter`.

3. Create a new task:

- In the **Actions** pane, click on **Create Basic Task**.

4. Provide the name and description of the task:

- **Name:** `Run Market Data Script`
- **Description:** `Runs the Python script every weekday at 11:00 pm.`

5. Set the trigger:

- Choose **Daily**.
- **Start:** Set today's date and **11:00 pm**.
- **Recur every:** **1 day**.
- Check **Weekdays** in the advanced settings.

6. Set the action:

- Choose **Start a program**.
- **Program/script:** Browse and select the **.bat** file you created.
- **Add arguments:** **last SPY XNYS**

7. Finish the setup:

- Click on **Finish**.

Mac/Unix/Linux

Mac and Unix users can create an executable shell file to run the Python script:

1. Create a shell script:

- Create a new file called **run_script.sh**.
- Inside, add the following:

```
#!/bin/bash
source /path_to_anaconda/anaconda3/bin/activate quant-stack
python /path_to_your_script/market_data_sqlite.py $1 $2
```

- Give execute permissions:

```
chmod +x run_script.sh
```

2. Open the cron table:

- Open terminal.
- Enter **crontab -e**.

3. Add a cron job:

- To run the script at 11:00 p.m. EST on weekdays, append the following line:

```
0 23 * * 1-5 /path_to_shell_script/run_script.sh last SPY XNYS
```

NOTE

The time may need adjustment for Daylight Saving or based on your server's time zone.

4. Save and exit:

- Press *Ctrl + O* to save (if you're using nano).
- Press *Ctrl + X* to exit.

5. Verify the cron job:

- Enter `crontab -l` in the terminal to ensure your job is listed.

In both cases, we assume your virtual environment is named `quant-stack`.

See also...

SQLite is an extremely fast SQL-compatible file format. You can use all the SQL you already know with SQLite.

- SQLite home page: <https://www.sqlite.org/index.html>
- Documentation for the pandas `to_sql` method: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_sql.html
- Documentation for the `exchange_calendars` package: https://github.com/gerrymanoim/exchange_calendars

Storing data in a PostgreSQL database server

PostgreSQL, commonly known as **Postgres**, is an advanced open source relational database system. Its ability to handle vast datasets, coupled with intricate querying capabilities, makes it a good option for algorithmic traders who need improved performance over on-disk options. Postgres is also a popular database choice for cloud providers such as AWS, in case you need cloud storage for your data.

The scalability and robustness of Postgres are especially relevant when dealing with higher-frequency trading data or when multiple systems and strategies require concurrent access to shared data resources. While the setup may be more involved compared to other storage solutions, the benefits of centralized, networked access with stringent data integrity checks are desirable for sophisticated trading operations.

Getting ready...

To follow this recipe, you'll either need access to an existing remote Postgres database server or one installed on your computer. Follow these steps to get Postgres running on your local computer. Depending on your operating system, you can install Postgres and its dependencies at the command line using one of the following options.

For Windows:

1. Download the Postgres installation file from the Postgres downloads page.

2. Double-click the installation file and follow the instructions.
3. Open pgAdmin from the **Start** menu under the PostgreSQL folder.

For Debian/Ubuntu:

Run the following command from your command line:

```
sudo apt-get install libpq-dev python3-dev
```

For Red Hat/CentOS/Fedora

Run the following command from your command line:

```
sudo yum install postgresql-devel python3-devel
```

For macOS (using Homebrew)

Run the following command from your terminal window:

```
brew install postgresql
```

Once Postgres is installed, follow the instructions for your operating system (usually printed on the screen) to start the Postgres database server. After the Postgres database server is installed, use `pip` to install SQLAlchemy and the `psycopg2` driver:

```
pip install sqlalchemy psycopg2
```

We'll build a script that can be set to run automatically using a CRON job or Task Scheduler. For this recipe, you'll create a Python script called `market_data_postgres.py` and run it from the command line.

How to do it...

All the following code should be written in the `market_data_postgres.py` script file:

1. Import the required libraries:

```
import pandas as pd
from sqlalchemy import create_engine, text
from sqlalchemy.exc import ProgrammingError
import exchange_calendars as xcals
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Implement a function that creates a database to store market data if one does not exist:

```
def create_database_and_get_engine(db_name, base_engine):
    conn = base_engine.connect()
    conn = conn.execution_options(
        isolation_level="AUTOCOMMIT")
    try:
        conn.execute(text(f"CREATE DATABASE {db_name};"))
```

```

        except ProgrammingError:
            pass
    finally:
        conn.close()
    conn_str = base_engine.url.set(database=db_name)
    return create_engine(conn_str)

```

3. Reuse the same `get_stock_data()` function as in the previous two recipes.

4. Slightly modify the `save_data_range` function to change the variable name `conn` to `engine` to match what is being passed:

```

def save_data_range(symbol, engine, start_date=None, end_date=None):
    data = get_stock_data(symbol, start_date, end_date)
    data.to_sql(
        "stock_data",
        engine,
        if_exists="append",
        index=False
    )

```

5. Change the function so it only saves the last trading session's data:

```

def save_last_trading_session(symbol, engine, today):
    data = get_stock_data(symbol, today, today)
    data.to_sql(
        "stock_data",
        engine,
        if_exists="append",
        index=False
    )

```

6. Create the script's main execution code, which creates the database connection and calls our Python code to download and save the data:

```

if __name__ == "__main__":
    username = ""
    password = ""
    host = "127.0.0.1"
    port = "5432"
    database = "market_data"
    DATABASE_URL = f"postgresql://{{username}}:{{password}}@{{host}}:{{port}}/postgres"
    base_engine = create_engine(DATABASE_URL)
    engine = create_database_and_get_engine(
        "stock_data", base_engine)
    if argv[1] == "bulk":
        symbol = argv[2]
        start_date = argv[3]
        start_date = argv[4]
        save_data_range(symbol, engine,
                        start_date=None, end_date=None)
        print(f"{symbol} saved between {start_date} and {end_date}")
    elif argv[1] == "last":
        symbol = argv[2]
        calendar = argv[3]
        cal = xcals.get_calendar(calendar)
        today = pd.Timestamp.today().date()
        if cal.is_session(today):
            save_last_trading_session(symbol, engine, today)
            print(f"{symbol} saved")
    else:

```

```
print(f"{today} is not a trading day. Doing  
nothing.")
```

7. To save a range of data, run the following command from your terminal:

```
python market_data_sqlite.py bulk SYMBOL START_DATE END_DATE
```

Here, **SYMBOL** is the ticker symbol, **START_DATE** is the first date you want to download data for and **END_DATE** is the last date you want to download data for. Here's an example:

```
python market_data_sqlite.py bulk SPY 2022-01-01 2022-10-20
```

This command downloads and saves data for ticker symbol SPY between **2022-01-01** and **2022-10-20**.

IMPORTANT

By default, Postgres does not set a username and password. Depending on your operating system and installed tools (for example, pgAdmin), the process for creating a username and password differs. It is critical that you set both of these to ensure the integrity of the data. It's also best practice to create an `.env` file that stores your credentials used within Python code, read them using the `dotenv` package, and set them from environment variables.

How it works...

We started by importing the necessary Python libraries. In this recipe, we introduced SQLAlchemy, which provides tools to connect to and interact with databases (more on this follows).

Next, we implement a function to create a new database. If the database already exists, it simply connects to it. The **AUTOCOMMIT** isolation level is set to bypass PostgreSQL's restriction against creating databases within transaction blocks. After the database is created, the function returns the engine.

Next is a series of functions to fetch financial market data and use the pandas **to_sql** method to store the data in the Postgres database. We create two functions so we can both bulk save data for new tickers and save the last trading day's data. These functions are similar to what we built in the previous recipes except for a change in variable names.

In the main code execution block, we set up the connection parameters and create an "engine" which is the way we will connect to the Postgres database through pandas. The rest of the code operates the same as the previous recipe.

There's more...

SQLAlchemy is a powerful toolkit for interacting with databases in Python, enabling seamless communication between Python applications and relational databases. Its **Object Relational Mapping (ORM)** layer lets developers interact with databases using native Python classes, abstracting away the intricacies of raw SQL. Additionally, it is database-agnostic which means applications can be built once and then deployed across various database backends with minimal changes, ensuring flexibility and scalability. This is very useful when building a development database on your local computer and transitioning to a database on a remote server later.

See also...

For those that wish to further explore the advanced features of SQLAlchemy, there are a series of quick start guides describing how to model databases using Python classes on the SQLAlchemy website: <https://docs.sqlalchemy.org/en/20/orm/quickstart.html>

- SQLAlchemy documentation page: <https://docs.sqlalchemy.org/en/20/index.html>
- PostgreSQL home page: <https://www.postgresql.org>

A great way to manage a Postgres database server is the free pgAdmin software.

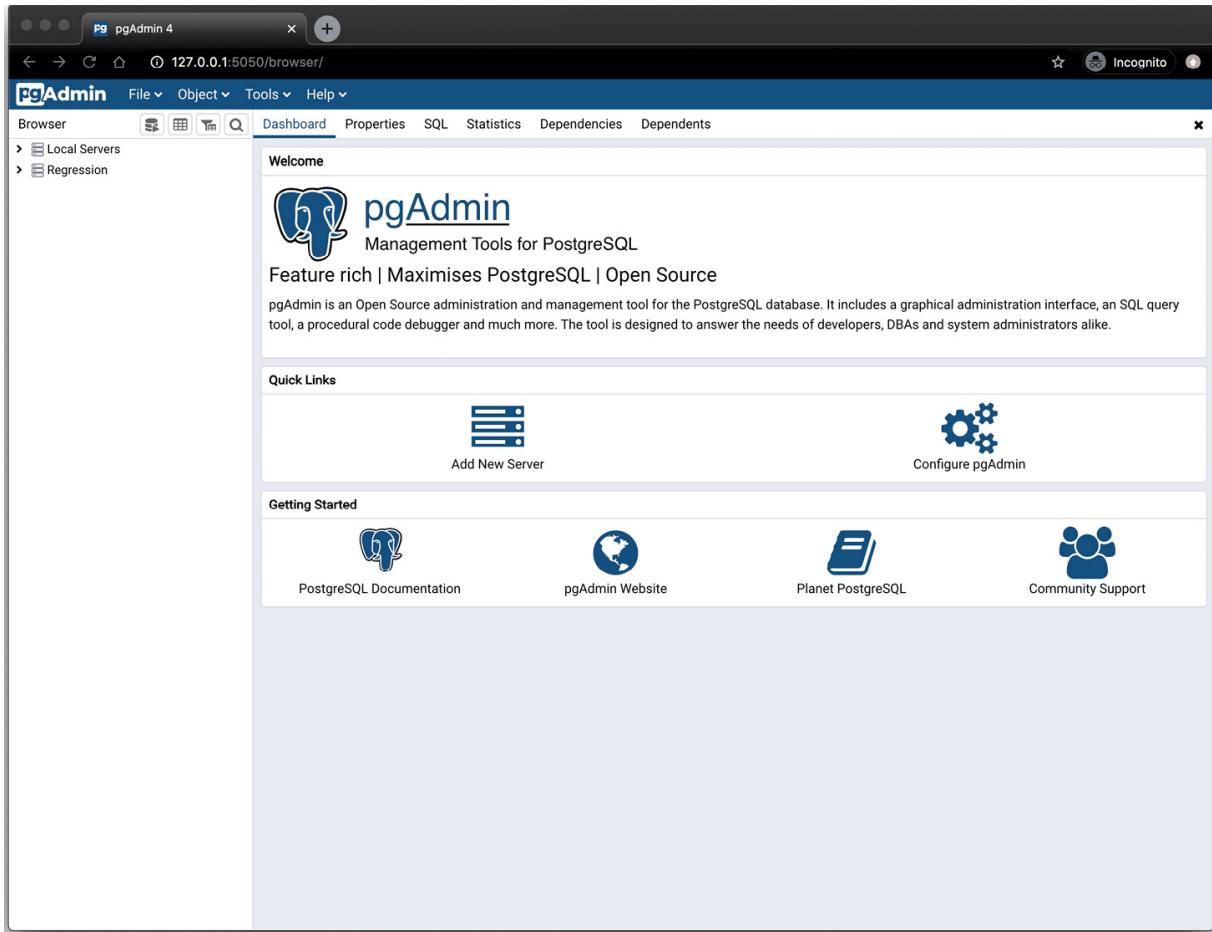


Figure 4.1: pgAdmin 4 user interface

pgAdmin provides a graphical user interface to manage server resources and write queries. To learn more and download pgAdmin, visit the following URL: <https://www.pgadmin.org/>.

Storing data in ultra-fast HDF5 format

Hierarchical Data Format (HDF) is made up of a collection of file formats, namely HDF4 and HDF5, engineered for the hierarchical storage and management of voluminous data. Initially developed at the U.S. National Center for Supercomputing Applications, HDF5 is an open source format that accommodates large and complex heterogeneous datasets. It uses a directory-like structure, enabling versatile data organization within the file, similar to file management on a computer. HDF5 has two primary object types: datasets, which are typed multidimensional arrays, and groups, which are container structures capable of holding both datasets and other groups. In Python, HDF5 is supported through two libraries: `h5py`, offering both high- and low-level access to HDF5 constructs, and `PyTables`, providing a high-level interface with advanced indexing and query

capabilities. We'll use pandas to write to and read data in this recipe, which uses `PyTables` under the hood. `PyTables` is a Python library for managing large datasets and hierarchical databases using the HDF5 file format. It provides tools for efficiently storing, accessing, and processing data, making it well-suited for high-performance, data-intensive applications.

Storing data using HDF5 is advantageous when storing related data in a hierarchy. This could be different fundamental data for stocks, expirations for futures, or chains for options.

Getting ready...

To follow this recipe, you'll need `PyTables` installed on your computer, which pandas uses to access data in HDF5 file format.

You can install `PyTables` using `conda` like this:

```
conda install -c conda-forge pytables
```

How to do it...

For this recipe, we'll move back to our Jupyter notebook:

1. Import the necessary libraries and set up the variables:

```
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
STOCK_DATA_STORE = "stocks.h5"
FUTURES_DATA_STORE = "futures.h5"
ticker = "SPY"
root = "CL"
```

2. Load data for `SPY` price and options chains using the OpenBB Platform:

```
spy_equity = obb.equity.price.historical(
    ticker,
    start_date="2021-01-01",
    provider="yfinance"
)
spy_chains = obb.derivatives.options.chains(
    ticker,
    provider=".cboe"
)
spy_expirations = (
    spy_chains
    .expiration
    .astype(str)
    .unique()
    .tolist()
)
spy_historic = (
    obb
```

```

        .equity
        .price
        .historical(
            ticker + spy_expirations[-10].replace("-", "")[2:] + "C"
            + "00400000",
            start_date="2021-01-01",
            provider="yfinance"
        )
    )
)

```

3. Store the data in the HDF5 file:

```

with pd.HDFStore(STOCKS_DATA_STORE) as store:
    store.put("equities/spy/stock_prices", spy_equity)
    store.put("equities/spy/options_prices",
              spy_historic)
    store.put("equities/spy/chains", spy_chains)

```

4. Read the data from the HDF5 file into pandas DataFrames:

```

with pd.HDFStore(STOCKS_DATA_STORE) as store:
    spy_prices = store["equities/spy/stock_prices"]
    spy_options = store["equities/spy/options_prices"]
    spy_chains = store["equities/spy/chains"]

```

5. Now iterate through e-mini futures expirations, storing the historical data for each one in a different file path:

```

with pd.HDFStore(FUTURES_DATA_STORE) as store:
    for i in range(24, 31):
        expiry = f"20{i}-12"
        df = obb.derivatives.futures.historical(
            symbol=[root],
            expiry=expiry,
            start_date="2021-01-01",
        )
        df.rename(
            columns={
                "close": expiry
            },
            inplace=True
        )
        prices = df[expiry]
        store.put(f'futures/{root}/{expiry}', prices)

```

6. Read the data the same as with the ETF:

```

with pd.HDFStore(FUTURES_DATA_STORE) as store:
    es_prices = store[f"futures/{root}/2023-12"]

```

How it works...

We first download historic price data and options chains for the SPY ETF using the OpenBB Platform. We extract the expirations then use `obb.equity.price.historical` to construct an options ticker symbol to request historic data. With this data stored in pandas DataFrames, we open the `assets.h5` file using the pandas `HDFStore` method. The Python `with` statement creates a context that allows you to run a group of statements under the control of a context manager. Here, we open the

`assets.h5` file as a pandas `HDFStore` object. `HDFStore` has a method called `put`, which allows us to easily store the data in the DataFrame in the HDF5 file.

The futures example shows how you can iterate through a list of data sources, saving each one as a separate path within the HDF5 file. We start by opening the HDF5 file and iterating through a list of expiration dates. For each expiration date, we use the OpenBB Platform to download price data, rename the columns, and put the data into the HDF5 file.

There's more...

HDF5 is considered one of the fastest on-disk, columnar data storage formats for strictly numerical data. This format also generally shares the smallest memory footprint with compressed CSV format. Another file format is Parquet, which is a binary, columnar storage format that provides efficient data compression and encoding. It's also available through pandas using the PyArrow library under the hood.

See also...

To learn more about HDF, visit the documentation here:

- Documentation for pandas `HDFStore`: <https://pandas.pydata.org/docs/reference/api/pandas.HDFStore.put.html>
- Documentation for PyTables: <https://www.pytables.org>
- Documentation for the Python bindings of PyArrow: <https://arrow.apache.org/docs/python/index.html>.

5

Build Alpha Factors for Stock Portfolios

Professional traders often construct factor portfolios to target and exploit market inefficiencies, such as anomalies in value, size, or momentum, to generate better risk-adjusted returns. By systematically identifying and weighing securities based on these specific characteristics or factors, investors can create a portfolio that captures the desired exposures while minimizing unintended risks. Factors act as the fundamental building blocks of investing, being the persistent forces that influence returns across various asset classes. A trading edge is a consistent, non-random inefficiency in the market that can be exploited for profit. Factors are the inefficiencies that drive asset prices and form the basis of this edge, allowing traders to capitalize on these persistent anomalies.

Factor analysis is a broad topic but comes down to identifying the factors, determining the sensitivity of a portfolio to those factors, and taking action. That action can be hedging undesirable risk based on the factor exposure or increasing exposure to the factor. In this chapter, we explore the key elements of identifying factors, hedging out unwanted risks, and setting up forward returns to assess the predictive power of factors. We'll use Python libraries for statistical modeling to build a principal component analysis and linear regression. Then, we'll introduce the Zipline Reloaded Pipeline API, which will prepare us for analyzing factors.

In this chapter, we present the following recipes:

- Identifying latent return drivers using principal component analysis
- Finding and hedging portfolio beta using linear regression
- Analyzing portfolio sensitivities to the Fama-French factors
- Assessing market inefficiency based on volatility
- Preparing a factor ranking model using the Zipline Reloaded Pipeline API

Identifying latent return drivers using principal component analysis

Principal component analysis (PCA) is a dimensionality reduction technique that is widely used in data science. It transforms the original features into a new set of features, called **principal components**, which reflect the maximum variance in the data. In other words, it transforms a large set of variables into a smaller set of variables, while still containing most of the information from the larger set.

There are various sources of risk in an asset portfolio, including market risk, sector risk, and asset-specific risk. PCA helps identify and quantify these risks by breaking down the returns of the portfolio into components that explain the maximum variance. The first few principal components usually capture most of the variance and they can be analyzed to understand the major sources of risk in the portfolio.

This recipe will use scikit-learn to run PCA on a portfolio of eight stocks made of up mining and technology companies.

Getting ready

For this recipe, we introduce scikit-learn, which provides algorithms for classification, regression, clustering, dimensionality reduction, and more. It's built on NumPy, SciPy, and Matplotlib, which makes it easy to integrate with other scientific Python libraries. We'll use it to conduct our analysis.

You can install scikit-learn using `pip`:

```
pip install scikit-learn
```

How to do it...

Scikit-learn makes it easy to run a PCA. Here's how to do it:

1. Import the libraries we need for the analysis:

```
import numpy as np
import pandas as pd
from openbb import obb
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
obb.user.preferences.output_type = "dataframe"
```

2. Download data for gold mining stocks and healthcare stocks and compute their daily returns:

```
symbols = ["NEM", "RGLD", "SSRM", "CDE", "LLY", "UNH",
           "JNJ", "MRK"]
data = obb.equity.price.historical(
    symbols,
    start_date="2020-01-01",
    end_date="2022-12-31",
    provider="yfinance",
).pivot(columns="symbol", values="close")
returns = data.pct_change().dropna() Run the PCA using three components and fit the
model:
pca = PCA(n_components=3)
pca.fit(returns)
```

3. Extract the explained variance ratio for each component and extract the principal components:

```
pct = pca.explained_variance_ratio_
```

```
pca_components = pca.components_
```

4. Plot the contribution of each principal component and the cumulative percent of explained variance:

```
cum_pct = np.cumsum(pct)
x = np.arange(1, len(pct) + 1, 1)
plt.subplot(1, 2, 1)
plt.bar(x, pct * 100, align="center")
plt.title("Contribution (%)")
plt.xticks(x)
plt.xlim([0, 4])
plt.ylim([0, 100])
plt.subplot(1, 2, 2)
plt.plot(x, cum_pct * 100, "ro-")
plt.title("Cumulative contribution (%)")
plt.xticks(x)
plt.xlim([0, 4])
plt.ylim([0, 100])
```

The last code block results in the following chart:

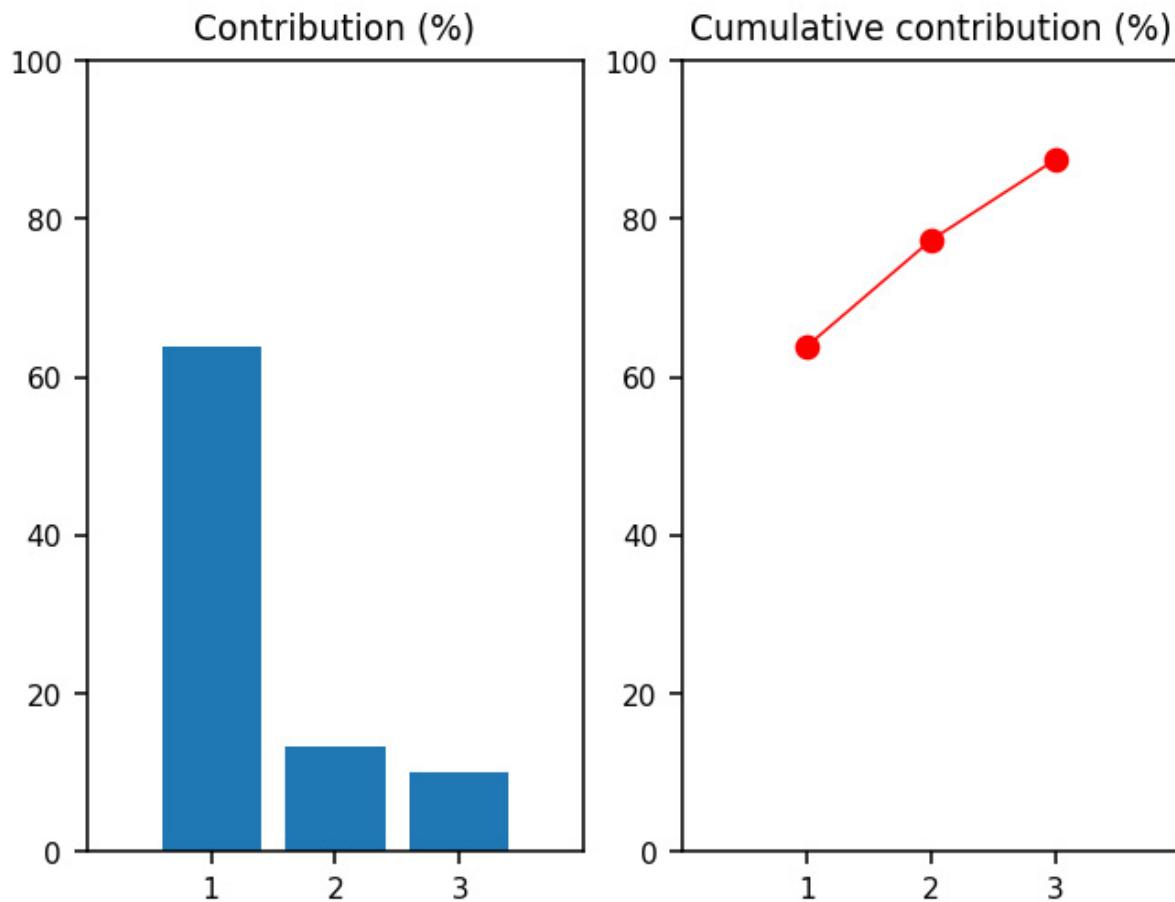


Figure 5.1: Contribution and cumulative contribution of explained variance for the first three principal components

How it works...

There is ample documentation on how PCA works so we will not cover the math behind the process. But in summary, the `pca(n_components=3)` code creates a PCA object that will compute the first three principal components of the `returns` data. The `n_components=3` parameter specifies that we want to select the top three principal components. The call to `pca.fit` fits the PCA model to the `returns` data, which hides most of the complexity of PCA. Under the hood, PCA standardizes the returns and computes a covariance matrix. Next, the eigen decomposition is performed on the covariance matrix, which computes the eigenvectors and eigenvalues that indicate the directions and magnitude of data variance. Finally, eigenvectors are ordered by their eigenvalues in descending order and the top three are chosen as principal components.

There's more...

From the principal components, we can transform the original `returns` data into a new set of features that represent the statistical risk factors that explain the most variance in the returns, as shown in the following code:

```
X = np.asarray(returns)
factor_returns = X.dot(pca_components.T)
factor_returns = pd.DataFrame(
    columns=["f1", "f2", "f3"],
    index=returns.index,
    data=factor_returns
)
```

The preceding code will return the following DataFrame containing a time series of the statistical risk factors:

Date	f1	f2	f3
2020-01-03	-0.028845	0.007909	-0.013464
2020-01-06	-0.081169	-0.006508	0.058871
2020-01-07	0.003694	0.015644	0.013973
2020-01-08	-0.105891	-0.028187	0.005725
2020-01-09	0.001918	-0.018469	-0.023525

Figure 5.2: DataFrame containing a time series of the statistical risk factors

The statistical risk factors are similar to more conventional risk factors such as the Fama-French factors we'll explore in the *Analyzing portfolio sensitivities to the Fama-French factors* recipe in this chapter. The `returns` data is projected onto the principal components obtained from the PCA. This is

done by taking the dot product of `x` with the transpose of the `pca_components` matrix. The `pca_components` matrix contains the principal components of the `returns` data, which were previously computed using the `fit` method.

This step transforms the original `returns` data into a new set of features (the principal components or factors) that explain the most variance in the data. These factors should give us an idea of how much of the portfolio's returns come from some unobservable statistical feature.

With the principal components, we can create the exposure of each asset to the three principal components:

```
factor_exposures = pd.DataFrame(
    index=["f1", "f2", "f3"],
    columns=returns.columns,
    data=pca_components
) .T
```

The resulting `factor_exposures` DataFrame is shown in the following screenshot. It shows how much each asset in the portfolio is exposed to each of the three factors. The exposure of each asset to each of the three factors ("f1", "f2", "f3") represents how much the returns of that asset are influenced by changes in those factors.

	f1	f2	f3
NEM	0.292250	0.091439	0.394104
RGLD	0.308669	0.125098	0.416761
SSRM	0.427029	0.227828	0.485778
CDE	0.783970	-0.030216	-0.618241
LLY	0.062208	-0.605558	0.131696
UNH	0.101891	-0.535551	0.136475
JNJ	0.061262	-0.338930	0.108649
MRK	0.065923	-0.393423	0.070823

Figure 5.3: DataFrame containing the exposure of each asset to each factor

We can visualize each asset's exposure to the first two principal components on an annotated scatter plot:

```
labels = factor_exposures.index
data = factor_exposures.values
plt.scatter(data[:, 0], data[:, 1])
plt.xlabel("factor exposure of PC1")
plt.ylabel("factor exposure of PC2")
for label, x, y in zip(labels, data[:, 0], data[:, 1]):
```

```

plt.annotate(
    label,
    xy=(x, y),
    xytext=(-20, 20),
    textcoords="offset points",
    arrowprops=dict(
        arrowstyle="->",
        connectionstyle="arc3, rad=0"
    ),
)

```

The result is the following scatter plot showing the exposure to the factors:

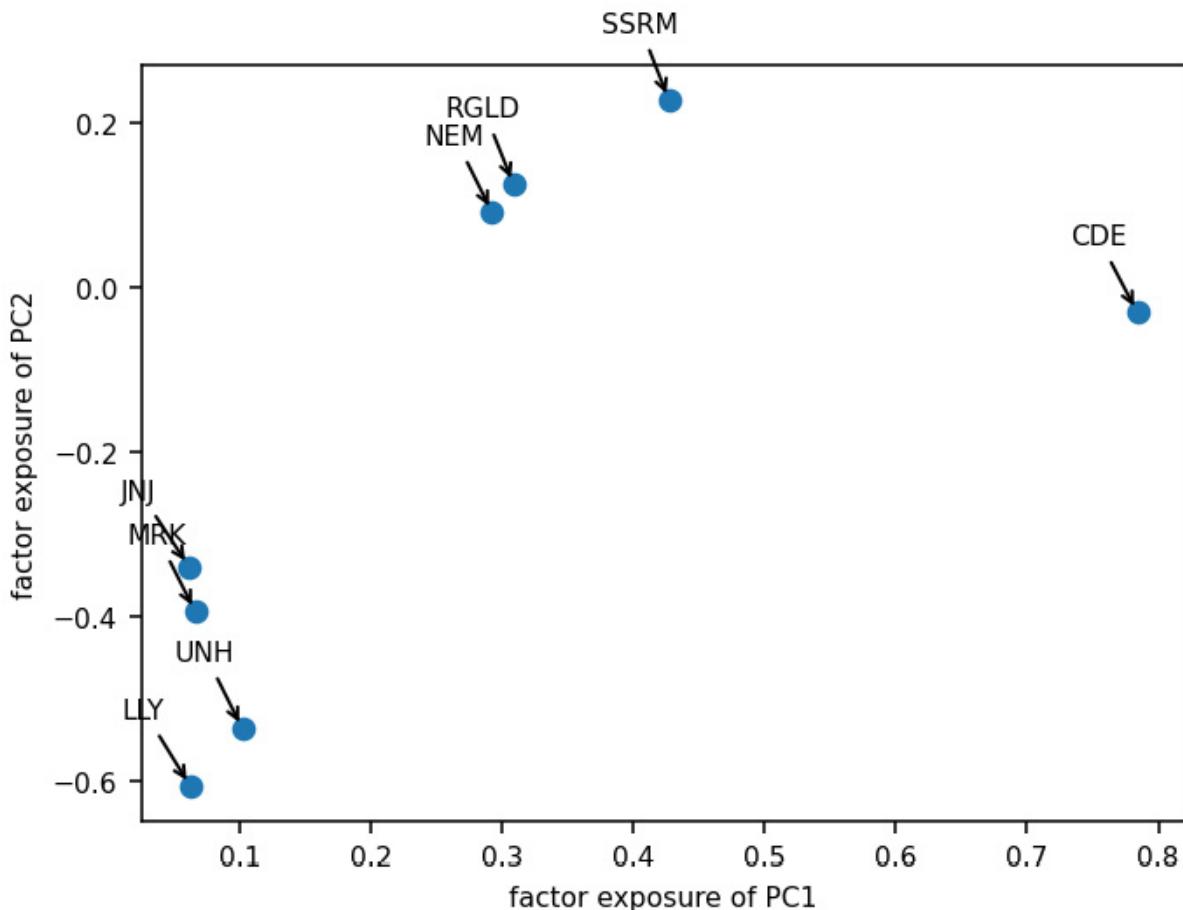


Figure 5.4: Scatter plot showing the stock exposure to the first two principal components

The gold mining stocks have higher factor exposures compared to the healthcare stocks, indicating that they are more sensitive to changes in the underlying factors. The healthcare stocks have lower and negative exposures to "f2", indicating that they may provide some diversification benefits in a portfolio that is highly exposed to this factor. The factors may represent different sources of systematic risk in the market, such as market risk, interest rate risk, or industry-specific risks.

See also

PCA is an important dimensionality reduction technique commonly used in portfolio management. You can visit the following links to learn more about PCA:

- More on PCA: https://en.wikipedia.org/wiki/Principal_component_analysis
- Documentation for scikit-learn: <https://scikit-learn.org/stable/>
- Documentation for the scikit-learn implementation of PCA used in this recipe: <https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html>

Finding and hedging portfolio beta using linear regression

Algorithmic traders often seek exposure to specific risks that they believe will yield outsized returns while hedging other risks they deem unfavorable or unnecessary. For instance, a trader might want exposure to stocks with the lowest price-to-earnings ratios, believing they will outperform while hedging against the broader market risk. This selective exposure helps traders maximize returns by capitalizing on perceived opportunities while minimizing the potential downside by hedging against certain risks.

Factor models are a way of explaining the returns of an asset or portfolio through a combination of the returns of another asset, portfolio, or factor. The general form of a factor model using a linear combination is as follows:

$$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_n X_n$$

The sensitivity of portfolio returns to a risk factor X is described by the beta. It's the beta that must be hedged to concentrate exposure to the risk factors.

In this recipe, we'll construct a portfolio of stocks, compute the beta of the portfolio returns to `SPY` returns, and construct a hedging portfolio to neutralize the broader market exposure.

Getting ready

For this recipe, we introduce Statsmodels, which provides classes and functions for estimating many statistical models and for conducting statistical tests and statistical data exploration. We'll use it to conduct our analysis.

You can install Statsmodels using `pip`:

```
pip install statsmodels
```

How to do it...

We'll use Statsmodels to measure the sensitivity of a portfolio of stocks to a benchmark, then offset the portfolio with a short position to hedge beta:

1. Import the libraries for the analysis:

```
import numpy as np
import pandas as pd
from openbb import obb
import statsmodels.api as sm
from statsmodels import regression
import matplotlib.pyplot as plt
obb.user.preferences.output_type = "dataframe"
```

2. Download data for the same portfolio we used in the previous recipe. Note the inclusion of the **SPY exchange-traded fund (ETF)**, which we'll use to represent the broad market returns:

```
symbols = ["NEM", "RGLD", "SSRM", "CDE", "LLY", "UNH", "JNJ", "MRK", "SPY"]
data = obb.equity.price.historical(
    symbols,
    start_date="2020-01-01",
    end_date="2022-12-31",
    provider="yfinance"
).pivot(columns="symbol", values="close")
```

3. Pop the column with **SPY** data off the DataFrame and compute the returns:

```
benchmark_returns = (
    data
    .pop("SPY")
    .pct_change()
    .dropna()
)
```

4. Now, compute the returns for the portfolio:

```
portfolio_returns = (
    data
    .pct_change()
    .dropna()
    .sum(axis=1)
)
```

5. Set the **name** property on the series for plotting purposes:

```
portfolio_returns.name = "portfolio"
```

6. Plot the portfolio returns and the benchmark returns to visualize the difference:

```
portfolio_returns.plot()
benchmark_returns.plot()
plt.ylabel("Daily Return")
plt.legend()
```

7. The result is a plot of the daily returns during the analysis period:

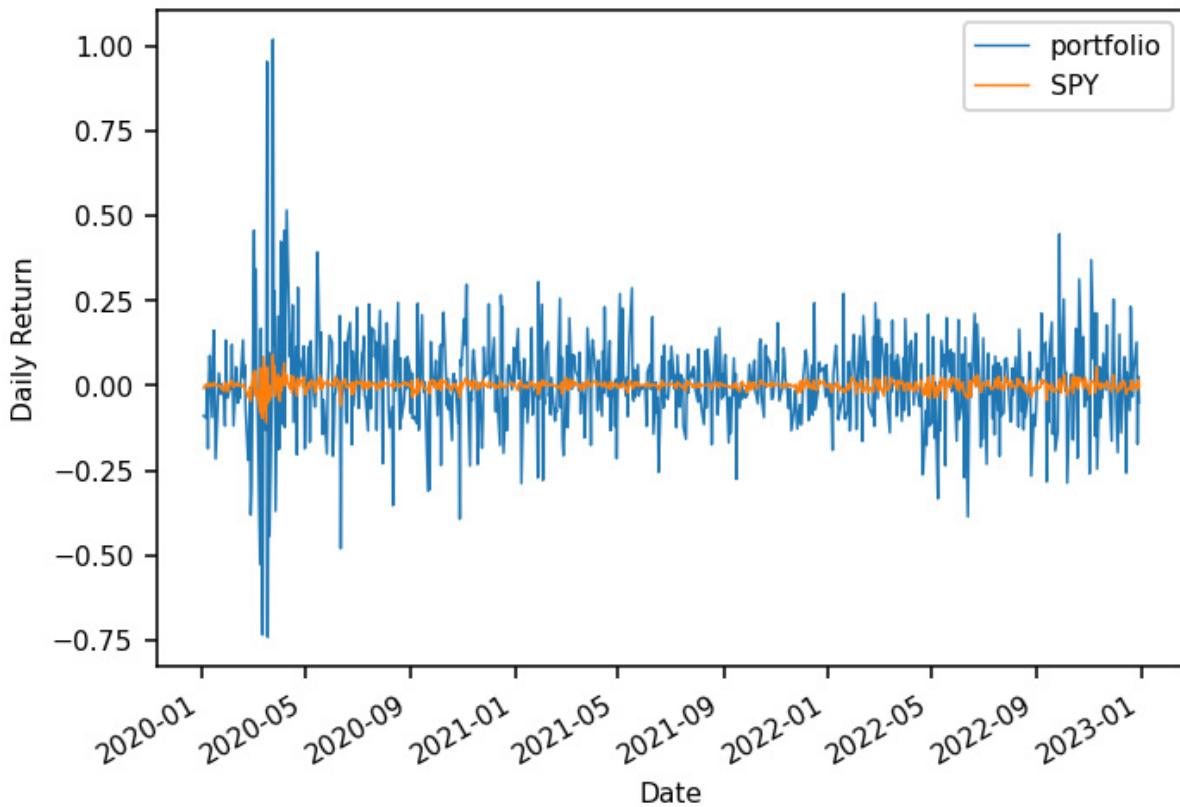


Figure 5.5: Plot of portfolio and benchmark returns

8. Create a function that returns the alpha and beta coefficients from a linear regression:

```
X = benchmark_returns.values
Y = portfolio_returns.values
def linreg(x, y):
    x = sm.add_constant(x)
    model = regression.linear_model.OLS(y, x).fit()
    x = x[:, 1]
    return model.params[0], model.params[1]
```

9. Generate the alpha and beta coefficients from the regression between the portfolio and benchmark returns:

```
alpha, beta = linreg(X, Y)
print(f"Alpha: {alpha}") # => 0.0028
print(f"Beta: {beta}") # => 5.5745
```

10. Create a scatter plot that visualizes the linear relationship between the portfolio and benchmark returns:

```
X2 = np.linspace(X.min(), X.max(), 100)
Y_hat = X2 * beta + alpha
plt.scatter(X, Y, alpha=0.3)
plt.xlabel("SPY daily return")
plt.ylabel("Portfolio daily return")
plt.plot(X2, Y_hat, "r", alpha=0.9)
```

The result is the following scatter plot showing the daily returns and their linear relationship:

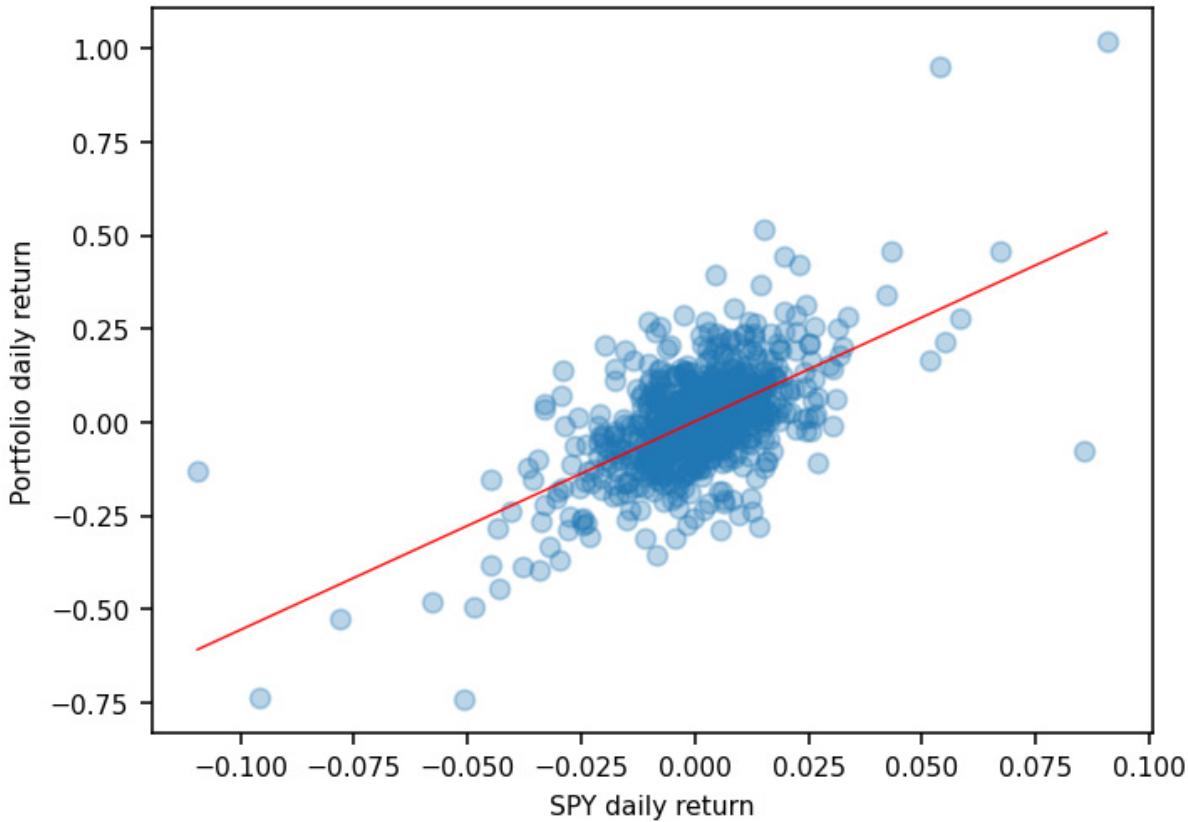


Figure 5.6: Scatter plot showing the linear relationship between the portfolio and benchmark returns

11. Finally, construct a time series of portfolio returns including the beta hedge:

```
hedged_portfolio_returns = -1 * beta * benchmark_returns + portfolio_returns
```

12. Rerun the regression to confirm that the beta is 0:

```
P = hedged_portfolio_returns.values
_, beta = linreg(X, P)
print(f"Beta: {beta}") => 0.0000
```

How it works...

We demonstrate the beta hedge by first constructing a portfolio of eight stocks by adding up the daily returns of each stock for each day. We use the S&P 500 tracking ETF as a proxy for the broader market. The ETF symbol is SPY. We download price data for SPY along with our portfolio for convenience since we can use the pandas `pop` method in the SPY column as a separate series. We then compute the daily returns for both the portfolio and the benchmark.

The next step is running a regression that is a few lines of code. `x` is the independent variable representing the benchmark returns, and `y` is the dependent variable representing the portfolio returns.

The `linreg` function takes these returns, adds a constant term to `x` for the intercept, and then fits a linear model using ordinary least squares regression. It then returns the parameters of the fitted model, which are the intercept (alpha) and the slope (beta).

IMPORTANT NOTE

Adding a constant term for the intercept is crucial for accurately modeling the relationship between the dependent and independent variables in a linear regression. If we do not include a constant term for the intercept, we are essentially forcing the regression line to pass through the origin (0, 0), which may not be an accurate representation of the relationship between the variables.

After we have our alpha and beta terms, we can build a hedged portfolio. As we learned in the introduction to this recipe, traders hedge exposure to risk that they don't want. If we can determine that our portfolio returns are linked to a risk factor through a linear relationship, then we can initiate a short position in that risk factor – in our case, the broader market represented by SPY. The amount we want to trade is represented by our beta. This is effective because if our returns are made up of $\alpha + \beta\text{SPY}$, then going short βSPY will result in our new returns being $\alpha + \beta\text{SPY} - \beta\text{SPY} = \alpha$. That is, no exposure to the returns of SPY. We validate that the beta of the hedged portfolio is 0 by rerunning the regression.

There's more...

The information ratio is a measure used to evaluate the risk-adjusted performance of a trading strategy. It is calculated by dividing the portfolio's active return (the difference between the portfolio return and the benchmark return) by the active risk (the standard deviation of the active return). The information ratio is a useful way to compare two portfolios against their benchmark. Let's build a function that computes it:

```
def information_ratio(
    portfolio_returns,
    benchmark_returns
):
    active_return = portfolio_returns - benchmark_returns
    tracking_error = active_return.std()
    return active_return.mean() / tracking_error
```

Let's use the information ratio to compare the differences between our hedged and unhedged portfolios:

```
hedged_ir = information_ratio(
    hedged_portfolio_returns,
    benchmark_returns
)
unhedged_ir = information_ratio(
    portfolio_returns,
    benchmark_returns
```

```
)  
print(f"Hedged information ratio: {hedged_ir}")  
print(f"Unhedged information ratio: {unhedged_ir}")
```

Running the preceding code block shows us the unhedged portfolio has a lower risk-adjusted return. That's to be expected since the returns attributable to the benchmark have been hedged.

See also

Hedging is an important part of proper risk management and successful trading. Using a simple linear relationship between the portfolio returns and the factor returns is a great way to neutralize unwanted risk. To learn more about the topic that we covered in this recipe, visit the following links:

- Statsmodels documentation on ordinary least squares: <https://www.statsmodels.org/stable/examples/notebooks/generated/ols.html>
- More about how beta is used in portfolio management and trading: <https://www.investopedia.com/terms/b/beta.asp>
- More about how to use the information ratio: <https://pyquantnews.com/how-to-measure-skill-portfolio-manager/>

Analyzing portfolio sensitivities to the Fama-French factors

The Fama-French factors are a set of factors identified by economists Eugene F. Fama and Kenneth R. French to explain the variation in stock returns. These factors serve as the foundation of the Fama-French three-factor model. In [*Chapter 1, Acquire Free Financial Market Data with Cutting-edge Python Libraries*](#), we learned how to use `pandas_datareader` to download Fama-French factor data.

In this recipe, we'll compute the exposure of our portfolio to the size and market value factors.

Getting ready

You should already have `pandas_datareader` installed from [*Chapter 1, Acquire Free Financial Market Data with Cutting-edge Python Libraries*](#). If not, you can install it using `pip`:

```
pip install pandas_datareader
```

We'll also assume you have the `data` DataFrame loaded with historic prices created in the last recipe.

How to do it...

We'll reuse what we learned in the prior recipes about using beta to compute factor sensitivities.

1. Import the libraries we need for the analysis:

```
import numpy as np
```

```

import pandas as pd
import pandas_datareader as pdr
from openbb import obb
import statsmodels.api as sm
from statsmodels import regression
from statsmodels.regression.rolling import RollingOLS
obb.user.preferences.output_type = "dataframe"

```

2. Now, resample the daily return data to monthly using `asfreq` and replace the row labels to monthly format using `to_period`.

This is the format of the Fama-French factor data. A consistent format lets us align the data in the same DataFrame later:

```

monthly_returns = (
    data
    .asfreq("M")
    .dropna()
    .pct_change(fill_method=None)
    .to_period("M")
)

```

3. The result is a DataFrame with monthly returns suitable for alignment with the Fama-French monthly factor data:

	NEM	RGLD	SSRM	CDE	LLY	UNH	JNJ	MRK	SPY
Date									
2015-01	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2015-02	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2015-03	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2015-04	0.220175	0.022500	0.203090	0.108280	-0.010736	-0.058247	-0.013917	0.036187	0.009834
2015-05	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
...
2022-07	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2022-08	-0.306854	-0.139352	-0.188644	-0.092105	-0.067904	0.011098	-0.084986	-0.063727	0.047528
2022-09	0.029778	0.020892	0.090437	0.239130	0.073432	-0.024465	0.012520	0.017037	-0.092446
2022-10	0.006900	0.015798	-0.061863	0.105263	0.119808	0.099220	0.064949	0.175104	0.081275
2022-11	0.121692	0.182919	0.103917	-0.074074	0.027687	-0.013312	0.029769	0.088142	0.055592

Figure 5.7: DataFrame with monthly portfolio returns

4. Next, we compute the active returns and use them as the dependent variable. This will help us understand the sensitivity of the Fama-French factors to our active returns:

```

bench = monthly_returns.pop("SPY")
R = monthly_returns.mean(axis=1)
active = (R - bench).dropna()

```

5. Now, let's download the Fama-French data using `pandas_datareader`:

```

factors = pdr.get_data_famafrench(
    'F-F_Research_Data_Factors',
    start="2015-01-01",
    end="2022-12-31"
) [0] [1:] / 100

```

```
SMB = factors.loc[active.index, "SMB"]
HML = factors.loc[active.index, "HML"]
```

6. Now that we have our returns and factor data, build a pandas DataFrame that aligns the data along the common date index:

```
df = pd.DataFrame(
    {
        "R": active,
        "SMB": SMB,
        "HML": HML,
    },
    index=active.index
).dropna()
```

7. Run the regression and get the beta coefficients for each factor. These coefficients represent the exposure of our active returns to the Fama-French factors:

```
b1, b2 = regression.linear_model.OLS(
    df.R,
    df[["SMB", "HML"]]
).fit().params
```

8. To see how these sensitivities evolve through time, use the Statsmodels **RollingOLS** class:

```
exog = sm.add_constant(df[["SMB", "HML"]])
rols = RollingOLS(active, exog, window=12)
rres = rols.fit()
fig = rres.plot_recursive_coefficient(
    variables=["SMB", "HML"],
    figsize=(5.5, 6.6)
)
```

This results in the following two charts that plot the 12-month rolling regression over time along with the 95% confidence intervals:

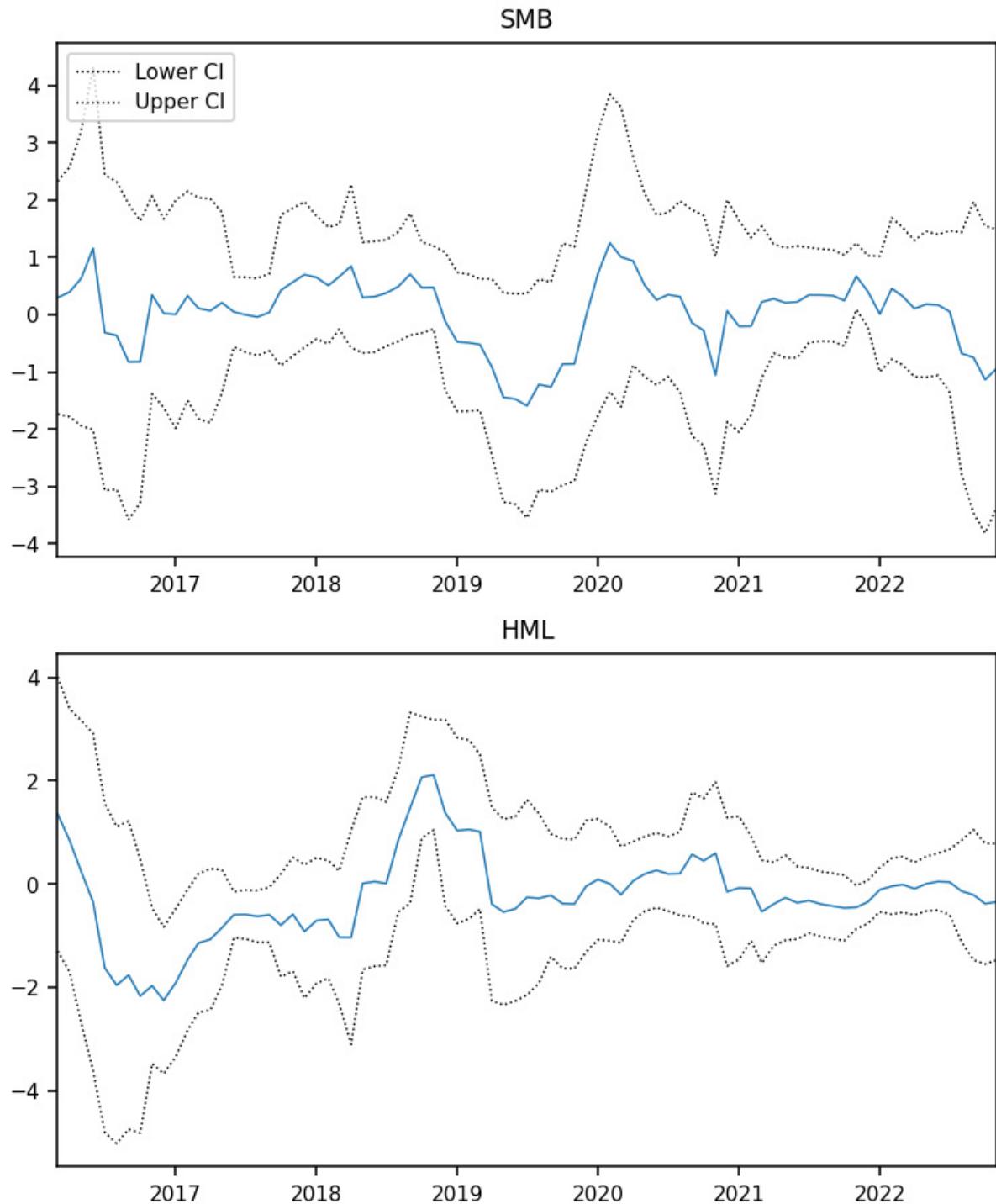


Figure 5.8: Betas for two of the Fama-French factors using a 12-month rolling window

How it works...

We follow the same procedure in downloading financial market data using OpenBB and computing portfolio returns as in the *Finding and hedging portfolio beta using linear regression* recipe in this chapter. The Fama-French factor data is in monthly resolution, so we use `asfreq` and `to_period` to resample our data price data to monthly, then compute monthly returns.

From there, we compute active returns, which are the portion of portfolio returns that cannot be attributed to the market's overall movement and are instead a result of active portfolio management decisions. As we learned, investment factors are systematic risk factors that explain the differences in returns between different securities. A portfolio's active return can be influenced by its exposure to these factors.

To measure the exposure, we compute the beta coefficients of the factors using the `OLS` method from the `Statsmodels` library to fit a linear regression model. The dependent variable is `df.R`, which represents the active returns of our portfolio, while the independent variables are `df["SMB"]` and `df["HML"]`, representing the **Small Minus Big (SMB)** and **High Minus Low (HML)** factors of the Fama-French three-factor model, respectively. The `fit` method is used to estimate the parameters of the model, which, in this case, are the betas for the `SMB` and `HML` factors stored in `b1` and `b2`.

To get an idea of how the exposures evolve through time, we build a rolling regression using the `RollingOLS` class. The exogenous variable is created by adding a constant column (for the intercept) to the `SMB` and `HML` factors. The `RollingOLS` class is then initialized with a window size of `12`, and the `fit` method is called to compute the rolling OLS estimates. Finally, the `plot_recursive_coefficient` method is used to create a plot of the rolling estimates of the regression coefficients for `SMB` and `HML` over the 12-period window.

There's more...

Marginal Contribution to Active Risk (MCAR) quantifies the additional active risk each factor brings to your portfolio. To compute the MCAR of the factors, we multiply the factor sensitivity by the covariance between the factors and then divide by the square of the standard deviation of the active returns. This calculation shows the amount of risk incurred by being exposed to each factor, considering the exposures to other factors already in your portfolio. The risk contribution that is not explained is the exposure to factors other than the ones being analyzed.

```
F1 = df.SMB
F2 = df.HML
cov = np.cov(F1, F2)
ar_squared = (active.std()) ** 2
mcar1 = (b1 * (b2 * cov[0, 1] + b1 * cov[0, 0])) / ar_squared
mcar2 = (b2 * (b1 * cov[0, 1] + b2 * cov[1, 1])) / ar_squared
print(f"SMB risk contribution: {mcar1}")
```

```

print(f"HML risk contribution: {mcar2}")
print(f"Unexplained risk contribution: {
      1 - (mcar1 + mcar2)}")

```

Running the preceding code shows only about 0.7% of the risk is explained by these two factors.

Let's plot the MCAR for these factors over time:

1. Compute the 12-month rolling covariances between the factors:

```

covariances = (
    df[["SMB", "HML"]]
    .rolling(window=12)
    .cov()
) .dropna()

```

2. Compute the 12-month rolling active return squared:

```

active_risk_squared = (
    active.rolling(window=12).std()**2
) .dropna()

```

3. Combine the rolling factor betas:

```

betas = pd.concat(
    [rres.params.SMB, rres.params.HML],
    axis=1
) .dropna()

```

4. Create an empty DataFrame to store the rolling MCAR data:

```

MCAR = pd.DataFrame(
    index=betas.index,
    columns=betas.columns
)

```

5. Loop through each factor and each date computing the MCAR value:

```

for factor in betas.columns:
    for t in betas.index:
        s = np.sum(
            betas.loc[t] * covariances.loc[t][factor])
        b = betas.loc[t][factor]
        AR = active_risk_squared.loc[t]
        MCAR[factor][t] = b * s / AR

```

6. Finally, plot the MCAR for each factor as follows:

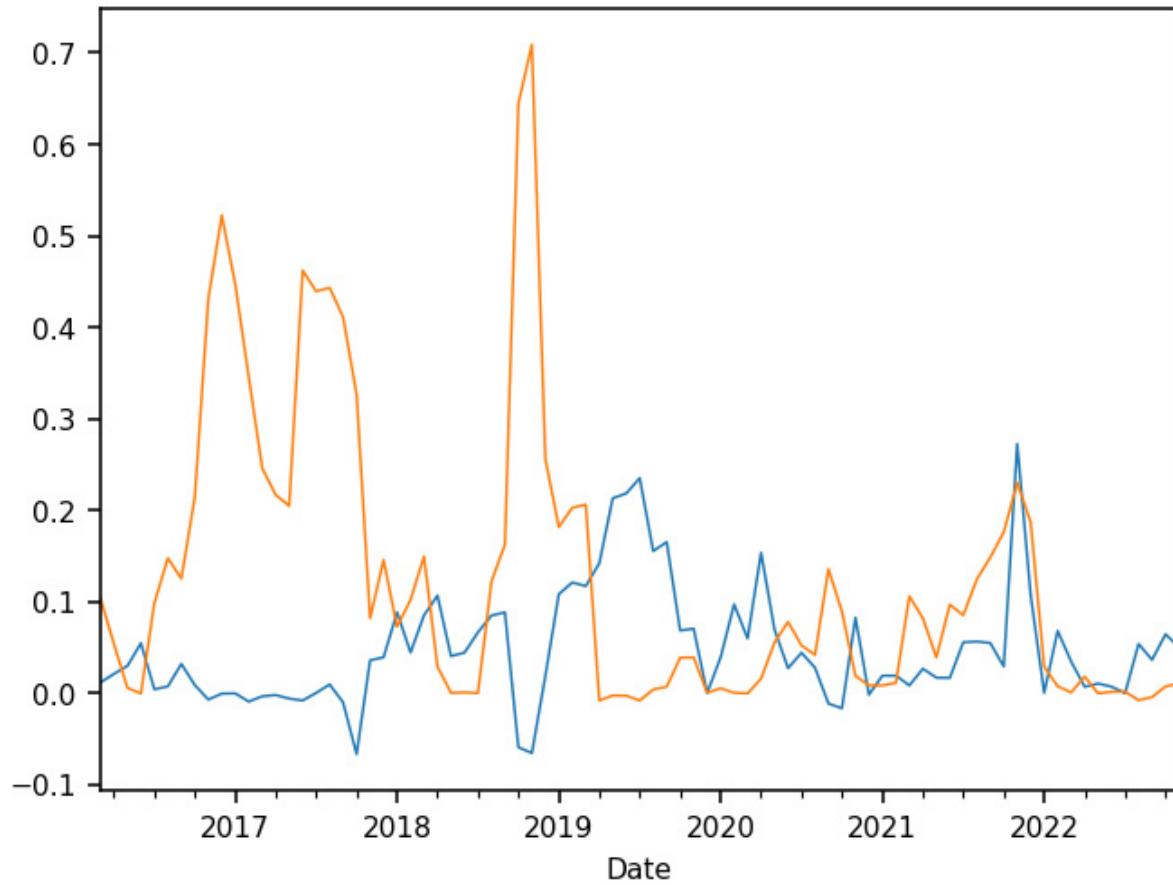


Figure 5.9: Rolling marginal contribution to risk for the two Fama-French factors

See also

By systematically selecting securities with certain factor characteristics, investors aim to achieve better risk-adjusted returns compared to the broader market or a specific benchmark. Factor investing is a broad and deep topic. To learn more about factor investing, visit the following links:

- BlackRock's introduction to factor investing: <https://www.blackrock.com/us/individual/investment-ideas/what-is-factor-investing>
- *Advances in Active Portfolio Management* by Richard Grinold and Ronald Kahn is an advanced guide that delves deep into the quantitative aspects of active portfolio management, including topics such as portfolio construction, risk management, and implementation: <https://amzn.to/44zJeMk>

Assessing market inefficiency based on volatility

Using volatility as a factor reflects the market inefficiency related to the pricing of volatile stocks. Historically, stocks with lower volatility have tended to outperform those with higher volatility on a

risk-adjusted basis. This phenomenon contradicts the traditional finance theory that higher risk should be compensated with higher return and thus represents a market inefficiency.

One of the first steps in analyzing a factor's performance is calculating forward returns. Forward returns are the returns of a security in a future period. Once the forward returns are calculated, we can use the Spearman rank correlation to understand the relationship between the factor and the forward returns. The Spearman rank correlation assesses how well the relationship between two variables can be described using a monotonic function. A high Spearman rank correlation indicates that the factor ranks securities in a way that is closely aligned with the ranking of securities by their forward returns, suggesting that the factor has predictive power. Conversely, a low Spearman rank correlation suggests that the factor is not a good predictor of forward returns.

The Parkinson estimator uses the high and low prices over a period to get a more accurate measure of volatility. The basic idea is that the high and low prices encapsulate more information about the price movement than just the closing prices. In this recipe, we'll build a factor using Parkinson volatility, compute the forward returns, and determine the Spearman rank correlation between the factor and forward returns.

How to do it...

We'll calculate Parkinson volatility and forward returns to measure the predictive power of Parkinson volatility as a tradeable factor:

1. Import the libraries we'll use for the analysis:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from openbb import obb
from scipy.stats import spearmanr
obb.user.preferences.output_type = "dataframe"
```

2. Use OpenBB to download data:

```
symbols = ["NEM", "RGLD", "SSRM", "CDE", "LLY", "UNH",
           "JNJ", "MRK"]
data = obb.equity.price.historical(
    symbols,
    start_date="2015-01-01",
    end_date="2022-12-31",
    provider="yfinance"
)
prices = data[["high", "low", "close", "volume",
               "symbol"]]
```

3. As a preprocessing step, let's make sure all our tickers have at least two years of data. We'll create a mask and grab the stocks that meet our criteria:

```

nobs = prices.groupby("symbol").size()
mask = nobs[nobs > 2 * 12 * 21].index
prices = prices[prices.symbol.isin(mask)]

```

4. Next, set the symbol column as an index, reorder, and drop duplicates:

```

prices = (
    prices
    .set_index("symbol", append=True)
    .reorder_levels(["symbol", "date"])
    .sort_index(level=0)
).drop_duplicates()

```

The result is the following MultiIndex DataFrame with `symbol` as the first index and `date` as the second:

			high	low	close	volume
	symbol	date				
CDE	2015-01-02	5.30	4.96	5.30	2864400	
	2015-01-05	5.45	5.14	5.44	2997000	
	2015-01-06	5.73	5.45	5.69	3885200	
	2015-01-07	5.94	5.47	5.61	4099700	
	2015-01-08	5.85	5.51	5.60	2903300	

Figure 5.10: MultiIndex DataFrame with symbol and date as indexes

5. Next, we'll create a function that returns the normalized Parkinson volatility estimate:

```

def parkinson(data, window=14, trading_days=252):
    rs = (1.0 / (4.0 * np.log(2.0))) * ((data.high / data.low).apply(np.log))**2.0
    def f(v):
        return (trading_days * v.mean())**0.5
    result = rs.rolling(
        window=window,
        center=False
    ).apply(func=f)
    return result.sub(result.mean()).div(result.std())

```

6. We'll apply that function to each group of stocks and add the Parkinson estimator as a new column:

```

prices["vol"] = (
    prices
    .groupby("symbol", group_keys=False)
    .apply(parkinson)
)
prices.dropna(inplace=True)

```

The result is the same MultiIndex DataFrame with each stock's normalized Parkinson volatility added as a new column:

symbol	date	high	low	close	volume	vol
CDE	2015-01-22	6.43	6.15	6.31	3404900	0.502344
	2015-01-23	6.25	5.93	5.99	2234000	0.466274
	2015-01-26	6.13	5.69	6.11	2149800	0.512970
	2015-01-27	6.42	6.12	6.32	3697700	0.508157
	2015-01-28	6.36	5.89	5.99	2668300	0.488355

Figure 5.11: Updated MultiIndex DataFrame with per-stock normalized Parkinson volatility

7. Now that we have the normalized Parkinson volatility, we can compute historic and forward returns. First, compute the historic returns over **1, 5, 10, 21, 42, and 63** periods representing one day through three months:

```
lags = [1, 5, 10, 21, 42, 63]
for lag in lags:
    prices[f"return_{lag}d"] = (
        prices
        .groupby(level="symbol")
        .close
        .pct_change(lag)
    )
```

8. Compute the forward returns for the same periods:

```
for t in [1, 5, 10, 21, 42, 63]:
    prices[f"target_{t}d"] = (
        prices
        .groupby(level="symbol") [f"return_{t}d"]
        .shift(-t)
    )
```

The result is new columns in the prices DataFrame representing the historic and forward returns, as shown in the following screenshot:

date	high	low	close	volume	vol	return_1d	return_5d	return_10d	return_21d	return_42d	return_63d	target_1d	target_5d	target_10d	target_21d	target_42d	target_63d
2015-01-22	24.69	23.98	24.29	13132700	1.021514	NaN	NaN	NaN	NaN	NaN	NaN	-0.005764	-0.004940	0.021820	0.068753	-0.058872	-0.034582
2015-01-23	24.43	23.67	24.15	12056800	0.976772	-0.005764	NaN	NaN	NaN	NaN	NaN	0.014079	0.041408	-0.004141	0.068737	-0.069151	0.034369
2015-01-26	24.63	23.35	24.49	11189200	1.059908	0.014079	NaN	NaN	NaN	NaN	NaN	0.026541	0.025316	0.004900	0.063699	-0.095141	0.044916
2015-01-27	25.25	24.53	25.14	10774500	0.948081	0.026541	NaN	NaN	NaN	NaN	NaN	-0.035004	-0.020684	-0.029037	0.046539	-0.115752	0.048130
2015-01-28	25.05	24.03	24.26	11251300	0.962112	-0.035004	NaN	NaN	NaN	NaN	NaN	-0.003710	0.023083	-0.004946	0.085326	-0.092333	0.091509
2015-01-29	24.34	23.42	24.17	8850000	1.016856	-0.003710	-0.004940	NaN	NaN	NaN	NaN	0.040546	0.026893	0.014067	0.071163	-0.101779	0.095987
2015-01-30	25.22	23.92	25.15	11594400	1.139382	0.040546	0.041408	NaN	NaN	NaN	NaN	-0.001590	-0.043738	-0.014712	0.018688	-0.096620	0.053280
2015-02-02	25.20	24.56	25.11	7497000	1.109351	-0.001590	0.025316	NaN	NaN	NaN	NaN	-0.019514	-0.019912	-0.036639	0.003186	-0.110315	0.037435
2015-02-03	25.07	24.14	24.62	9038100	0.957277	-0.019514	-0.020684	NaN	NaN	NaN	NaN	0.008123	-0.008530	0.004874	0.027620	-0.082859	0.054021
2015-02-04	25.03	24.50	24.82	7071100	0.708258	0.008123	0.023083	NaN	NaN	NaN	NaN	0.000000	-0.027397	-0.014504	-0.061241	-0.106769	0.033441

Figure 5.12: DataFrame with historic and forward returns for each stock in our portfolio

9. Print the first 10 rows for the first symbol to inspect the historic returns for each period:

	high	low	close	volume	vol	return_1d	return_5d	return_10d	return_21d	return_42d	return_63d	target_1d	target_5d	target_10d	target_21d	target_42d	target_63d
date																	
2015-01-22	24.69	23.98	24.29	13132700	1.021514	NaN	NaN	NaN	NaN	NaN	NaN	-0.005764	-0.004940	0.021820	0.068753	-0.058872	-0.034582
2015-01-23	24.43	23.67	24.15	12056800	0.976772	-0.005764	NaN	NaN	NaN	NaN	NaN	0.014079	0.041408	-0.004141	0.068737	-0.069151	0.034369
2015-01-26	24.63	23.35	24.49	11189200	1.059908	0.014079	NaN	NaN	NaN	NaN	NaN	0.026541	0.025316	0.004900	0.063699	-0.095141	0.044916
2015-01-27	25.25	24.53	25.14	10774500	0.948081	0.026541	NaN	NaN	NaN	NaN	NaN	-0.035004	-0.020684	-0.029037	0.046539	-0.115752	0.048130
2015-01-28	25.05	24.03	24.26	11251300	0.962112	-0.035004	NaN	NaN	NaN	NaN	NaN	-0.003710	0.023083	-0.004946	0.085326	-0.092333	0.091509
2015-01-29	24.34	23.42	24.17	8850000	1.016856	-0.003710	-0.004940	NaN	NaN	NaN	NaN	0.040546	0.026893	0.014067	0.071163	-0.101779	0.095987
2015-01-30	25.22	23.92	25.15	11594400	1.139382	0.040546	0.041408	NaN	NaN	NaN	NaN	-0.001590	-0.043738	-0.014712	0.018688	-0.096620	0.053280
2015-02-02	25.20	24.56	25.11	7497000	1.109351	-0.001590	0.025316	NaN	NaN	NaN	NaN	-0.019514	-0.019912	-0.036639	0.003186	-0.110315	0.037435
2015-02-03	25.07	24.14	24.62	9038100	0.957277	-0.019514	-0.020684	NaN	NaN	NaN	NaN	0.008123	-0.008530	0.004874	0.027620	-0.082859	0.054021
2015-02-04	25.03	24.50	24.82	7071100	0.708258	0.008123	0.023083	NaN	NaN	NaN	NaN	0.000000	-0.027397	-0.014504	-0.061241	-0.106769	0.033441

Figure 5.13: Slice of the prices DataFrame showing historic returns for stock NEM

10. Finally, use the Seaborn plotting library to visualize how the one-day forward return is related to the normalized Parkinson volatility factor:

```
target = "target_1d"
metric = "vol"
j = sns.jointplot(x=metric, y=target, data=prices)
plt.tight_layout()
df = prices[[metric, target]].dropna()
r, p = spearmanr(df[metric], df[target])
```

The result is the following joint plot, which shows the relationship along with distributions of the values:

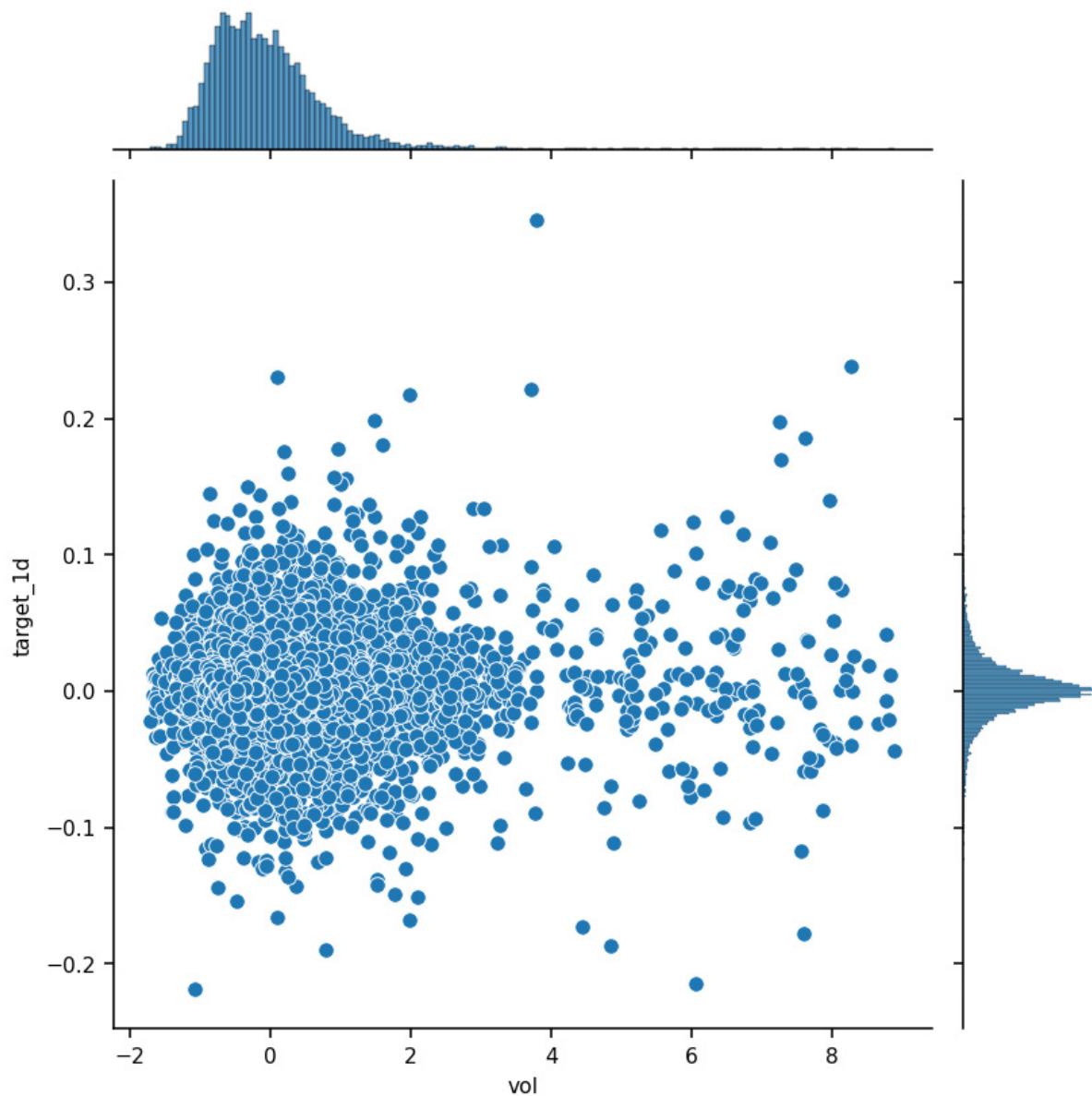


Figure 5.14: Joint plot of normalized Parkinson volatility versus one-day forward return

How it works...

In this recipe, we begin to touch on the data preprocessing required when preparing data for factor analysis.

We modify the DataFrame with the historical data by setting `symbol` as a part of a MultiIndex, reordering the levels of the index to have `symbol` first, and then removing any duplicate rows. This allows us to apply the normalized Parkinson volatility to each group in the next step. To do it, we

group by `symbol` and use the `apply` method to apply our custom Parkinson volatility function to each chunk of data.

Finally, we compute the historic and forward returns. In the first loop, the code computes historical returns over several periods specified in the `lags` list. It does this by grouping the DataFrame by `symbol` and then computing the percentage change over each of the specified lags. These historical returns are then added to the DataFrame as new columns with the names `return_1d`, `return_5d`, and so on.

In the second loop, the code computes future returns for several periods specified in the list. It does this by taking the already computed historical returns (`return_1d`, `return_5d`, and so on) and shifting them up by the specified number of periods. This essentially assigns to each row the return that will occur `t` days in the future. These future returns are then added to the DataFrame as new columns with the names `target_1d`, `target_5d`, and so on.

There's more...

The `spearmanr` function computes the Spearman rank-order correlation coefficient between two data arrays. We can get the rank correlation and *p*-value using the `spearmanr` SciPy method:

```
stat, pvalue = spearmanr(df[metric], df[target])
```

The `stat` coefficient measures the strength and direction of the relationship between the two variables, with a value between -1 and 1. A value of 0 indicates no correlation, 1 indicates a perfect positive correlation, and -1 indicates a perfect negative correlation.

In this context, `stat` is the correlation between Parkinson volatility and the forward returns which is `0.0378`, which is a weak positive correlation.

The `pvalue` tests the null hypothesis that the data is uncorrelated. A small `pvalue` (typically ≤ 0.05) indicates that you can reject the null hypothesis. In this case, the `pvalue` is `0.0158`, so we reject the null hypothesis, which suggests there is a statistically significant correlation between the Parkinson volatility and the returns of this portfolio.

See also

For more resources on the tools and techniques used in this recipe, visit the following links:

- Walk-through of Parkinson volatility: <https://www.ivolatility.com/help/3.html>
- Average true range: https://en.wikipedia.org/wiki/Average_true_range
- Spearman rank correlation: https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

- SciPy's implementation of the Spearman rank correlation coefficient documentation:
<https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html>

Preparing a factor ranking model using Zipline Pipelines

Zipline Reloaded is an open source Python library that provides a comprehensive backtesting framework for algorithmic trading. The library provides tools to manage the algorithm's capital, set trading commissions, and simulate orders.

One of the most powerful features of Zipline Reloaded is the Pipeline API. Pipelines let us define factors from columns of data from bundles. Pipelines efficiently process large amounts of data in a single pass while also filtering down to a smaller set of assets of interest. Pipelines allow us to rank a universe of stocks based on the computed factor and output those data in a format suitable for the Alphalens library, which analyzes factor performance and risk. We'll look at Alphalens in [Chapter 8, Evaluate Factor Risk and Performance with Alphalens Reloaded](#).

In this recipe, we'll build a data bundle using free stock market data, define a momentum factor, and rank a universe of stocks based on the performance in that factor.

Getting ready

The steps to install Zipline Reloaded differ depending on your operating system.

For Windows, Unix/Linux, and Mac Intel users

If you're running on an Intel x86 chip, you can use `conda`:

```
conda install -c conda-forge zipline-reloaded pyfolio-reloaded alphalens-reloaded -y
```

For Apple Silicon users

If you have a Mac with an M-series chip, you need to install some dependencies first. The easiest way is to use Homebrew (<https://brew.sh>).

Install the dependencies with Homebrew:

```
brew install freetype pkg-config gcc openssl hdf5 ta-lib
```

Install the Python dependencies with `conda`:

```
conda install -c conda-forge pytables h5py -y
```

Install the Zipline Reloaded ecosystem:

```
pip install zipline-reloaded pyfolio-reloaded alphalens-reloaded
```

In this example, we'll use the free data bundle provided by Nasdaq Data Link. This dataset has market price data on 3,000 stocks through 2018.

The free dataset is great for getting up and running but it is limited. Once you start hitting the limitations of the free data, you can either pay for a premium dataset or ingest your own data. Make sure you have your Nasdaq Data Link API key handy.

How to do it...

To build the analysis, we'll import the required Zipline Reloaded modules.

1. Import the libraries we need for the analysis:

```
import os
import numpy as np
import pandas as pd
from zipline.data.bundles.core import load
from zipline.pipeline import Pipeline
from zipline.pipeline.data import USEquityPricing
from zipline.pipeline.engine import SimplePipelineEngine
from zipline.pipeline.factors import AverageDollarVolume, CustomFactor, Returns
from zipline.pipeline.loaders import USEquityPricingLoader
```

2. Set your API key as an environment variable and load the free Nasdaq data into a bundle:

```
os.environ["QUANDL_API_KEY"] = "YOUR_API_KEY"
bundle_data = load("quandl", os.environ, None)
```

3. Build a US equity pricing loader:

```
pipeline_loader = USEquityPricingLoader(
    bundle_data.equity_daily_bar_reader,
    bundle_data.adjustment_reader,
    fx_reader=None
)
```

4. Use the pricing loader to create a Pipeline engine:

```
engine = SimplePipelineEngine(
    get_loader=lambda col: pipeline_loader,
    asset_finder=bundle_data.asset_finder
)
```

5. Implement a custom momentum factor that returns a measure of price momentum:

```
class MomentumFactor(CustomFactor):
    inputs = [USEquityPricing.close, Returns(window_length=126)]
    window_length = 252
    def compute(self, today, assets, out, prices, returns):
        out[:] = (
            (prices[-21] - prices[-252]) / prices[-252]
            - (prices[-1] - prices[-21]) / prices[-21]
        ) / np.nanstd(returns, axis=0)
```

6. Create a function that instantiates the custom momentum factor, builds a filter for average dollar volume over the last 30 days, and returns a Pipeline:

```
def make_pipeline():
    momentum = MomentumFactor()
    dollar_volume = AverageDollarVolume(
        window_length=30)
    return Pipeline(
        columns={
            "factor": momentum,
            "longs": momentum.top(50),
            "shorts": momentum.bottom(50),
            "rank": momentum.rank()
        },
        screen=dollar_volume.top(100)
    )
```

7. Run the Pipeline:

```
results = engine.run_pipeline(
    make_pipeline(),
    pd.to_datetime("2012-01-04"),
    pd.to_datetime("2012-03-01")
)
```

8. Clean up the resulting DataFrame by removing records with no factor data, adding names to the MultiIndex and sorting the values first by date and then by factor value:

```
results.dropna(subset="factor", inplace=True)
results.index.names = ["date", "symbol"]
results.sort_values(by=["date", "factor"], inplace=True)
```

The result is a MultiIndex DataFrame including the raw factor value, Boolean values indicating a short or long position, and how the stock's factor value is ranked among the universe:

			factor	longs	shorts	rank
	date	symbol				
2012-01-04	Equity(300 [BAC])	-2.522045	False	False	165.0	
	Equity(1264 [GS])	-2.215784	False	False	220.0	
	Equity(1888 [MS])	-2.204802	False	False	225.0	
	Equity(1894 [MSFT])	-1.949654	False	False	295.0	
	Equity(457 [C])	-1.830819	False	False	345.0	
...
2012-03-01	Equity(3105 [WMT])	3.409414	False	False	2607.0	
	Equity(1690 [LLY])	3.809608	False	False	2642.0	
	Equity(399 [BMY])	4.689588	True	False	2685.0	
	Equity(1770 [MCD])	4.816880	True	False	2691.0	
	Equity(1789 [MDLZ])	5.680276	True	False	2706.0	

Figure 5.15: MultiIndex DataFrame with factor information and trading indicators

How it works...

The first step is to initialize a Pipeline loader for US equities. `usequityPricingLoader` is responsible for loading pricing and adjustment data for US equities from the specified data sources. The `simplePipelineEngine` class is used to run a pipeline. The `get_loader` parameter is a function that returns a loader to be used to load the data needed for the pipeline. In this case, it's a lambda function that always returns the previously defined `pipeline_loader`. The `asset_finder` parameter is used to do asset lookups. Here, it is set to the `asset_finder` of `bundle_data`, which is an object that knows how to look up asset metadata for the bundle.

Our code defines a custom factor, `MomentumFactor`. This factor computes momentum as the percentage change in price over the first 126 days of the 252-day window minus the percentage change in price over the last 21 days, divided by the standard deviation of the returns over the 252-day window. `inputs` is the data that the factor needs: the close prices of the US equities and the returns over a 126-day window. `window_length` is the number of days of data that the `compute` method will receive. The `compute` method is where the actual computation of the factor is done. The `out` array is where the computed factor values are stored. The `[:]` after `out` is used to modify the `out` array in place.

Next, we implement a function called `make_pipeline` that brings the factor, the universe screener, and Pipeline together. The `MomentumFactor` class is instantiated to compute momentum, and the `AverageDollarVolume` function is used with a 30-day window to calculate average dollar volume. The pipeline includes columns for the momentum factor, Boolean masks for selecting the top and bottom 50 securities based on momentum, and the momentum rank across all securities. Additionally, the pipeline employs a screen to focus on the top 100 securities by dollar volume.

There's more...

Zipline Reloaded comes with many built-in factor classes that can be used within trading algorithms. Some of the more commonly used algorithms are as follows:

- `ExponentialWeightedMovingAverage`: Computes the exponential weighted moving average over a specified window length
- `BollingerBands`: Computes the Bollinger Bands over a specified window length
- `VWAP`: Computes the volume-weighted average price over a specified window length
- `AnnualizedVolatility`: Computes the annualized volatility of an asset over a specified window length
- `MaxDrawdown`: Computes the maximum drawdown of an asset over a specified window length

See also

To learn more about Zipline Reloaded, see the documentation here: <https://zipline.ml4trading.io>.

6

Vector-Based Backtesting with VectorBT

Now that we've touched on the fundamental Python tools for algorithmic trading, we'll move to the next phase of the workflow: backtesting. Since most strategies will not consistently make money, and those that do may only make money for a short time, quickly iterating through ideas is critical. This chapter demonstrates how to use vector-based backtesting for the simulation and optimization of trading strategies.

VectorBT is a high-performance, vector-based backtesting framework that allows for efficient evaluation of trading strategies by processing entire time-series data arrays at once, rather than one data point at a time. This method significantly speeds up backtesting operations, making it ideal for rapid strategy iteration. The technique is highly customizable, enabling traders to fine-tune parameters and assess multiple strategies concurrently. We will explore the optimization of these strategies with VectorBT.

In this chapter, we will explore the following recipes:

- Building technical strategies with VectorBT
- Conducting walk-forward optimization with VectorBT
- Optimizing the SuperTrend strategy with VectorBT Pro

Building technical strategies with VectorBT

This recipe introduces you to the powerful vector-based backtesting library VectorBT. One of the most compelling advantages of using VectorBT is its speed in running simulations. Whether you are testing a single strategy or optimizing across a multi-dimensional parameter space, VectorBT's performance is optimized to deliver results in a fraction of the time traditional methods would require.

Built on top of well-established libraries such as pandas, NumPy, and Numba, VectorBT seamlessly integrates into the data science ecosystem. It leverages pandas for its DataFrame structure, which is familiar to most quants. NumPy's numerical computing abilities provide the mathematical backbone, ensuring that heavy calculations are performed efficiently. However, the real game-changer is Numba, a **Just-In-Time (JIT)** compiler that translates Python functions to optimized machine code at runtime. Thanks to Numba, VectorBT can execute loops and mathematical operations at speeds comparable to those of a low-level language, all while allowing you to write in Python.

This recipe introduces VectorBT by building a simple moving average crossover strategy (the “Hello World” of trading strategy development).

Getting ready

There are two versions of VectorBT: a free open source library and a more full-featured Pro version. This recipe uses the free open source version.

You can install it using `pip`:

```
pip install vectorbt
```

How to do it...

VectorBT can be considered an extension of pandas. Let’s see it in action:

1. Import the libraries needed for the analysis:

```
import pandas as pd
import vectorbt as vbt
```

2. Download data using the built-in data downloading class:

```
start = "2016-01-01 UTC"
end = "2020-01-01 UTC"
prices = vbt.YFData.download(
    ["META", "AAPL", "AMZN", "NFLX", "GOOG"],
    start=start,
    end=end
).get("Close")
```

The result is the following pandas DataFrame with the closing prices for the selected symbols:

symbol	META	AAPL	AMZN	NFLX	GOOG
Date					
2016-01-04 05:00:00+00:00	102.220001	24.009066	31.849501	109.959999	37.091999
2016-01-05 05:00:00+00:00	102.730003	23.407415	31.689501	107.660004	37.129002
2016-01-06 05:00:00+00:00	102.970001	22.949339	31.632500	117.680000	37.181000
2016-01-07 05:00:00+00:00	97.919998	21.980770	30.396999	114.559998	36.319500
2016-01-08 05:00:00+00:00	97.330002	22.096996	30.352501	111.389999	35.723499

Figure 6.1: A pandas DataFrame with market data

3. Build the moving average indicators using VectorBT’s built-in MA class:

```
fast_ma = vbt.MA.run(prices, 10, short_name="fast")
slow_ma = vbt.MA.run(prices, 30, short_name="slow")
```

4. Next, we'll find the entry positions. In this example, these occur when the fast-moving average crosses above the slow-moving average:

```
entries = fast_ma.ma_crossed_above(slow_ma)
```

The result is the following pandas DataFrame containing boolean values where trades should be entered:

fast_window	10				
slow_window	30				
symbol	META	AAPL	AMZN	NFLX	GOOG
Date					
2016-01-04 05:00:00+00:00	False	False	False	False	False
2016-01-05 05:00:00+00:00	False	False	False	False	False
2016-01-06 05:00:00+00:00	False	False	False	False	False
2016-01-07 05:00:00+00:00	False	False	False	False	False
2016-01-08 05:00:00+00:00	False	False	False	False	False

Figure 6.2: A pandas DataFrame with locations of entry positions

5. Do the opposite for the exit positions:

```
exits = fast_ma.ma_crossed_below(slow_ma)
```

6. Run the backtest using the entry and exit signals:

```
pf = vbt.Portfolio.from_signals(prices, entries, exits)
```

7. Visualize the mean daily return for each symbol:

```
pf.total_return().groupby(
    "symbol").mean().vbt.barplot()
```

The result is an interactive Plotly bar chart with the mean daily returns for each symbol:

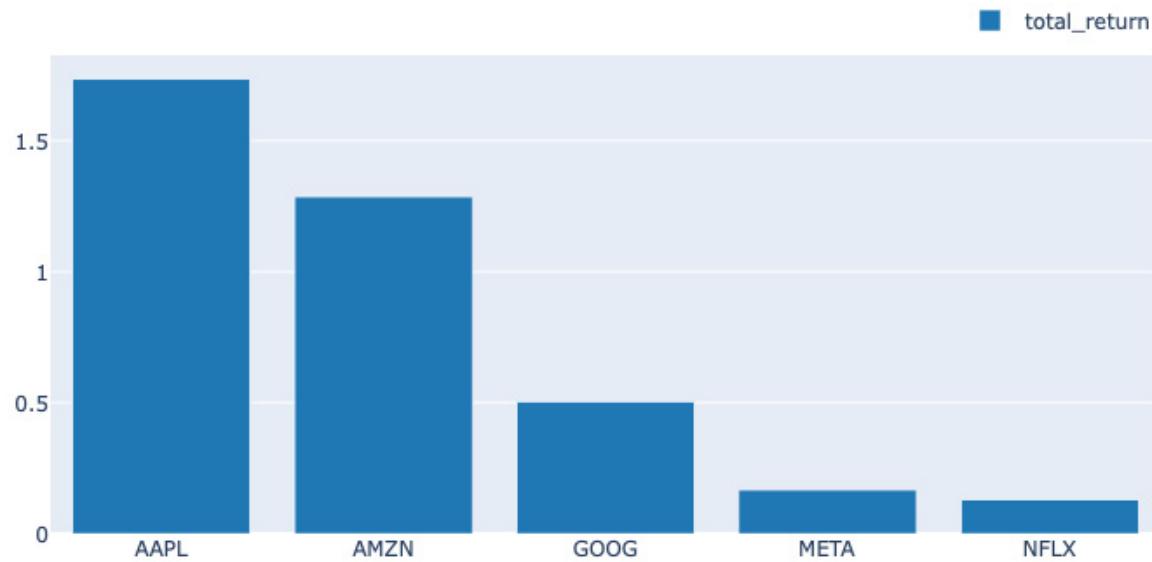


Figure 6.3: Visualizing the mean daily returns of each symbol

8. Inspect the returns for each symbol by simply holding each throughout the analysis period:

```

(
    vbt
    .Portfolio
    .from_holding(
        prices,
        freq='1d'
    )
    .total_return()
    .groupby("symbol")
    .mean()
    .vbt
    .barplot()
)

```

The result is the following bar chart with the mean daily returns for each symbol, assuming that they were simply held throughout the analysis period:

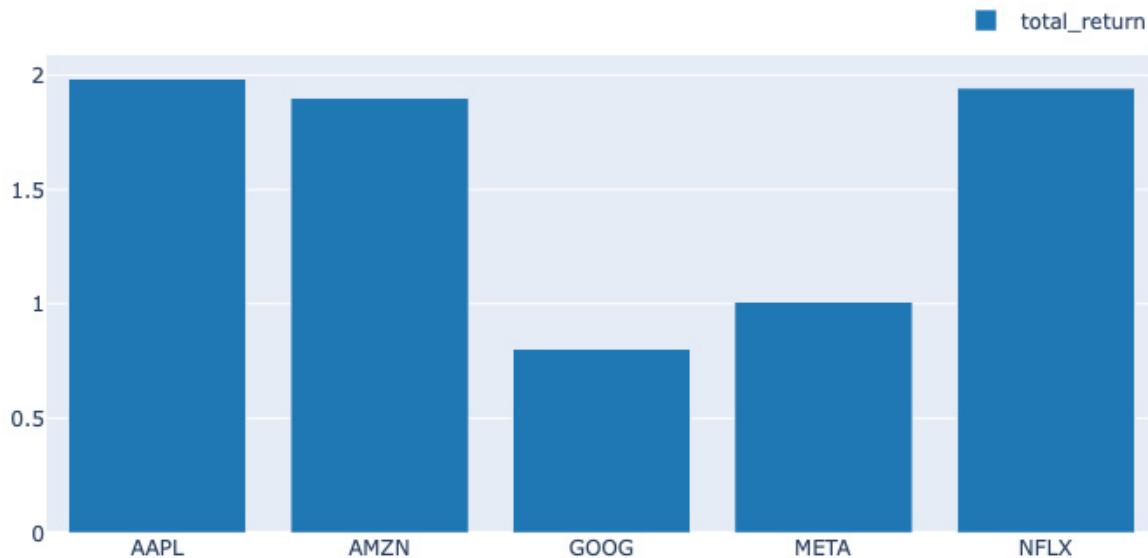


Figure 6.4: Returns from a simple holding strategy

How it works...

The code performs a simple moving average crossover strategy on the FAANG stocks to demonstrate the VectorBT backtesting framework. We download historical closing prices for the META, AAPL, AMZN, NFLX, and GOOG stocks using `yfData.download`. We then use the `get` method and the `close` key to return the closing prices from the `yfData` object.

From there, we calculate two moving averages: a fast moving average with a 10-day window and a slow moving average with a 30-day window. We do this using the `MA` class, which takes the price data, the window size, and a short name for the moving average as arguments. The `ma_crossed_above` method identifies points where the fast moving average crosses above the slow moving average, signaling a buy entry. Conversely, the `ma_crossed_below` method identifies points where the fast moving average crosses below the slow moving average, signaling a sell or exit.

Finally, the `from_signals` method is used to simulate the portfolio's performance based on these entry and exit signals and the historical price data. The resulting portfolio object, `pf`, contains various statistics and data that can be used for further analysis of the strategy's performance. We extract the total returns using the `total_returns` method, group by each symbol, and plot the mean returns.

There's more...

The power in VectorBT is not backtesting simple trading strategies. It's in VectorBT's capacity for running split tests and different iterations of parameters very quickly:

1. Let's split the data into four panels:

```
mult_prices, _ = prices.vbt.range_split(n=4)
```

The result is the following MultiIndex DataFrame, which has historic price data split across four separate split indexes:

symbol	split_idx	0	1												2												3												
			META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG							
0	102.220001	24.009066	31.849501	109.959999	37.071999	116.860001	27.059305	37.683498	127.489998	39.306999	181.419998	40.776524	59.450500	201.070007	53.250000	131.740005	34.163826	75.014000	271.200012	50.803001																			
1	102.730003	23.407412	31.689501	107.660004	37.129002	118.690002	27.029024	37.859001	129.410003	39.345001	184.669998	40.769421	60.209999	205.050003	54.124001	137.949997	35.622265	78.769501	297.570007	53.535500																			
2	102.970001	22.949341	31.632500	117.680000	37.181000	120.669998	27.166475	39.022499	131.809998	39.701000	184.330002	40.958790	60.479500	205.630005	54.320000	138.050003	35.542969	81.475502	315.339996	53.419498																			
3	97.919998	21.980772	30.396999	114.559998	36.319500	123.410004	27.469334	39.799500	131.070007	40.307499	188.850006	41.425125	61.457001	209.990005	55.111500	142.529998	36.220528	82.829002	320.269989	53.813999																			
4	97.330002	22.097000	30.352501	111.389999	35.723499	124.900002	27.720940	39.846001	130.949997	40.332500	188.279999	41.271263	62.343498	212.050003	55.347000	144.229996	36.835613	82.971001	319.959991	53.730002																			
...					
246	117.400002	27.091925	38.317001	125.580002	39.563000	177.199997	41.427494	58.417999	189.940000	53.006001	124.059998	35.278683	67.197998	233.880005	48.811001	205.119995	69.327438	89.460503	333.200012	67.178001																			
247	117.269997	27.145506	38.029499	125.589996	39.495499	175.990005	40.376480	58.838001	187.759995	52.837002	134.179993	37.763054	73.544998	253.669998	51.973000	207.789993	70.702927	93.438499	332.630005	68.019997																			
248	118.100002	27.317902	38.570000	128.350000	39.574799	177.619995	40.383887	59.112999	186.240000	52.468498	134.520004	37.517971	73.082001	255.570007	52.194000	208.100000	70.676102	93.489994	329.089996	67.594498																			
249	116.919998	27.201422	38.606499	125.889999	39.252499	177.919998	40.497204	59.305000	192.710007	52.407001	133.199997	37.537189	73.901001	256.079987	51.854000	204.410004	71.095573	92.344498	323.309998	66.806999																			
250	116.349998	27.194427	38.257500	125.330002	39.139500	176.460007	40.059280	58.473499	191.960007	52.320000	131.089996	37.900017	75.098503	267.660004	51.780499	205.250000	71.615021	92.391998	323.570007	66.850998																			

Figure 6.5: A pandas DataFrame with split indexes

2. Within each split, we can run different combinations of our fast and slow moving average windows:

```
fast_ma = vbt.MA.run(mult_prices, [10, 20],  
short_name="fast")  
slow_ma = vbt.MA.run(mult_prices, [30, 30],  
short_name="slow")
```

3. Rerun the methods to find entries and exits:

```
entries = fast_ma.ma_crossed_above(slow_ma)  
exits = fast_ma.ma_crossed_below(slow_ma)
```

4. Inspect the exits DataFrame. The result is a MultiIndex DataFrame that includes each combination of moving average windows within each split:

symbol	split_idx	0												1												2													
		META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG								
0	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False			
1	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False			
2	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False			
3	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False			
4	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False			
...
246	False	False	False	False	False	False	False	False	False	True	False																												
247	False	False	False	False	False	False	True	False																															
248	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False			
249	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False			
250	True	False	True	False																																			

Figure 6.6: A MultiIndex DataFrame with each combination of moving average windows within each split

5. Rerun the backtest analysis with the split data:

```
pf = vbt.Portfolio.from_signals(  
    mult_prices,  
    entries,  
    exits,  
    freq="1D"  
)
```

6. Visualize the results by grouping total returns by split index and symbol, finding the mean, and plotting:

```
(  
    pf  
        .total_return()  
        .groupby(  
            ['split_idx', 'symbol'])  
    )  
    .mean()  
    .unstack(level=-1)  
    .vbt  
    .barplot()  
)
```

The result is the following bar chart with the mean daily returns for each symbol across each split:



Figure 6.7: A visualization of mean daily returns grouped by split and symbol

7. VectorBT supports trading statistics based on the **orders** property on the **pf** object:

```
pf.orders.stats(group_by=True)
```

The result is a Series with various trading statistics:

```

Start          0
End           250
Period       251 days 00:00:00
Total Records    291
Total Buy Orders   157
Total Sell Orders   134
Min Size        0.253075
Max Size         4.677407
Avg Size         1.629121
Avg Buy Size      1.637868
Avg Sell Size      1.618872
Avg Buy Price     103.523933
Avg Sell Price     104.650348
Total Fees        0.0
Min Fees          0.0
Max Fees          0.0
Avg Fees          0.0
Avg Buy Fees       0.0
Avg Sell Fees       0.0
Name: group, dtype: object

```

Figure 6.8: Trading statistics from the backtest analysis

8. We can also extract specific performance metrics for each combination of split, moving average window, and symbol:

```
pf.sharpe_ratio()
```

The result is the following MultiIndex Series:

fast_window	slow_window	split_idx	symbol	sharpe_ratio
10	30	0	META	-0.472595
			AAPL	1.125228
			AMZN	1.633329
			NFLX	-0.470585
			GOOG	-0.051110
1	30	1	META	0.449941
			AAPL	1.043138
			AMZN	0.611364
			NFLX	-0.359590
			GOOG	1.754603

Figure 6.9: The Sharpe ratio for each combination of split, moving average window, and symbol

See also

The free, open source version of VectorBT has good documentation, examples, and additional resources to help you get started. You can find it here: <https://vectorbt.dev>.

Conducting walk-forward optimization with VectorBT

Walk-forward optimization is an advanced technique in algorithmic trading that aims to address the issue of curve-fitting in strategy development. Unlike traditional backtesting, whereby a strategy is optimized once over a historical dataset and then applied to out-of-sample data, walk-forward optimization divides the entire dataset into multiple in-sample and out-of-sample periods. The strategy is optimized on each in-sample period. The optimized parameters are then validated on the corresponding out-of-sample period. This process is repeated, or “walked forward,” through the entire dataset. The objective is to assess how well the strategy adapts to changing market conditions over time. By continually re-optimizing and validating the strategy, walk-forward optimization provides a more realistic representation of a strategy’s robustness and potential for future performance. This method is computationally intensive but offers a more rigorous approach to strategy validation.

This recipe uses VectorBT’s built-in `rolling_split` method to take advantage of the library’s speed and let us run thousands of simulations in seconds.

Getting ready

This recipe uses the free, open source version of VectorBT that we installed in the last recipe.

How to do it...

We’ll run the backtest analysis and test whether the out-of-sample Sharpe ratios are statistically greater than the in-sample results:

1. Import the libraries for the analysis. We’ll use SciPy to run a statistical test to collect evidence of overfitting:

```
import numpy as np
import scipy.stats as stats
import vectorbt as vbt
```

2. Download market price data using the built-in downloader:

```
start = "2016-01-01 UTC"
end = "2020-01-01 UTC"
prices = vbt.YFData.download(
    "AAPL",
    start=start,
    end=end
).get("Close")
```

3. Create data splits for the walk-forward optimization. This code segments the prices into 30 splits, each two years long, and reserves 180 days for the test:

```
(in_price, in_indexes), (out_price, out_indexes) = prices.vbt.rolling_split(
```

```

        n=30,
        window_len=365 * 2,
        set_lens=(180,),
        left_to_right=False,
    )

```

4. Now create the function that returns the Sharpe ratios for all combinations of moving average windows:

```

def simulate_all_params(price, windows, **kwargs):
    fast_ma, slow_ma = vbt.MA.run_combs(
        price,
        windows,
        r=2,
        short_names=["fast", "slow"]
    )
    entries = fast_ma.ma_crossed_above(slow_ma)
    exits = fast_ma.ma_crossed_below(slow_ma)
    pf = vbt.Portfolio.from_signals(price, entries,
                                    exits, **kwargs)
    return pf.sharpe_ratio()

```

5. Create two helper functions that return the indexes and parameters where the performance is maximized:

```

def get_best_index(performance):
    return performance[
        performance.groupby("split_idx").idxmax()
    ].index
def get_best_params(best_index, level_name):
    return best_index.get_level_values(
        level_name).to_numpy()

```

6. Implement a function that runs the backtest given the best moving average values and returns the associated Sharpe ratio:

```

def simulate_best_params(price, best_fast_windows,
                        best_slow_windows, **kwargs):
    fast_ma = vbt.MA.run(
        price,
        window=best_fast_windows,
        per_column=True
    )
    slow_ma = vbt.MA.run(
        price,
        window=best_slow_windows,
        per_column=True
    )
    entries = fast_ma.ma_crossed_above(slow_ma)
    exits = fast_ma.ma_crossed_below(slow_ma)
    pf = vbt.Portfolio.from_signals(
        price, entries, exits, **kwargs)
    return pf.sharpe_ratio()

```

7. Finally, we will run the analysis by passing in a range of moving average windows to `simulate_all_params`. This returns the Sharpe ratio for every combination of moving average windows for every data split. In other words, these are the in-sample Sharpe ratios:

```

windows = np.arange(10, 40)
in_sharpe = simulate_all_params(
    in_price,
    windows,
    direction="both",
    freq="d"
)

```

8. Next, we will get the best in-sample moving average windows and combine them into a single array:

```
in_best_index = get_best_index(in_sharpe)
in_best_fast_windows = get_best_params(
    in_best_index,
    "fast_window"
)
in_best_slow_windows = get_best_params(
    in_best_index,
    "slow_window"
)
in_best_window_pairs = np.array(
    list(
        zip(
            in_best_fast_windows,
            in_best_slow_windows
        )
    )
)
```

9. The last step is to retrieve the out-of-sample Sharpe ratios using the optimized moving average windows:

```
out_test_sharpe = simulate_best_params(
    out_price,
    in_best_fast_windows,
    in_best_slow_windows,
    direction="both",
    freq="d"
)
```

The result is the following MultiIndex Series that identifies the optimal moving average window values for each split along with the associated Sharpe ratio:

ma_window	ma_window	split_idx	
10	11	0	0.104954
12	13	1	0.318318
		2	0.971219
10	11	3	1.386785
12	13	4	1.303272
10	11	5	2.133298
		6	2.043526

Figure 6.10: Maximized Sharpe ratios across each data split

How it works...

The code executes a walk-forward optimization on a moving average crossover strategy for AAPL stock. We begin by fetching historical closing prices for AAPL for the period from January 1, 2016, to January 1, 2020. The data is then partitioned into in-sample and out-of-sample sets using VectorBT's `rolling_split` method. The in-sample set is designated for optimization, while the out-of-sample set is reserved for validation.

The `rolling_split` method in VectorBT is designed to split a time series into rolling in-sample and out-of-sample periods for walk-forward optimization or other time-based analyses. These are the parameters we use in the recipe:

- `n`: This refers to the number of splits. It determines how many in-sample and out-of-sample periods will be created.
- `window_len`: This describes the length of each rolling window in the time series. This is often specified in terms of the number of time steps (e.g., days).
- `set_lens`: This is a tuple specifying the length of each in-sample and out-of-sample set within each rolling window. The sum of these lengths should not exceed `window_len`.
- `left_to_right`: Determines whether to resolve `set_lens` from left to right. Otherwise, the first set will be variable.

In the optimization phase, the `simulate_all_params` function is responsible for strategy backtesting across a range of moving average window sizes. It calculates both fast and slow moving averages, generates entry and exit signals, and simulates the portfolio's performance, returning the Sharpe ratio as the performance metric.

TIP

You can select any performance metric to optimize for. The `pF` portfolio object has dozens of different metrics to choose from. You can execute `dir(pF)` to inspect the properties and methods of the object.

Next, we use the `get_best_index` and `get_best_params` functions to identify the optimal moving average window sizes based on the highest Sharpe ratios achieved during the in-sample testing.

Finally, the code proceeds to the out-of-sample testing phase. The `simulate_best_params` function takes the optimal moving average window sizes identified in the in-sample phase and applies them to the out-of-sample dataset. It then simulates the portfolio's performance using these parameters, again computing the Sharpe ratio as the performance metric.

The code is structured to leverage VectorBT's efficient backtesting capabilities, which is particularly advantageous in a walk-forward optimization context where multiple iterations of backtesting and re-optimization are required.

There's more...

It's common to overfit backtesting models to market noise. This is especially acute when brute force optimizing technical analysis strategies. To collect evidence to this effect, we can use a one-sided independent t-test to assess the statistical significance between the means of Sharpe ratios for in-sample and out-of-sample datasets:

```
in_sample_best = in_sharpe[in_best_index].values
out_sample_test = out_test_sharpe.values
```

```
t, p = stats.ttest_ind(  
    a=out_sample_test,  
    b=in_sample_best,  
    alternative="greater"  
)
```

First, the line `in_sample_best = in_sharpe[in_best_index].values` filters the Sharpe ratios stored in `in_sharpe` to include only those corresponding to the best-performing parameter sets. It then extracts these filtered Sharpe ratios as a NumPy array and stores them in the `in_sample_best` variable. We do the same for the out-of-sample dataset.

The `ttest_ind` function from the SciPy stats module takes the two independent `out_sample_test` and `in_sample_best` samples as its arguments. The `alternative="greater"` parameter specifies that the test is one-sided, which we use to evaluate whether the mean Sharpe ratio of the out-of-sample set is statistically greater than that of the in-sample set. The function returns the calculated t-statistic and the p-value.

The results give us a t-statistic of approximately `-1.085` and a p-value of approximately `0.859`. The negative value of the t-statistic suggests that the mean of the out-of-sample Sharpe ratios is negative. Further, the high p-value tells us there is not enough statistical evidence to conclude that the out-of-sample Sharpe ratios are greater than the in-sample Sharpe ratios. The negative t-statistic and the high p-value together suggest that the strategy may not perform as well on new, unseen data as it does on the data on which it was optimized. This could be a warning sign regarding the strategy's robustness and its ability to generalize to new data. Ideally, you'd hope to see a t-statistic over `1.0` and a p-value under `0.05`.

See also

Using VectorBT to quickly iterate on optimized strategies, coupled with SciPy to test the statistical significance of those strategies, is a powerful workflow. You can learn more about SciPy's stats module here: <https://docs.scipy.org/doc/scipy/reference/stats.html#module-scipy.stats>.

Optimizing the SuperTrend strategy with VectorBT Pro

The SuperTrend indicator is a trend-following indicator that is used in technical analysis to identify the direction of an asset's momentum. It is constructed using two primary components: the **Average True Range (ATR)** and a multiplier. The ATR measures the asset's volatility over a specified period, while the multiplier is a user-defined constant that adjusts the sensitivity of the indicator.

The SuperTrend is calculated as follows:

1. Compute the ATR for a given period.

2. Calculate the **basic upper band** as the sum of the high and low prices, divided by 2, plus the product of the multiplier and the ATR.
3. Calculate the **basic lower band** as the sum of the high and low prices, divided by 2, minus the product of the multiplier and the ATR.
4. The SuperTrend is then defined as the upper band when the price is below it, and as the lower band when the price is above it.

The indicator flips between the upper and lower bands, signaling a change in trend. When the price is above the SuperTrend line, it suggests an uptrend and a buy signal is generated. Conversely, when the price is below the SuperTrend line, it suggests a downtrend and a sell signal is generated.

To build the SuperTrend indicator, we will introduce VectorBT Pro. VectorBT Pro is a more full-featured version of VectorBT that offers enhancements such as pulling data from Nasdaq Data Link, AlphaVantage, and Polygon, as well as synthetic data generators, multi-threading, stop laddering, time stops, order delays, portfolio optimization with RiskFolio-Lib and PyPortfolioOpt, and more.

This recipe will demonstrate how to construct a custom indicator using integrations with TA-Lib and Numba.

Getting ready

We need to install a few dependencies to get VectorBT working. First is TA-Lib which is a technical analysis library. The second is H5DF which is a data storage solution.

For Windows, Unix/Linux, and Mac Intel users

If you're running on an Intel x86 chip, you can use `conda`:

```
conda install -c conda-forge pytables h5py ta-lib -y
```

For Apple Silicon users

If you have a Mac with an M-series chip, you need to install some dependencies first. The easiest way is to use Homebrew (<https://brew.sh>).

Install the dependencies with Homebrew:

```
brew install hdf5 ta-lib
```

Install the Python dependencies with `conda`:

```
conda install -c conda-forge pytables h5py ta-lib -y
```

Now we're ready to install VectorBT Pro.

VectorBT Pro has a small monthly subscription fee. Details can be found at <https://vectorbt.pro/become-a-member/>. After you've been added to the list of collaborators and have accepted the repository invitation, the next step is to create a Personal Access Token for your GitHub account to access the Pro repository. To do so, follow these steps:

1. Go to <https://github.com/settings/tokens>.
2. Click on **Generate a new token**.
3. Enter a name (such as **terminal**).
4. Set the expiration to a fixed number of days.
5. Select the **repo** scope.
6. Generate the token and save it in a safe place.

Once your token has been created, you can install Pro using **pip**:

```
pip install -U "vectorbtpro[base] @ git+https://github.com/polakowo/vectorbt.pro.git"
```

When you're prompted for a password, use the token that you generated in the previous steps. For more installation details, see the *Getting Started* guide on the Pro website, which is accessible after signing up.

Next, we'll need TA-Lib. TA-Lib is a technical analysis library. Because it's written in C++ with Python wrappers, we need a little special handling to get it installed.

How to do it...

We'll examine a few key features of VectorBT Pro, including the multi-threaded data downloading and indicator factory:

1. Import the libraries we need for the analysis. Note the import of **vectorbtpro** instead of **vectorbt**:

```
import talib
import numpy as np
from numba import njit
import vectorbtpro as vbt
```

2. Use VectorBT Pro's multi-threading capability to download the market data in 517 milliseconds:

```
start = "2016-01-01 UTC"
end = "2020-01-01 UTC"
with vbt.Timer() as timer:
    prices = vbt.YFData.pull(
        ["META", "AAPL", "AMZN", "NFLX", "GOOG"],
        start=start,
        end=end,
        execute_kwarg=dict(engine="threadpool")
    )
print(timer.elapsed())
```

3. Extract the high, low, and closing prices from the **prices** object:

```
high = prices.get("High")
low = prices.get("Low")
close = prices.get("Close")
```

4. Implement a helper function that calculates the basic bands:

```
def get_basic_bands(med_price, atr, multiplier):
    matr = multiplier * atr
    upper = med_price + matr
    lower = med_price - matr
    return upper, lower
```

5. Implement the function that calculates the final bands. This function returns the trend and direction, as well as both the long and short positions. Note the `@njit` decorator, which compiles the code using Numba to achieve machine language-like speeds:

```
@njit
def get_final_bands(close, upper, lower):
    trend = np.full(close.shape, np.nan)
    dir_ = np.full(close.shape, 1)
    long = np.full(close.shape, np.nan)
    short = np.full(close.shape, np.nan)
    for i in range(1, close.shape[0]):
        if close[i] > upper[i - 1]:
            dir_[i] = 1
        elif close[i] < lower[i - 1]:
            dir_[i] = -1
        else:
            dir_[i] = dir_[i - 1]
            if dir_[i] > 0 and lower[i] < lower[i - 1]:
                lower[i] = lower[i - 1]
            if dir_[i] < 0 and upper[i] > upper[i - 1]:
                upper[i] = upper[i - 1]
        if dir_[i] > 0:
            trend[i] = long[i] = lower[i]
        else:
            trend[i] = short[i] = upper[i]
    return trend, dir_, long, short
```

6. Put it all together in the final function, which returns the output from the `get_final_bands` function. Note the use of TA-Lib's **MEDPRICE** and **ATR** methods, which further speed up the analysis:

```
def supertrend(high, low, close, period=7,
               multiplier=3):
    avg_price = talib.MEDPRICE(high, low)
    atr = talib.ATR(high, low, close, period)
    upper, lower = get_basic_bands(avg_price, atr,
                                   multiplier)
    return get_final_bands(close, upper, lower)
```

7. Use VectorBT Pro's indicator factory class to convert our **supertrend** function into an indicator that we can use with Pro's analysis capabilities:

```
SuperTrend = vbt.IF(
    class_name="SuperTrend",
    short_name="st",
    input_names=["high", "low", "close"],
    param_names=["period", "multiplier"],
    output_names=["supert", "superd", "superl",
                 "supers"],
```

```

).with_apply_func(
    supertrend,
    takes_1d=True,
    period=7,
    multiplier=3,
)

```

8. We will then create a custom class that inherits the indicator that we just built using VectorBT Pro's indicator factory. This class includes a method plot that encapsulates the formatting and setup of a plot:

```

class SuperTrend(SuperTrend):
    def plot(
        self,
        column=None,
        close_kwargs=None,
        superl_kwargs=None,
        supers_kwargs=None,
        fig=None,
        **layout_kwargs
    ):
        close_kwargs = close_kwargs if close_kwargs else {}
        superl_kwargs = superl_kwargs if superl_kwargs else {}
        supers_kwargs = supers_kwargs if supers_kwargs else {}
        close = self.select_col_from_obj(self.close,
            column).rename("Close")
        supers = self.select_col_from_obj(self.supers,
            column).rename("Short")
        superl = self.select_col_from_obj(self.superl,
            column).rename("Long")
        fig = close.vbt.plot(fig=fig, **close_kwargs,
            **layout_kwargs)
        supers.vbt.plot(fig=fig, **supers_kwargs)
        superl.vbt.plot(fig=fig, **superl_kwargs)
        return fig

```

9. We're now ready to create an instance of our SuperTrend indicator and visualize where the indicator suggests long and short positions:

```

st = SuperTrend.run(high, low, close)
st.loc["2016-01-01":"2020-01-01",
       "AAPL"].plot().show()

```

The result is a Plotly chart with the closing price for AAPL and the locations of long and short signals:



Figure 6.11: A chart with the long and short signals for AAPL

10. The next step is to run the backtest. First, let's find the entry and exit signals:

```
entries = (~st.superl.isnull()).vbt.signals.fshift()
exits = (~st.supers.isnull()).vbt.signals.fshift()
```

The result is two DataFrames with boolean values for each day indicating where a long or short position exists:

	symbol	META	AAPL	AMZN	NFLX	GOOG
	Date					
2016-01-04 00:00:00-05:00		False	False	False	False	False
2016-01-05 00:00:00-05:00		False	False	False	False	False
2016-01-06 00:00:00-05:00		False	False	False	False	False
2016-01-07 00:00:00-05:00		False	False	False	False	False
2016-01-08 00:00:00-05:00		False	False	False	False	False

Figure 6.12: A DataFrame with entry locations

11. Finally, we can run the backtest and inspect the risk and performance results:

```
pf = vbt.Portfolio.from_signals(
    close=close,
    entries=entries,
    exits=exits,
    fees=0.001,
    freq="1d")
```

```

)
pf.stats(group_by=True)

```

The result is a Series with the consolidated statistics of the strategy:

Start	2016-01-04 00:00:00-05:00
End	2019-12-31 00:00:00-05:00
Period	1006 days 00:00:00
Start Value	500.0
Min Value	458.811175
Max Value	744.719854
End Value	739.035201
Total Return [%]	47.80704
Benchmark Return [%]	152.73135
Total Time Exposure [%]	86.481113
Max Gross Exposure [%]	100.0
Max Drawdown [%]	11.348404
Max Drawdown Duration	184 days 00:00:00
Total Orders	161
Total Fees Paid	18.243214
Total Trades	83
Win Rate [%]	56.410256
Best Trade [%]	27.086949
Worst Trade [%]	-22.352306
Avg Winning Trade [%]	7.681076
Avg Losing Trade [%]	-5.1919
Avg Winning Trade Duration	50 days 13:38:10.909090909
Avg Losing Trade Duration	16 days 22:35:17.647058823
Profit Factor	1.843733
Expectancy	2.132568
Sharpe Ratio	0.967803
Calmar Ratio	0.905523
Omega Ratio	1.177398
Sortino Ratio	1.3654
Name: group, dtype: object	

Figure 6.13: A sample of the Series with the risk and performance statistics for the SuperTrend strategy

How it works...

The code fetches historical financial data on META, AAPL, AMZN, NFLX, and GOOG for the date range from January 1, 2016, to January 1, 2020. It uses the `pull` method from the `YFData` class to download this data. The `execute_kwarg=dict(engine="threadpool")` argument specifies that a thread pool should be used for concurrent execution, which speeds up the data retrieval process. The last data pre-processing step is to simply extract the high, low, and closing prices from the `YFData` object.

The `get_basic_bands` function calculates the basic upper and lower bands used in the SuperTrend indicator. It takes three arguments: `med_price`, which is the median price of the asset, `atr`, the ATR, and `multiplier`, a user-defined constant to adjust sensitivity. The function returns both the upper and lower bands.

Next, we implement the heart of the SuperTrend indicator. The `get_final_bands` function is JIT compiled, which is optimized using Numba's `@njit` decorator for better performance. It calculates the final upper and lower bands, as well as the trend direction for our SuperTrend indicator. It takes three arrays as input: `close` for closing prices, `upper` for the basic upper band, and `lower` for the basic lower band. The function returns four arrays: `trend` (final band based on trend direction), `dir_` (trend direction), `long` (lower band during uptrends), and `short` (upper band during downtrends).

`supertrend` calculates the SuperTrend indicator using high, low, and close prices. It takes five arguments: `high`, `low`, `close` for the high, low, and closing prices, respectively; `period` for the number of periods to consider for the ATR; and `multiplier` to adjust the sensitivity of the bands. The `supertrend` function returns the final upper and lower bands, the trend direction, and the bands used during uptrends and downtrends, as calculated by `get_final_bands`.

Next, we define a custom indicator class, `SuperTrend`, using VectorBT Pro's `IndicatorFactory` class. This class encapsulates the SuperTrend indicator logic for easier reuse and integration within the Pro ecosystem. The `with_apply_func` method attaches the previously defined `supertrend` function to this custom indicator class. This function will be called when the `SuperTrend` indicator is run.

Finally, we run the backtest. We first run the custom `SuperTrend` indicator on high, low, and close price data, generating SuperTrend values, trend directions, and bands for both long and short positions. Finally, a portfolio is constructed using the entry and exit signals with VectorBT Pro's `from_signals` method. The portfolio uses the close prices for trade execution, incorporates a trading fee of 0.1%, and assumes a daily trading frequency.

There's more...

Since we encapsulated the SuperTrend indicator logic using Pro's indicator factory, we can optimize it. The two parameters in question are `period` and `multiplier`:

1. First, let's create ranges of values for each of the parameters:

```
periods = np.arange(4, 20)
multipliers = np.arange(20, 41) / 10
```

2. Then we call the `run` method on the SuperTrend indicator, passing in the market prices, as well as our arrays of periods and multipliers:

```

        st = SuperTrend.run(
            high, low, close,
            period=periods,
            multiplier=multipliers,
            param_product=True,
        )
    )

```

3. Pro runs through every combination of period and multiplier. We then run through the same code as before that identifies the entries, exits, and runs the backtest. The difference is that now, our DataFrames containing entry and exit positions have a MultiIndex index for each combination of period and multiplier:

st_period	4					... 19					
st_multiplier	2.0					... 4.0					
symbol	META	AAPL	AMZN	NFLX	GOOG	...	META	AAPL	AMZN	NFLX	GOOG
Date											
2016-01-04 00:00:00-05:00	False	False	False	False	False	...	False	False	False	False	False
2016-01-05 00:00:00-05:00	False	False	False	False	False	...	False	False	False	False	False
2016-01-06 00:00:00-05:00	False	False	False	False	False	...	False	False	False	False	False
2016-01-07 00:00:00-05:00	False	False	False	False	False	...	False	False	False	False	False
2016-01-08 00:00:00-05:00	False	False	False	False	False	...	False	False	False	False	False

Figure 6.14: A DataFrame with entry locations for each stock for each combination of period and multiplier

4. Once we have created the entries and exits, we run the backtest:

```

pf = vbt.Portfolio.from_signals(
    close=close,
    entries=entries,
    exits=exits,
    fees=0.001,
    freq="1d"
)

```

5. To visualize the parameter hotspots (the combinations with the maximum Sharpe ratio), use a heatmap:

```

pf.sharpe_ratio.vbt.heatmap(
    x_level="st_period",
    y_level="st_multiplier",
    slider_level="symbol"
)

```

The result is an interactive heatmap that lets us select which asset to visualize. In this case, we select AAPL:

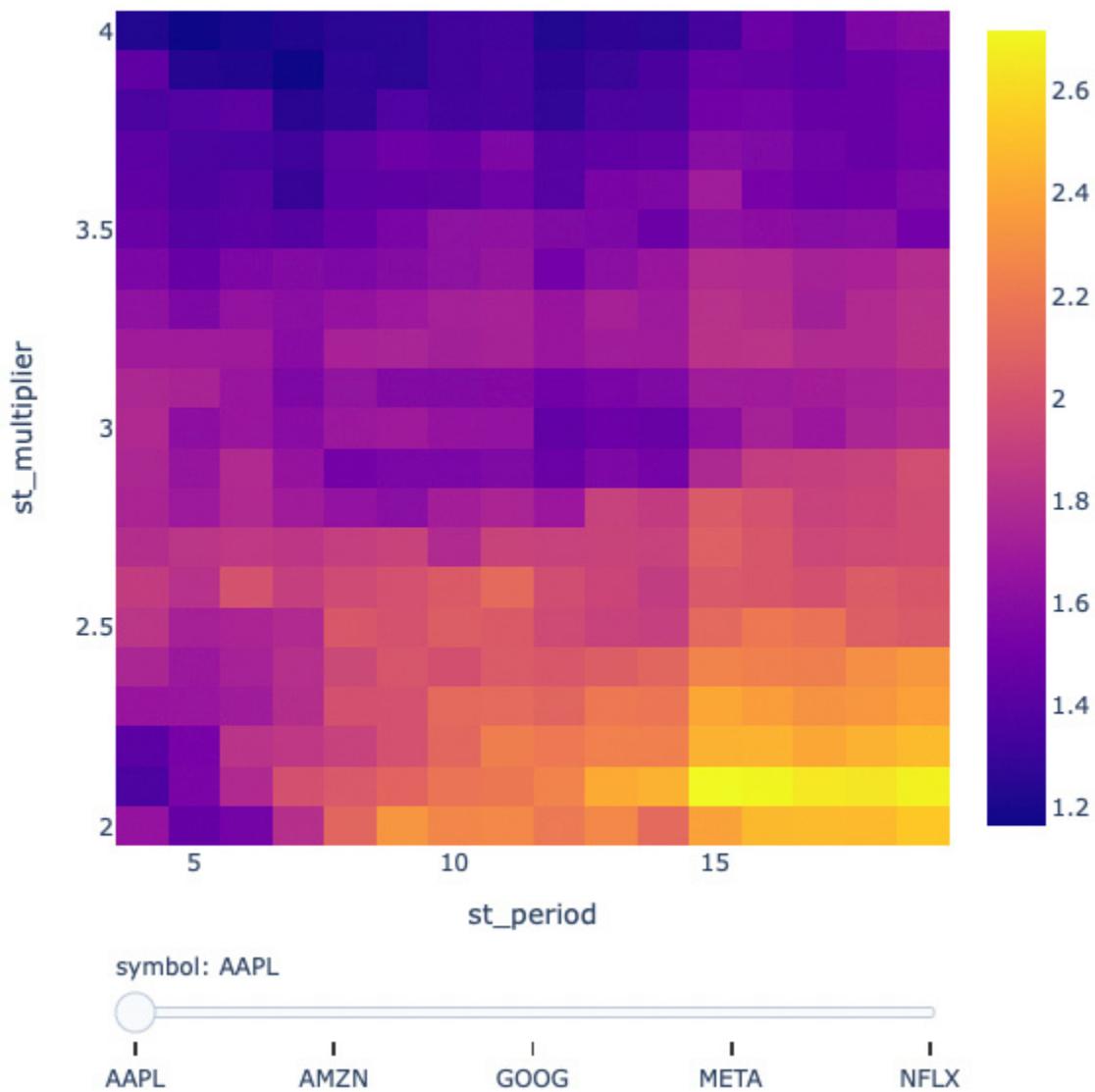


Figure 6.15: An interactive heatmap showing the Sharpe ratio at each parameter combination

TIP

The `pF` object has many risk and performance metrics available for plotting. To get an idea of what's available, print `dir(pF)` to see the object attributes. You can easily plot a different metric on a heatmap by replacing `sharpe_ratio` with whichever metric you're interested in.

See also

We've only begun to scratch the surface of VectorBT and VectorBT Pro. It's designed for speed, which is perfect for optimizing features of our trading strategies.

To learn more about the SuperTrend indicator, consult the VectorBT Pro information page at
<https://vectorbt.pro/become-a-member/>.

Event-Based Backtesting Factor Portfolios with Zipline Reloaded

Zipline Reloaded is an event-driven backtesting framework that processes market events sequentially, allowing for more realistic modeling of order execution and slippage. Unlike vector-based frameworks, it accounts for the temporal sequence of market events, making it suitable for complex strategies that involve conditional orders or asset interactions. While generally slower than vector-based approaches, event-based backtesting frameworks tend to better simulate market dynamics making them helpful for path-dependent strategies requiring intricate order logic, state management, and risk management.

Zipline Reloaded is well suited for backtesting large universes and complex portfolio construction techniques. The Pipeline API is designed for high-efficiency computation of factors among thousands of securities. We'll use Zipline Reloaded to backtest portfolio factor strategies, the results of which can be analyzed with other tools in the Zipline Reloaded ecosystem..

In this chapter, we present the following recipes:

- Backtesting a momentum factor strategy with Zipline Reloaded
- Exploring a mean reversion strategy with Zipline Reloaded

Technical Requirements

We installed Zipline Reloaded in [Chapter 5, Build Alpha Factors for Stock Portfolios](#). In case you missed it, follow along with the instructions here. The steps to install Zipline Reloaded differ depending on your operating system.

For Windows, Unix/Linux, and Mac Intel users

If you're running on an Intel x86 chip, you can use `conda`:

```
conda install -c conda-forge zipline-reloaded pyfolio-reloaded alphalens-reloaded -y
```

For Mac M1/M2 users

If you have a Mac with an M1 or M2 chip, you need to install some dependencies first. The easiest way is to use Homebrew (<https://brew.sh>).

Install the dependencies with Homebrew:

```
brew install freetype pkg-config gcc openssl hdf5 ta-lib
```

Install the Python dependencies with conda:

```
conda install -c conda-forge pytables h5py -y
```

Install the Zipline Reloaded ecosystem:

```
pip install zipline-reloaded pyfolio-reloaded alphalens-reloaded
```

In this example, we'll use the free data bundle provided by Nasdaq Data Link. This dataset has market price data on 3,000 stocks through 2018.

Backtesting a momentum factor strategy with Zipline Reloaded

Before we begin, it's important to understand the difference between vector-based backtesting frameworks and event-based backtesting frameworks. In [Chapter 6, Vector-Based Backtesting with VectorBT](#), we touched on the differences between vector-based and event-based backtesting. Here is a more detailed assessment:

Feature	Vector-Based	Event-Based
Processing Method	Vectorized operations	Sequentially
Computational Efficiency	Highly efficient, especially for large datasets	Less efficient due to the need to process each event individually
Complexity in Coding	Simpler and more concise	More detailed programming to simulate market events accurately
Suitability for Strategy Type	Without complex state or path dependencies.	Ideal for complex, state-dependent, and path-dependent strategies
Order Execution Modeling	Assumes immediate order execution; less accurate for modeling real market conditions like slippage and delays.	More accurately models order execution dynamics, including delays, slippage, and partial fills.
Risk Management Simulation	Basic risk management features; may not capture dynamic risk adjustments effectively.	Detailed risk management with realistic simulations of stop-losses and other real-time adjustments.
Scalability	Highly scalable for large-scale data analysis and multiple assets.	Challenging with large-scale tick-by-tick data due to computational demands.
Risk of Overfitting	Higher risk of overfitting	Risk exists but can be better managed
Development Cycle	Faster development and testing cycles due to computational efficiency and simpler code.	Potentially slower due to increased complexity and computational demands.
Realism	Less realistic as it does not simulate the sequential flow of market events.	Higher realism in simulating real-world market conditions and trader actions.

Figure 8.1: Comparison of vector-based and event-based backtesting frameworks

In [Chapter 5](#), *Build Alpha Factors for Stock Portfolios*, we defined a custom factor that computes a momentum score for US equities over a 252-day window. The factor's value is determined by comparing the 252-day relative price change against the 22-day relative price change. This difference is then standardized by dividing it by the standard deviation of the 126-day returns, effectively scaling the momentum score by the asset's recent volatility. The result provides a normalized momentum score for each asset, capturing both long-term and short-term price movements in relation to its recent volatility.

In this recipe, we'll incorporate the factor into the Zipline Reloaded backtesting framework and inspect the performance of the strategy.

Getting ready

We assume you ingested the free `quandl` data bundle and it's still available on your local machine. We also assume you still have your Nasdaq API key in the environment variables. If not, run the following code after the imports:

```
from zipline.data.bundles.core import load
os.environ["QUANDL_API_KEY"] = "YOUR_API_KEY"
bundle_data = load("quandl", os.environ, None)
```

The free data is limited to about 3,000 US equities and the data collection stops in 2018. If you'd like data coverage for nearly 20,000 US equities updated daily, you can consider the premium data service offered by Nasdaq.

The instructions to set up this premium data service are outlined in detail in the article *How to ingest premium market data with Zipline Reloaded*, which you can find at this URL:

<https://www.pyquantnews.com/free-python-resources/how-to-ingest-premium-market-data-with-zipline-reloaded>.

If you use this premium data, replace `quandl` with `quotemedia` in the `run_algorithm` function that follows.

How to do it...

In this recipe, we introduce several novel features of the Zipline Reloaded backtesting framework that require their own imports — namely, the date and time rules, the Pipeline API, custom factors, and commission and slippage models:

1. Start by importing the libraries we'll need for the backtest:

```
import pandas as pd
```

```

import numpy as np
from zipline import run_algorithm
from zipline.pipeline import Pipeline
from zipline.pipeline.data import USEquityPricing
from zipline.pipeline.factors import AverageDollarVolume, CustomFactor, Returns
from zipline.api import (
    attach_pipeline,
    calendars,
    pipeline_output,
    date_rules,
    time_rules,
    set_commission,
    set_slippage,
    record,
    order_target_percent,
    get_open_orders,
    get_datetime,
    schedule_function
)
import pandas_datareader as web

```

2. Next, define the number of long and short stocks we want in our portfolio:

```
N_LONGS = N_SHORTS = 50
```

3. Use the same custom momentum factor we defined in the previous chapter:

```

class MomentumFactor(CustomFactor):
    inputs = [USEquityPricing.close,
              Returns(window_length=126)]
    window_length = 252
    def compute(self, today, assets, out, prices, returns):
        out[:] =
            (prices[-21] - prices[-252]) / prices[-252]
            - (prices[-1] - prices[-21]) / prices[-21]
        ) / np.nanstd(returns, axis=0)

```

4. Use the same pipeline we defined in the previous chapter:

```

def make_pipeline():
    momentum = MomentumFactor()
    dollar_volume = AverageDollarVolume(
        window_length=30)
    return Pipeline(
        columns={
            "factor": momentum,
            "longs": momentum.top(N_LONGS),
            "shorts": momentum.bottom(N_SHORTS),
            "ranking": momentum.rank(),
        },
        screen=dollar_volume.top(100),
    )

```

5. Zipline Reloaded is an event-driven backtesting framework that allows us to “hook” into different events, including an event that fires before trading starts. We use this hook to “install” our factor pipeline:

```

def before_trading_start(context, data):
    context.factor_data = pipeline_output(
        "factor_pipeline")

```

6. Next, we define the **initialize** function, which is run when the backtest starts:

```

def initialize(context):
    attach_pipeline(make_pipeline(),
                  "factor_pipeline")
    schedule_function(
        rebalance,
        date_rules.week_start(),
        time_rules.market_open(),
        calendar=calendars.US_EQUITIES,
    )

```

7. Now, define a function that contains the logic to rebalance our portfolio. Here we buy the top **N_LONGS** stocks with the highest ranking factor and short the bottom **N_SHORTS** stocks with the lowest ranking factor:

```

def rebalance(context, data):
    factor_data = context.factor_data
    record(factor_data=factor_data.ranking)
    assets = factor_data.index
    record(prices=data.current(assets, "price"))
    longs = assets[factor_data.longs]
    shorts = assets[factor_data.shorts]
    divest = set(
        context.portfolio.positions.keys() - set(
            longs.union(shorts)))
    exec_trades(
        data,
        assets=divest,
        target_percent=0
    )
    exec_trades(
        data,
        assets=longs,
        target_percent=1 / N_LONGS
    )
    exec_trades(
        data,
        assets=shorts,
        target_percent=-1 / N_SHORTS
    )

```

8. We abstract away the order execution in an **exec_trades** function, which loops through the provided assets and executes the orders:

```

def exec_trades(data, assets, target_percent):
    for asset in assets:
        if data.can_trade(
            asset) and not get_open_orders(asset):
            order_target_percent(asset,
                                 target_percent)

```

9. Finally, we run the backtest using the **run_algorithm** function:

```

start = pd.Timestamp("2016")
end = pd.Timestamp("2018")
perf = run_algorithm(
    start=start,
    end=end,
    initialize=initialize,
    before_trading_start=before_trading_start,
    capital_base=100_000,
    bundle="quandl",
)

```

The output is a DataFrame that contains trading, risk, and performance statistics for each day in the backtest:

```
perf.info()
```

It's this DataFrame that we use to analyze the characteristics of the backtest:

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 503 entries, 2016-01-04 21:00:00+00:00 to 2017-12-29 21:00:00+00:00
Data columns (total 39 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   period_open      503 non-null    datetime64[ns, UTC]
 1   period_close     503 non-null    datetime64[ns, UTC]
 2   starting_value   503 non-null    float64 
 3   ending_value     503 non-null    float64 
 4   starting_cash    503 non-null    float64 
 5   ending_cash      503 non-null    float64 
 6   returns          503 non-null    float64 
 7   portfolio_value  503 non-null    float64 
 8   longs_count      503 non-null    int64   
 9   shorts_count     503 non-null    int64   
 10  long_value       503 non-null    float64 
 11  short_value      503 non-null    float64 
 12  long_exposure   503 non-null    float64 
 13  pnl              503 non-null    float64 
 14  short_exposure  503 non-null    float64 
 15  capital_used    503 non-null    float64 
 16  orders           503 non-null    object  
 17  transactions     503 non-null    object  
 18  gross_leverage   503 non-null    float64 
 19  positions        503 non-null    object  
 20  net_leverage     503 non-null    float64 
 21  starting_exposure 503 non-null    float64 
 22  ending_exposure  503 non-null    float64 
 23  factor_data      503 non-null    object  
 24  prices           503 non-null    object  
 25  sharpe           502 non-null    float64 
 26  sortino          502 non-null    float64 
 27  max_drawdown     503 non-null    float64 
 28  max_leverage     503 non-null    float64 
 29  excess_return    503 non-null    float64 
 30  treasury_period_return 503 non-null    float64 
 31  trading_days     503 non-null    int64   
 32  period_label     503 non-null    object  
 33  algorithm_period_return 503 non-null    float64 
 34  algo_volatility  502 non-null    float64 
 35  benchmark_period_return 503 non-null    float64 
 36  benchmark_volatility 502 non-null    float64 
 37  alpha             0 non-null     object  
 38  beta              0 non-null     object  
dtypes: datetime64[ns, UTC](2), float64(26), int64(3), object(8)
memory usage: 157.2+ KB
```

Figure 8.2: Output DataFrame from our Zipline Reloaded backtest

How it works...

We dove into the inner workings of the Pipeline API and custom factors in [Chapter 5, Build Alpha Factors for Stock Portfolios](#), so here, we'll focus on the remaining code to run the backtest. The `before_trading_start` function is invoked prior to the beginning of each trading day. It takes two parameters:

- `context`: A persistent namespace that allows the storage of variables between function calls. It retains its values across multiple days and can be used to store and manage data that the algorithm needs over time.
- `data`: An object that provides access to current and historical pricing and volume data, among other things.

Before every trading day, we fetch the output of a defined pipeline named `factor_pipeline`. The `pipeline_output` function retrieves the computed results of the named pipeline for the current day. These results are then stored in the `context` namespace under the `factor_data` key. This allows the algorithm to access and use the pipeline's output in subsequent functions or during the trading day.

IMPORTANT NOTE

The `context` object in the Zipline Reloaded backtesting framework serves as a persistent namespace, allowing algorithms to store and manage variables across multiple function calls and trading sessions. It retains its values throughout the algorithm's execution, acting as a central repository for data, counters, flags, and other essential information that the algorithm requires for its operations and decision-making processes.

The `initialize` function is used to set up initial configurations and operations that the algorithm will use throughout its execution. Here, the function first attaches a data pipeline, created by the `make_pipeline` method, and names it `factor_pipeline`. Subsequently, the `schedule_function` method is used to schedule the `rebalance` function to run at the start of each week at the market's opening time, using the US equities trading calendar. This ensures that the algorithm will regularly adjust its portfolio based on the logic defined in the `rebalance` function following the exchange's open days.

The `rebalance` function adjusts the portfolio based on specific criteria. Initially, the function retrieves the factor data stored in the `context` object and adds the rank to the output DataFrame using the `record` function. It then fetches the current prices of the assets under consideration and adds them to the output DataFrame as well. The assets are then categorized into three groups: `longs` (assets the algorithm intends to buy), `shorts` (assets it plans to sell short), and `divest` (current portfolio holdings in neither the `longs` nor `shorts` list, indicating they should be sold off).

We implement a function called `exec_trades` to abstract away the order execution. For each asset in the list, the function first checks whether the asset is tradable and that there are no open orders for it using the `can_trade` and `get_open_orders` methods. If both conditions are met, the function places an order for the asset to adjust its position to the desired target percentage using the `order_target_percent` method.

Finally, we call `run_algorithm` to start the backtest, taking in the start and end dates, the `initialize` and `before_trading_start` functions to set up and prepare for each trading day, a starting capital of \$100,000, and specifying the data bundle as `quandl`. The results of the backtest are stored in the `perf` variable for further analysis.

There's more...

The output of a Zipline Reloaded backtest provides a comprehensive overview of an algorithm's performance. It includes a time series of key metrics, such as portfolio value, returns, and specific asset positions. Additionally, it captures risk metrics, transaction logs, and other diagnostic data that helps in evaluating the strategy's robustness and potential pitfalls.

The output can be used with the risk and performance libraries `pyfolio` and `alphalens`, which we dive into in the next two chapters. For now, we'll inspect a few of the key outputs:

```
perf.portfolio_value.plot(title="Cumulative returns")
```

The result is a plot showing the cumulative equity of your algorithm.

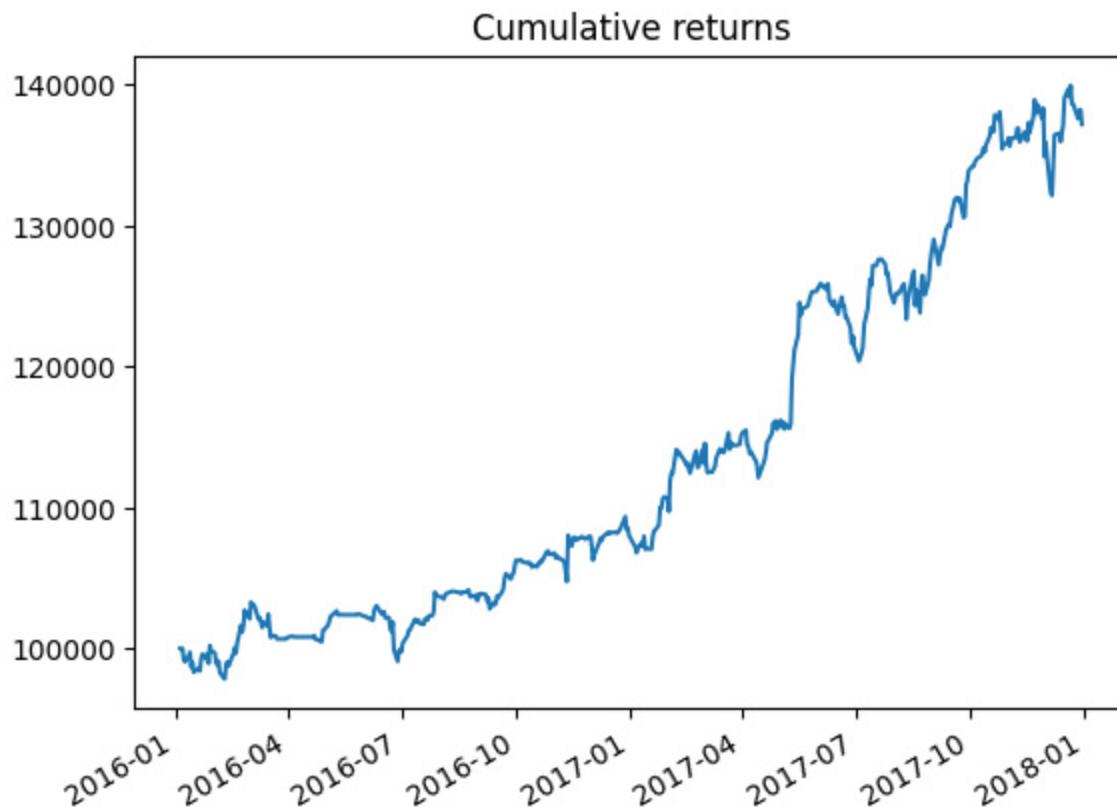


Figure 8.3: Cumulative equity of the algorithm

Create a histogram of daily returns:

```
perf.returns.hist(bins=50)
```

The result is a histogram showing the frequency of daily returns across 50 bins:

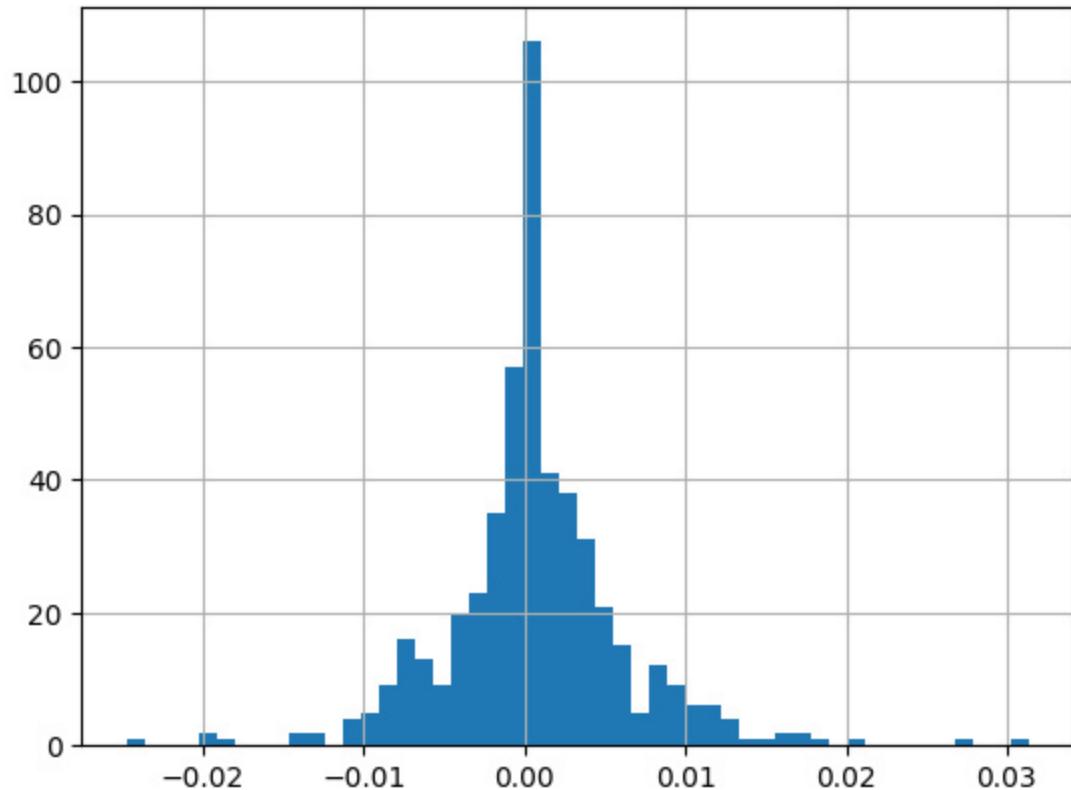


Figure 8.4: Histogram of daily portfolio returns

Plot the rolling Sharpe ratio for the algorithm:

```
perf.sharpe.plot()
```

The result is a plot visualizing the rolling Sharpe ratio of the algorithm:

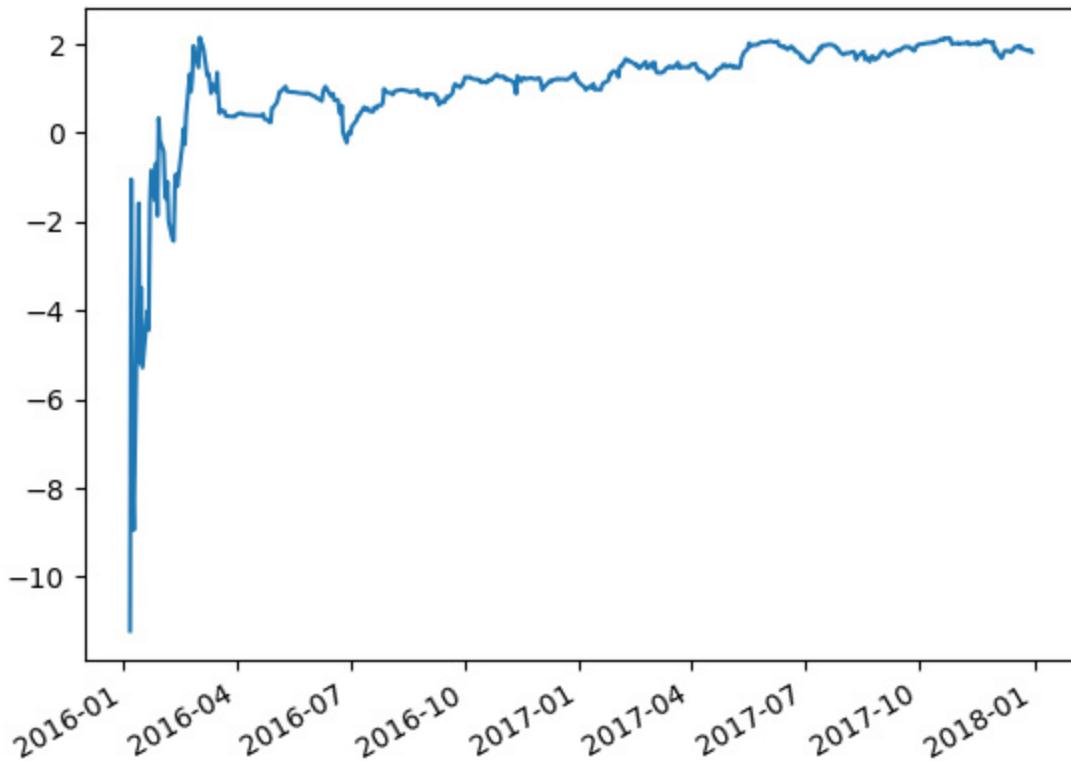


Figure 8.5: Plot of the rolling Sharpe ratio of the algorithm

HINT

The output is a pandas DataFrame, which means all the data manipulation methods we've learned thus far apply.

See also

Zipline Reloaded offers robust documentation for using its extensive features. You can find them here:

- Zipline Reloaded documentation: <https://zipline.ml4trading.io>
- An article that describes ingesting the premium US equities data: <https://www.pyquantnews.com/free-python-resources/how-to-ingest-premium-market-data-with-zipline-reloaded>

Exploring a mean reversion strategy with Zipline Reloaded

Mean reversion strategies are based on the financial principle that asset prices and returns eventually revert to their long-term mean or average level after periods of divergence or deviation. These strategies operate on the assumption that prices will bounce back to a historical mean or some form of equilibrium after moving away from it, either due to overreaction or other short-term factors.

Mean reversion suggests that assets are subject to inherent and stable equilibriums. When prices deviate significantly from these equilibriums, due to factors such as emotional trading, news, or events, they are likely to revert back over time. Traders and algorithms identify assets that have deviated significantly from their historical average price or some other benchmark. This deviation can be measured using various metrics, such as z-scores, Bollinger Bands, or percentage deviations. In this recipe, we use the z-score.

In this recipe, we'll use the Zipline Reloaded factor framework to build a portfolio. The strategy buys the top oversold stocks and sells the top overbought stocks.

Getting ready

Most of this recipe will be the same as the recipe demonstrating the momentum factor, with some notable differences:

- We've updated the factor to measure mean reversion and rank our universe based on the top and bottom mean-reverting assets
- We include commission and slippage rules to enhance the realism of the backtest
- We add simple logging to provide feedback during the execution of the algorithm and to demonstrate the available portfolio attributes
- We download benchmark price data and include returns for comparison against our algorithm
- We include a custom function that is executed when the backtest is complete
- We generate an image of the pipeline that shows how the logic filters the stocks in the universe

We assume you have the libraries imported and will skip that step.

How to do it...

We will expand on the previous recipe by adding complexity to our analysis:

1. Set the number of longs and shorts and the lookback periods:

```
N_LONGS = N_SHORTS = 50
MONTH = 21
YEAR = 12 * MONTH
```

2. Create the mean reversion factor:

```
class MeanReversion(CustomFactor):
    inputs = [Returns(window_length=MONTH) ]
    window_length = YEAR
    def compute(self, today, assets, out,
               monthly_returns):
        df = pd.DataFrame(monthly_returns)
        out[:] = df.iloc[-1].sub(
            df.mean()).div(df.std())
```

3. Implement the function that returns the pipeline using the factor:

```
def make_pipeline():
    mean_reversion = MeanReversion()
    dollar_volume = AverageDollarVolume(
        window_length=30)
    return Pipeline(
        columns={
            "longs": mean_reversion.bottom(N_LONGS),
            "shorts": mean_reversion.top(N_SHORTS),
            "ranking": mean_reversion.rank(
                ascending=False),
        },
        screen=dollar_volume.top(100),
    )
```

4. Implement the function that hooks into the event that fires before trading starts:

```
def before_trading_start(context, data):
    context.factor_data = pipeline_output(
        "factor_pipeline")
```

5. Implement the function that is invoked when the backtest begins. Note the addition of commission and slippage models:

```
def initialize(context):
    attach_pipeline(make_pipeline(),
                  "factor_pipeline")
    schedule_function(
        rebalance,
        date_rules.week_start(),
        time_rules.market_open(),
        calendar=calendars.US_EQUITIES,
    )
    set_commission(
        us_equities=commission.PerShare(
            cost=0.00075, min_trade_cost=0.01
        )
    )
    set_slippage(
        us_equities=slippage.VolumeShareSlippage(
            volume_limit=0.0025, price_impact=0.01
        )
    )
```

6. Add a **print** statement to the same **rebalance** function we created in the last recipe. This **print** statement provides feedback as the algorithm is running:

```
def rebalance(context, data):
    factor_data = context.factor_data
    record(factor_data=factor_data.ranking)
```

7. In the next section of the **rebalance** function, extract the symbols from the **factor_data** DataFrame and record the asset prices:

```
assets = factor_data.index
record(prices=data.current(assets, "price"))
```

8. Now we identify the assets to go long, to go short, and to divest from the portfolio:

```
longs = assets[factor_data.longs]
shorts = assets[factor_data.shorts]
```

```

        divest = set(
            context.portfolio.positions.keys() - set(
                longs.union(shorts)))

```

9. Finally, we print output to the user and call the **exec_trades** function to execute our desired orders:

```

print(
    f"{{get_datetime().date()} | Longs {len(longs)} | Shorts {len(shorts)} | "
{context.portfolio.portfolio_value}"
)
exec_trades(
    data,
    assets=divest,
    target_percent=0
)
exec_trades(
    data,
    assets=longs,
    target_percent=1 / N_LONGS
)
exec_trades(
    data,
    assets=shorts,
    target_percent=-1 / N_SHORTS
)

```

10. Implement the same **exec_trades** function as in the previous recipe:

```

def exec_trades(data, assets, target_percent):
    for asset in assets:
        if data.can_trade(
            asset) and not get_open_orders(asset):
            order_target_percent(
                asset, target_percent)

```

11. The **analyze** function is run after the backtest is complete. We have access to the **context** object and the output of the backtest in the **perf** DataFrame. This is useful to run reports or event trigger alerts if certain thresholds are passed. In this example, we simply plot the portfolio value:

```

def analyze(context, perf):
    perf.portfolio_value.plot()

```

12. Use **pandas_datareader** to compute the daily returns of a benchmark. In this case, we use the S&P 500 index:

```

start = pd.Timestamp("2016")
end = pd.Timestamp("2018")
sp500 = web.DataReader('SP500', 'fred', start,
                      end).SP500
benchmark_returns = sp500.pct_change()

```

13. Finally, run the backtest and cache the output:

```

perf = run_algorithm(
    start=start,
    end=end,
    initialize=initialize,
    analyze=analyze,
    benchmark_returns=benchmark_returns,
    before_trading_start=before_trading_start,
    capital_base=100_000,
    bundle="quandl"
)

```

```

)
perf.to_pickle("mean_reversion.pickle")

```

While the backtest is running, you'll see output that resembles the following:

2016-01-04		Longs 0		Shorts		1		100000.0
2016-01-11		Longs 0		Shorts		1		99993.16349999886
2016-01-19		Longs 2		Shorts		1		100021.36374999832
2016-01-25		Longs 2		Shorts		2		100047.47224999769
2016-02-01		Longs 3		Shorts		3		99889.9749999967
2016-02-08		Longs 2		Shorts		2		99616.54649999554
2016-02-16		Longs 2		Shorts		0		99849.90549999391
2016-02-22		Longs 1		Shorts		0		100195.42449999285
2016-02-29		Longs 1		Shorts		0		100155.76424999181
2016-03-07		Longs 2		Shorts		1		100114.18424999181
2016-03-14		Longs 0		Shorts		0		100304.28374999072
2016-03-21		Longs 1		Shorts		2		99210.60974998782
2016-03-28		Longs 1		Shorts		0		99355.18649998584
2016-04-04		Longs 1		Shorts		0		99108.98424998432
2016-04-11		Longs 2		Shorts		1		99483.63424998433
2016-04-18		Longs 2		Shorts		0		99556.3877499843
2016-04-25		Longs 2		Shorts		0		99573.0884999835
2016-05-02		Longs 4		Shorts		0		99557.5844999825
2016-05-09		Longs 3		Shorts		1		99399.40899998133
2016-05-16		Longs 3		Shorts		2		99355.67949998101
2016-05-23		Longs 3		Shorts		2		99271.2524999799
2016-05-31		Longs 1		Shorts		2		99356.98699997793
2016-06-06		Longs 4		Shorts		3		99366.66949997128
2016-06-13		Longs 3		Shorts		1		99543.62649996586
2016-06-20		Longs 2		Shorts		1		99459.02499996335
2016-06-27		Longs 4		Shorts		1		98925.75674996113
2016-07-05		Longs 0		Shorts		3		99260.76674996056
2016-07-11		Longs 3		Shorts		2		99335.8389999511
2016-07-18		Longs 1		Shorts		1		99421.94524994826
2016-07-25		Longs 0		Shorts		2		99641.29024993867
2016-08-01		Longs 0		Shorts		2		99574.93024993711

Figure 8.6: Logs from the algorithm run

- When the backtest completes, the `analyze` function is invoked and the performance is plotted:

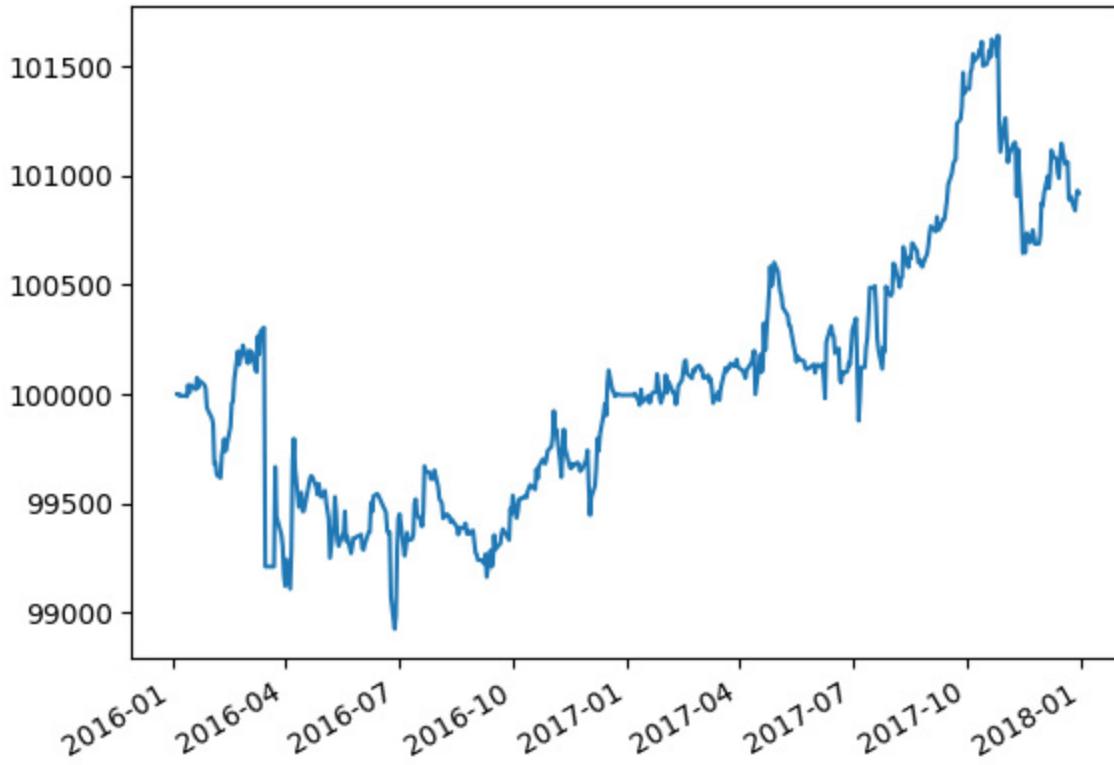


Figure 8.7: Plot automatically generated from the `analyze` function

IMPORTANT NOTE

The keen eye may spot what looks like a bug: why are we taking long or short positions in only a few stocks when we've set the number of longs and shorts to 50? The answer lies in the sequence in which the `Pipeline` class processes its filters. It's important to note that we've included a screen argument tied to the `dollar_volume` factor. This effectively filters the stocks in the pipeline to only those with a dollar volume exceeding \$100,000, and this screening occurs after the long and short selections are made. If you were to remove this screening criterion, the log would display 50 longs and 50 shorts as expected. Top of Form

Bottom of Form

How it works...

Our custom factor uses monthly returns as its input, specified by the `Returns` class with a window length set to a constant, `MONTH`, representing a month. The overall window length for the factor is set to a constant, `YEAR`, representing a year. Within the `compute` method, the monthly returns are converted into a DataFrame. The factor's value for each asset is then computed by taking the last month's return, subtracting the mean of all monthly returns, and then dividing by the standard deviation of those returns. This results in a z-score-like metric, indicating how many standard deviations the latest

month's return is from the mean, which can be used to gauge mean reversion tendencies for each asset.

The `make_pipeline` function is similar to the one we constructed in the last recipe except it uses the mean reversion factor. Within the function, an instance of the `MeanReversion` factor is created, which calculates a mean reversion score for each asset. Additionally, the `AverageDollarVolume` class is used to compute the average dollar volume over a 30-day window for liquidity screening. The core of the function is the `Pipeline` object, which is set up to produce three columns:

- `longs`: Identifies the assets with the lowest mean reversion scores (i.e., the most undervalued assets)
- `shorts`: Pinpoints the assets with the highest scores (i.e., the most overvalued assets)
- `ranking`: Provides a rank for each asset based on its mean reversion score in descending order

To ensure the pipeline focuses on liquid assets, a screen is applied that only considers the top 100 assets based on their average dollar volume.

In the `initialize` function, we attach the pipeline and schedule the rebalancing in the same way as the last recipe. However, in this implementation, we include commission and slippage models. The `set_commission` function configures the commission model for US equities to be based on a per-share cost. Specifically, the algorithm will be charged \$0.00075 for each share traded, with a minimum trade cost set at \$0.01, ensuring that even small trades incur a nominal fee. The `set_commission` function allows for a large range of commission models to match your actual broker's commission schedule. Following this, the `set_slippage` function establishes the slippage model for US equities using the `volumeShareSlippage` method. This model simulates the impact of an order on the stock price based on the order's size relative to the stock's average volume. The parameters dictate that an order can consume up to 0.25% of the stock's daily volume and that each order will impact the stock price by 1%.

Finally, to compare our algorithm's return to the benchmark, we compute daily returns for the S&P 500 index.

There's more...

Our pipeline is simple: we generate a mean reversion factor, screen the top and bottom stocks for a dollar volume greater than \$100,000, and return the results. Zipline Reloaded supports compound factor models where several factors are combined. In those cases, it helps to visualize what's going on. Luckily, we can generate an illustration of how the pipeline is constructed:

```
p = make_pipeline()
p.show_graph()
```

The result is a pictorial depiction of the pipeline:

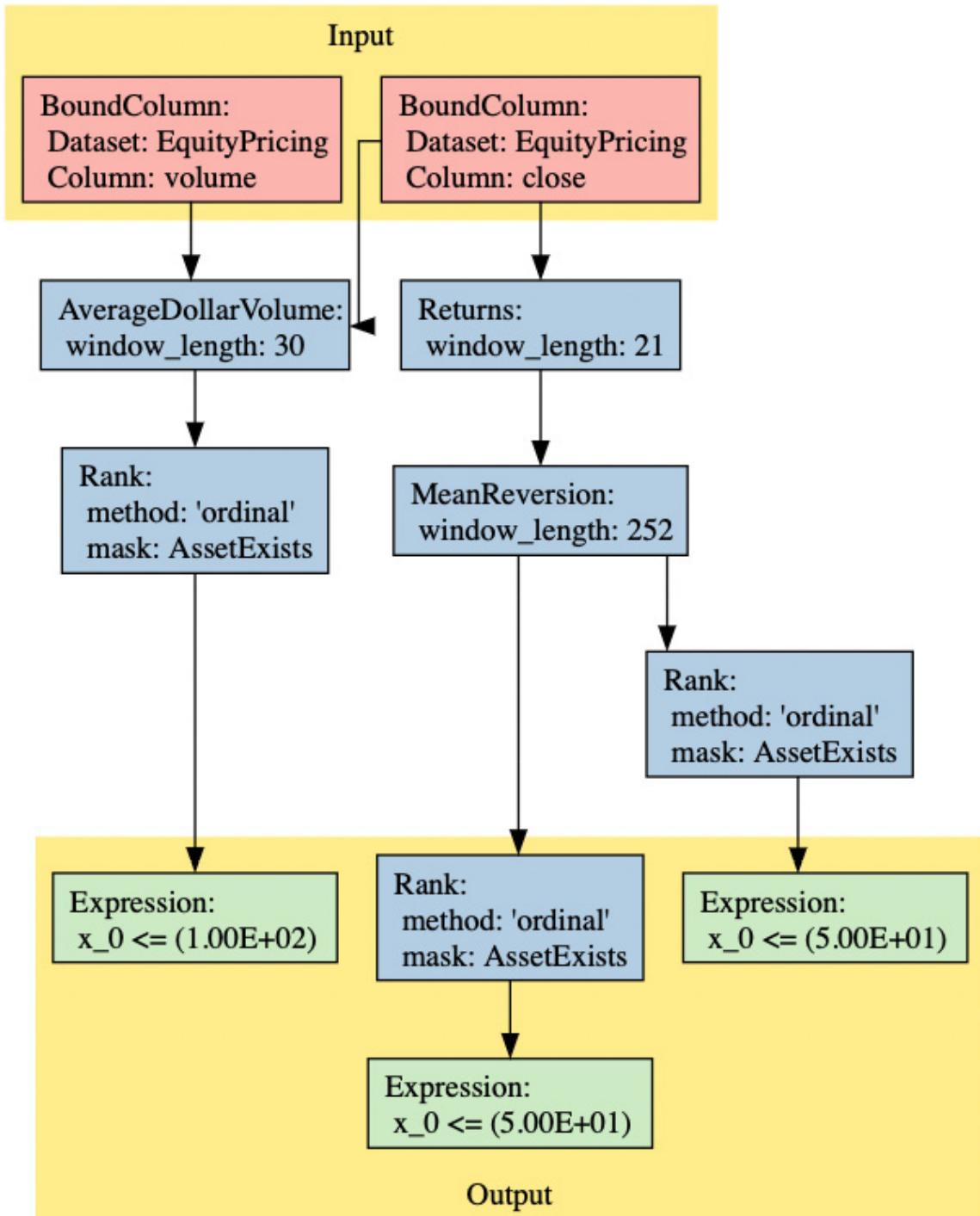


Figure 8.8: Graphical representation of the pipeline

By including benchmark returns, Zipline Reloaded computes the rolling alpha and beta of our portfolio against the benchmark. We learned how to hedge the beta and amplify the alpha in the last

chapter. Using the output of the backtest, we have these metrics at our disposal.

Plot the rolling beta against the benchmark:

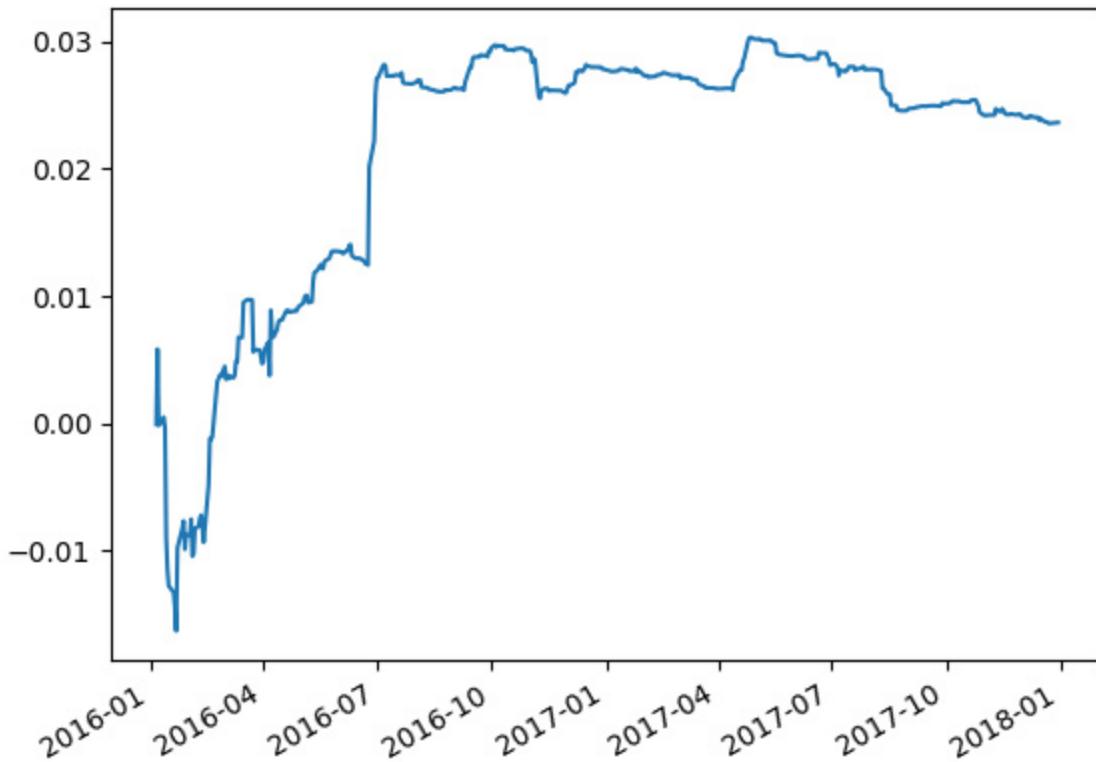


Figure 8.9: Rolling beta of our algorithm's returns against the benchmark

Plot the rolling alpha against the benchmark:

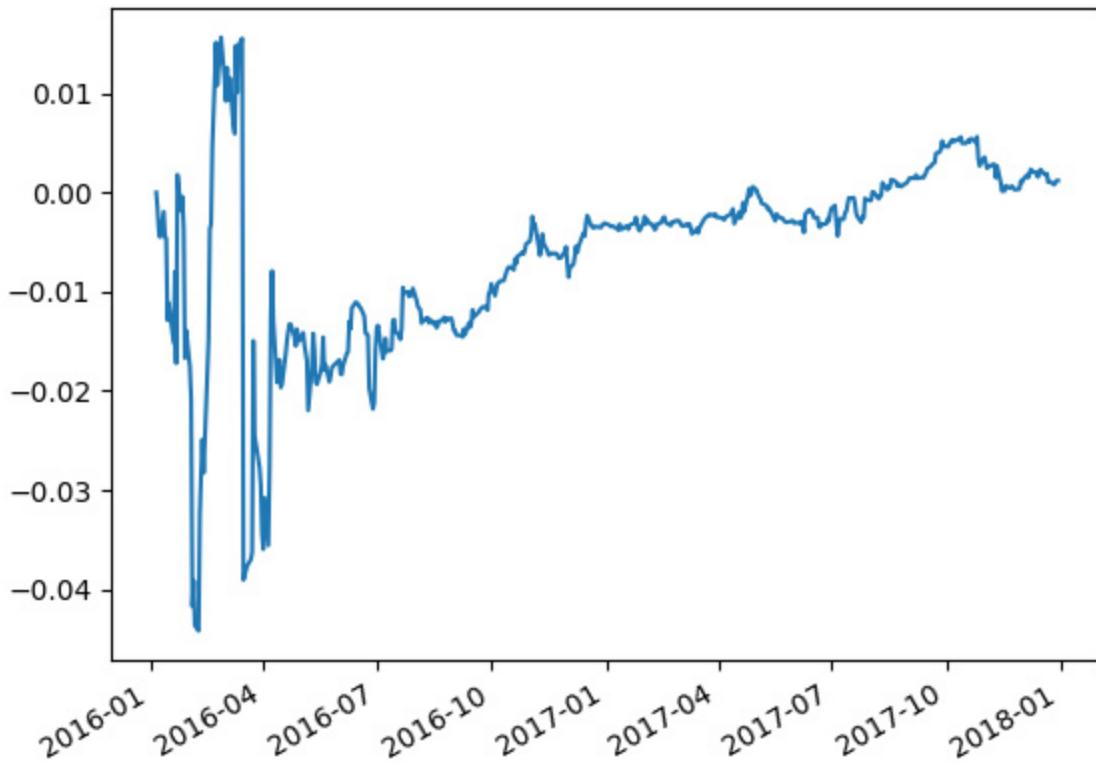


Figure 8.10: Rolling alpha of our algorithm's returns against the benchmark

See also

We went deeper into some more advanced features of Zipline Reloaded. These features aim to create a more realistic simulation of the market dynamics algorithmic traders face every day. To dive deeper, check out the documentation listed here:

- API documentation for the Pipeline API, which describes its available input parameters: <https://zipline.ml4trading.io/api-reference.html#pipeline-api>
- Different built-in slippage models available within Zipline Reloaded: <https://zipline.ml4trading.io/api-reference.html#slippage-models>
- Different built-in commission models available within Zipline Reloaded: <https://zipline.ml4trading.io/api-reference.html#commission-models>

Evaluate Factor Risk and Performance with Alphalens Reloaded

Factor investing is a strategic approach where assets are chosen based on attributes or **factors** that are associated with higher returns. This method differs from traditional investment strategies which focus on asset classes like stocks, bonds, or sectors. Factor investing emphasizes the underlying drivers of risk and return in securities. The crux of factor investing lies in the systematic identification and harnessing of these and other factors. By understanding the sources of risk and return, we can aim for returns above traditional benchmarks. It's essential to note, however, that while factor investing can enhance portfolio diversification and potential returns, it does not eliminate risk. Market conditions, economic changes, and other externalities can influence the effectiveness of factor-based strategies at any given time.

In [Chapter 5, Build Alpha Factors for Stock Portfolios](#), we explored recipes to construct alpha factors. In this chapter, we explore how to analyze the risk and performance of these alpha factors using Alphalens Reloaded. Alphalens Reloaded is a library designed specifically for performance analysis of predictive alpha factors. It's useful for assessing the quality of signals generated from various factors, which allow us to evaluate how well these signals predict future returns. Alphalens Reloaded integrates with Zipline Reloaded and turns the output of backtests into statistics and visualizations. The library provides various utilities, including tear sheets that consolidate performance metrics and visualizations. These tear sheets can show cumulative returns, turnover analysis, and information coefficients, among other insights.

In this chapter, we present the following recipes:

- Preparing backtest results
- Evaluating the information coefficient
- Examining factor return performance
- Evaluating factor turnover

Preparing backtest results

Zipline Reloaded is a robust backtesting library that has an integrated ecosystem of tools designed to assess trading strategy performance. This ecosystem makes it easier for traders to transition from

strategy development to evaluation. An example of an integrated tool is Alphalens Reloaded which is the focus of this chapter.

We learned in [Chapter 7, Event-Based Backtesting Factor Portfolios with Zipline Reloaded](#) that the output DataFrame of a Zipline backtest provides a detailed analysis of a trading strategy's performance over a specified historical data period. The output includes metrics like cumulative returns, alpha, beta, Sharpe ratio, and maximum drawdown, among many others. We need to manipulate the output DataFrame to extract some of the data so it's suitable for use with Alphalens Reloaded.

This recipe will walk through the process of extracting the relevant information.

Getting ready...

To install Alphalens Reloaded in your virtual environment, use `pip`:

```
pip install alphalens-reloaded
```

We assume that a file `mean_reversion.pickle` exists in the same directory as the code for this recipe. The file is the cached output from the Zipline backtest we ran in [Chapter 5, Build Alpha Factors for Stock Portfolios](#).

How to do it...

This recipe focuses on manipulating the output of the Zipline backtest using pandas. There's one Alphalens Reloaded method we can use to create DataFrame used for most analysis in Alphalens Reloaded.

1. Import the libraries we need for the analysis:

```
import pandas as pd
from alphalens.utils import get_clean_factor_and_forward_returns
```

2. Read in the cached backtest output into a DataFrame:

```
mean_reversion = pd.read_pickle('mean_reversion.pickle')
```

3. Construct a DataFrame with symbols in the columns and dates in the rows:

```
prices = pd.concat(
    [df.to_frame(d) for d, df in mean_reversion.prices.dropna().items()],
    axis=1
).T
```

IMPORTANT

A Python list comprehension is a concise way to create lists by iterating over an iterable and applying an expression to each element. It condenses a loop and a list appending operation into a single line of code, making it more readable and elegant. The syntax includes brackets containing an expression followed by a for clause, as we see in Step 3, above.

4. Convert column names to strings:

```
prices.columns = [col.symbol for col in prices.columns]
```

5. Normalize the timestamps to midnight, preserving time zone information:

```
prices.index = prices.index.normalize()
```

The result is the following DataFrame containing the prices of each asset in the backtest for each day:

	AAL	AAPL	ABBV	AET	AGN	...	ANTM	AMT	PCG	CBS	TMO
2016-01-04 00:00:00+00:00	40.91	105.35	57.61	109.26	307.47	...	NaN	NaN	NaN	NaN	NaN
2016-01-05 00:00:00+00:00	40.91	105.35	57.61	109.26	307.47	...	NaN	NaN	NaN	NaN	NaN
2016-01-06 00:00:00+00:00	40.91	105.35	57.61	109.26	307.47	...	NaN	NaN	NaN	NaN	NaN
2016-01-07 00:00:00+00:00	40.91	105.35	57.61	109.26	307.47	...	NaN	NaN	NaN	NaN	NaN
2016-01-08 00:00:00+00:00	40.91	105.35	57.61	109.26	307.47	...	NaN	NaN	NaN	NaN	NaN
...
2017-12-22 00:00:00+00:00	NaN	176.42	98.19	177.34	170.91	...	227.02	NaN	NaN	59.00	NaN
2017-12-26 00:00:00+00:00	NaN	170.57	97.75	180.42	164.44	...	225.43	NaN	44.45	60.19	NaN
2017-12-27 00:00:00+00:00	NaN	170.57	97.75	180.42	164.44	...	225.43	NaN	44.45	60.19	NaN
2017-12-28 00:00:00+00:00	NaN	170.57	97.75	180.42	164.44	...	225.43	NaN	44.45	60.19	NaN
2017-12-29 00:00:00+00:00	NaN	170.57	97.75	180.42	164.44	...	225.43	NaN	44.45	60.19	NaN

Figure 8.1: Price data from the backtest output

6. We repeat a similar process for the factor data. Start by constructing a DataFrame with symbols in the columns and factor rank in the rows:

```
factor_data = pd.concat(
    [df.to_frame(d) for d,
     df in mean_reversion.factor_data.dropna().items()],
    axis=1
).T
```

7. Convert column names to strings:

```
factor_data.columns = [
    col.symbol for col in factor_data.columns]
```

8. Normalize the timestamps to midnight, preserving time zone information:

```
factor_data.index = factor_data.index.normalize()
```

9. Create a MultiIndex with **date** in level 0 and **symbol** in level 1:

```
factor_data = factor_data.stack()
```

10. Rename the MultiIndex:

```
factor_data.index.names = ["date", "asset"]
```

The result is the following a MultiIndex series with the date and asset in the indexes and the factor ranking in the column:

```
date                                asset
2016-01-04 00:00:00+00:00    AAL      1156.0
                               AAPL     2547.0
                               ABBV     438.0
                               AET      893.0
                               AGN     1371.0
                               ...
2017-12-29 00:00:00+00:00    ISRG     2449.0
                               DWDP     1277.0
                               ANTM     1510.0
                               PCG      2440.0
                               CBS      292.0
Length: 50275, dtype: float64
```

Figure 8.2: MultiIndex Series containing factor ranks for each day and asset

11. The last step is to create a MultiIndex DataFrame with forward returns, factor values, and factor quantiles:

```
alphalens_data = get_clean_factor_and_forward_returns(
    factor=factor_data, prices=prices, periods=(5, 10, 21, 63)
)
```

The result is the following MultiIndex DataFrame containing the data we need to run all the factor analysis using Alphalens Reloaded:

			5D	10D	21D	63D	factor	factor_quantile
	date	asset						
2016-01-04 00:00:00+00:00	AAL	0.004155	-0.050110	-0.037399	0.041066	1156.0	3	
	AAPL	-0.064737	-0.082487	-0.084670	0.054770	2547.0	5	
	ABBV	-0.064746	-0.045478	-0.055893	0.027773	438.0	1	
	AET	-0.036061	-0.040454	-0.053908	-0.064800	893.0	2	
	AGN	-0.026344	-0.050997	-0.080561	-0.097310	1371.0	4	
...
2017-09-29 00:00:00+00:00	ADP	0.004595	0.004595	0.004595	0.004595	1239.0	3	
	COL	0.011153	0.024537	0.035920	0.035920	2196.0	4	
	BBY	0.059996	0.089902	0.089902	0.089902	2486.0	5	
	EFX	0.025883	0.068132	0.037301	0.027976	2501.0	5	
	SBAC	0.009967	0.044323	0.044323	0.044323	2390.0	5	

Figure 8.3: MultiIndex DataFrame containing clean factor and forward returns for each date and asset

How it works...

After the imports, we need to manipulate the backtest output. First, we construct a DataFrame called `prices` where each column represents a symbol and each row represents a date. We do this by iterating through each row of `prices` using the `items` method which returns an iterator. Each key is the date and each value is a DataFrame representing price data. Each of these DataFrames is converted to a column with the date as the column's name using the `to_frame()` method. The `concat()` function then concatenates these columns horizontally using the `axis=1` argument.

Next, we modify the column names of the `prices` DataFrame to be the symbols since the original column names were Equity objects. Finally, we adjust the row indices of the `prices` DataFrame to normalize the dates to midnight, while retaining any time zone information.

To extract the factor data from the backtest output, we follow similar steps. First, we iterate through `factor_data` column dropping any null values, and reshaping the data so that each date corresponds to a unique column. Next, we convert the column names to strings and adjust the dates like before. Finally, the DataFrame is transformed from a two-dimensional table into a one-dimensional series with a hierarchical index. We do this with the `stack` method, resulting in a multi-indexed Series where the primary level of the index is the date and the secondary level is the symbol. For clarity, these levels are named `date` and `asset`.

Now that the DataFrames are prepared, we can call the Alphalens Reloaded `get_clean_factor_and_forward_returns` function. The purpose of this function is to associate factor values with future returns. The output, stored in the variable `alphalens_data`, is a DataFrame that combines the factor data with the subsequent returns based on the provided prices. This merged structure allows us to analyze how specific factor values might have influenced or related to future asset returns. The resulting DataFrame has a MultiIndex with dates and symbols. The columns in the DataFrame represent different forward returns (5D, 10D, 21D, and 63D for 5-day, 10-day, 21-day, and 63-day returns), the factor values and quantile rankings of the factor values. The quantile rankings segment the factor values into groups, with the shown values like “3” or “5” indicating the specific quantile bucket the factor value for an asset falls into on a given date.

This DataFrame will be used for all subsequent analysis.

Top of Form

Bottom of Form

There's more...

The `get_clean_factor_and_forward_returns` function from Alphalens Reloaded is designed to prepare factor data for subsequent analysis. It aligns factor values with forward returns, ensuring that comparisons between factors and future performance are valid. The function also handles data cleaning tasks such as handling missing values. The result is a structured DataFrame that is optimized for factor analysis. The `get_clean_factor_and_forward_returns` is flexible in how it aligns the factor data with return data. These are the additional arguments available for controlling this alignment:

- **groupby**: An optional parameter, which, if provided, groups assets based on common characteristics like industry or sector, facilitating sector-neutral analysis.
- **by_group**: A boolean that decides whether to perform computations (like quantile analysis) separately for each group in `groupby` or for the entire dataset.
- **bins**: The number of quantiles to form or specific bin edges. This can be an alternative to the `quantiles` argument.
- **quantiles**: An integer denoting how many quantiles to form. Quantiles segment the factor data, allowing for the examination of returns by factor quantile.
- **periods**: A list of time periods for which forward returns will be computed, e.g., [1, 5, 10] would yield forward returns for 1 day, 5 days, and 10 days.
- **filter_zscore**: Removes outliers based on the z-score. This ensures that extreme factor values, which could be errors or anomalies, do not skew the analysis.
- **groupby_labels**: An optional set of labels for the groups when using the `groupby` argument, useful for more meaningful group naming.
- **max_loss**: The maximum percentage loss that is tolerated in case of missing data. Determines how much missing data is acceptable.

- `zero_aware`: A boolean that, if `True`, makes quantile computation treat zeros distinctly, ensuring that they receive their own bin. This is especially useful when zero has a distinct meaning in the factor context.

See also

Factor investing is a broad topic. *Active Portfolio Management: A Quantitative Approach for Producing Superior Returns and Controlling Risk* by Richard C. Grinold and Ronald N. Khan is a staple in the world of quantitative finance and portfolio management. It dives deep into the mathematics and strategies behind managing portfolios actively, including factor-based investing. The book covers various factors, models, and tools, emphasizing the importance of forward returns in the evaluation and prediction of factor performance. For practitioners and students alike, this book provides a comprehensive and rigorous exploration of quantitative techniques in portfolio management. You can buy the book on Amazon here: <https://amzn.to/3RtsovG>

Finally, the Alphalens Reloaded documentation provides examples and details on how to use the code. The documentation can be found here: <https://alphalens.ml4trading.io>

Evaluating the information coefficient

The **Information Coefficient (IC)** is a fundamental metric in quantitative portfolio construction. It gauges the predictive power of a forecast relative to future returns. At its core, IC uses the Spearman rank correlation, a non-parametric measure that assesses how well the relationship between two variables can be described using a monotonic function. The IC's value ranges from -1 to 1 with a positive IC indicating a forecast's genuine predictive power, a value near zero indicating an absence of predictive capacity, and a negative value indicating an inverse relationship between the forecast and subsequent returns.

The origins of the IC can be traced back to the 1960s and 1970s. The preliminary idea of the metric was introduced by Jack L. Treynor in the early 1960s as he presented the topic of correlating investment decisions with corresponding outcomes in his writings on performance measurement. By the mid-1970s, Fischer Black, who is also co-credited for the renowned Black-Scholes option pricing model, further honed and expanded upon the concept in academic literature. Today, IC is a cornerstone metric to gauge the predictive power of alpha factors.

Alphalens Reloaded consolidates extensive quantitative research into a single tool tailored for factor evaluation. It simplifies complex methodologies, offering us an easy way to analyze the predictive capabilities of factors.

This recipe will show us how.

Getting ready...

For this recipe and subsequent recipes, we assume the imports from the last recipe are available and `alphalens_data` is defined.

How to do it...

Alphalens Reloaded provides tear sheets that provide consolidated summaries of the analyses presented in this recipe. We'll look at each independently to deepen our understanding.

1. Import additional libraries from Alphalens Reloaded to evaluate the IC:

```
from alphalens.performance import (
    factor_information_coefficient,
    mean_information_coefficient,
)
from alphalens.plotting import (
    plot_ic_ts,
    plot_information_table,
)
```

2. Generate the information coefficient for each holding period on each date:

```
ic = factor_information_coefficient(alphalens_data)
```

The result is the following DataFrame with the IC for each day and forward return:

		5D	10D	21D	63D
	date				
	2016-01-04 00:00:00+00:00	-0.248066	-0.284107	-0.140796	-0.106800
	2016-01-05 00:00:00+00:00	-0.248066	-0.284107	-0.140796	-0.106800
	2016-01-06 00:00:00+00:00	-0.248066	-0.284107	-0.140796	-0.106800
	2016-01-07 00:00:00+00:00	-0.248066	-0.284107	-0.182742	-0.106800
	2016-01-08 00:00:00+00:00	-0.248066	-0.263740	-0.182742	-0.085559

	2017-09-25 00:00:00+00:00	-0.009765	0.020817	-0.009796	-0.037130
	2017-09-26 00:00:00+00:00	-0.009765	0.020817	-0.009796	-0.094793
	2017-09-27 00:00:00+00:00	-0.009765	0.020817	-0.009796	-0.094793
	2017-09-28 00:00:00+00:00	-0.009765	0.020817	-0.009796	-0.094793
	2017-09-29 00:00:00+00:00	-0.009765	0.020817	0.086333	-0.094793

Figure 8.4: DataFrame containing the IC for each period and forward return

3. Inspect the statistical properties of the IC at each forward return:

```
plot_information_table(ic)
```

The result is the following table containing the mean, standard deviation, and other properties of the IC:

	5D	10D	21D	63D
IC Mean	0.023	0.031	0.018	0.014
IC Std.	0.186	0.165	0.155	0.166
Risk-Adjusted IC	0.124	0.186	0.116	0.084
t-stat(IC)	2.599	3.894	2.426	1.757
p-value(IC)	0.010	0.000	0.016	0.080
IC Skew	0.153	0.104	0.530	0.404
IC Kurtosis	-0.247	-0.554	0.246	-0.221

Figure 8.5: DataFrame containing statistical properties of the IC including mean, standard deviation, risk adjusted IC, t-stat, p-value, skew, and kurtosis

4. Plot the IC over the forward return period to inspect how the alpha decays over longer holding periods as follows:

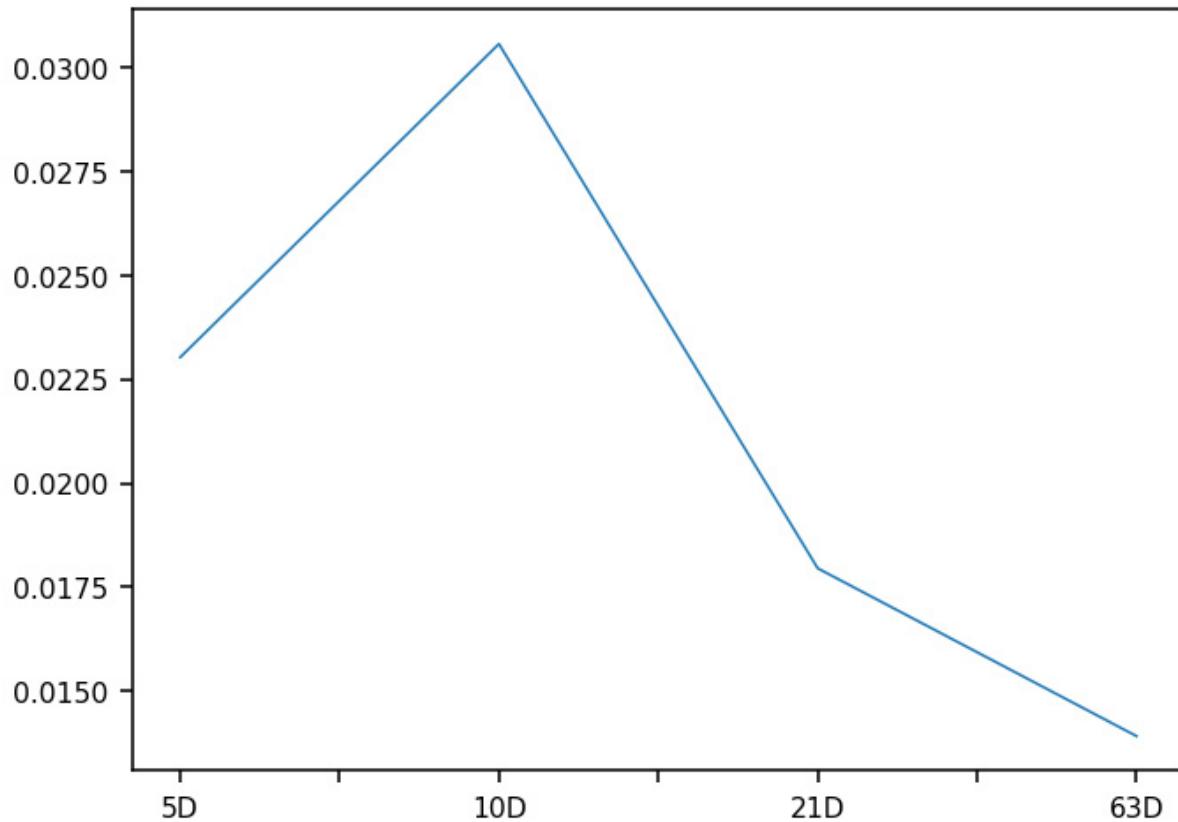


Figure 8.6: Plot demonstrating the decay of the IC as the forward return period lengthens

How it works...

The Spearman rank correlation, often denoted as ρ or Spearman's r , is a non-parametric measure used to assess the strength and direction of the monotonic relationship between two variables. Unlike the Pearson correlation coefficient, which measures linear relationships between continuous variables, Spearman's rank correlation focuses on the relative order of values rather than their absolute magnitudes.

To compute the Spearman rank correlation:

1. Rank each variable separately, assigning ranks from 1 to N for a dataset of N observations. For tied values, assign the average rank.
2. Calculate the difference between ranks for each observation, denoted as d .
3. Square these differences, d^2 .
4. Use the following formula to compute ρ :

$$\rho = 1 - \frac{6 \sum d^2}{N(N^2 - 1)}$$

Where:

- ρ is the Spearman rank correlation.
- d is the difference between paired ranks.
- N is the number of observations.

The resulting ρ will fall between -1 and 1 . A value of 1 indicates perfect positive correlation in ranks (i.e., as one variable increases, the other does too), -1 indicates perfect negative correlation in ranks, and a value close to 0 suggests no significant rank correlation.

In the context of the IC using the Spearman rank correlation:

- **Predicted or forward returns:** These are the returns that a factor model anticipates. They are typically derived from the factor scores, which are usually the output of some quantitative model or strategy.
- **Actual returns:** These are the realized returns of assets over a specific period. They serve as the ground truth against which predicted returns are evaluated.

When computing the Spearman rank correlation for the IC, the ranks of both predicted and actual returns are compared. The differences in ranks (dd) between these two sets of values are used in the Spearman formula to determine how well the predicted returns' order (or rank) matches the actual returns' order.

Alphalens Reloaded uses the Spearman rank correlation to compare the ranks of the factor values with the ranks of the forward returns. A high correlation would suggest that assets with higher factor scores tend to have higher (or lower, depending on the sign) future returns, affirming the predictive power of the factor.

In *Figure 8.6*, we observe the predictive power of the alpha factor decaying over time. This is referred to as IC decay. In algorithmic trading, it's important to measure how quickly the correlation between predicted and actual returns diminishes. A rapid IC decay indicates that the factor's predictive power is short-lived. Conversely, a slow IC decay suggests that the signal remains relevant for a more extended period, potentially allowing for longer investment horizons.

There's more...

Alphalens Reloaded ships with plotting functions to visualize how IC evolves through time. To see how the IC using 5-day forward returns evolves through time, use the `plot_ic_ts()` function:

```
plot_ic_ts(ic[["5D"]])
```

The result is a timeseries plot of the IC and its one-month moving average:

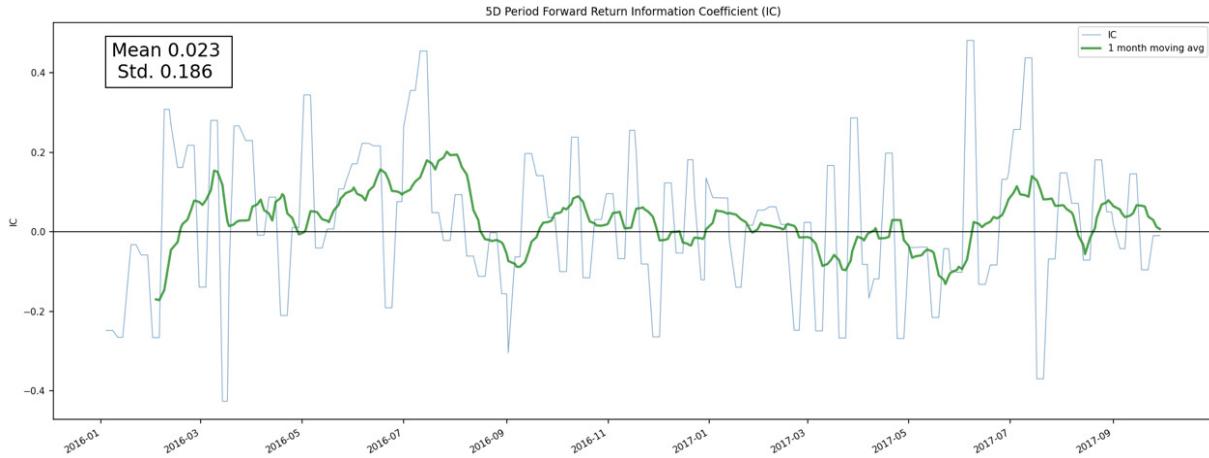


Figure 8.7: Evolution of the IC for 5-day forward returns

Since the return value of the `factor_information_coefficient()` function is a DataFrame, all the methods we learned previously can be used to further analyze the IC. This code resamples the daily IC into the quarterly mean and plots the value for each forward return:

```
ic_by_quarter = ic.resample("Q").mean()
ic_by_quarter.index = ic_by_quarter.index.to_period("Q")
ic_by_quarter.plot.bar(figsize=(14, 6))
```

The result is the following bar chart visualizing the mean IC over each quarter for each forward period :

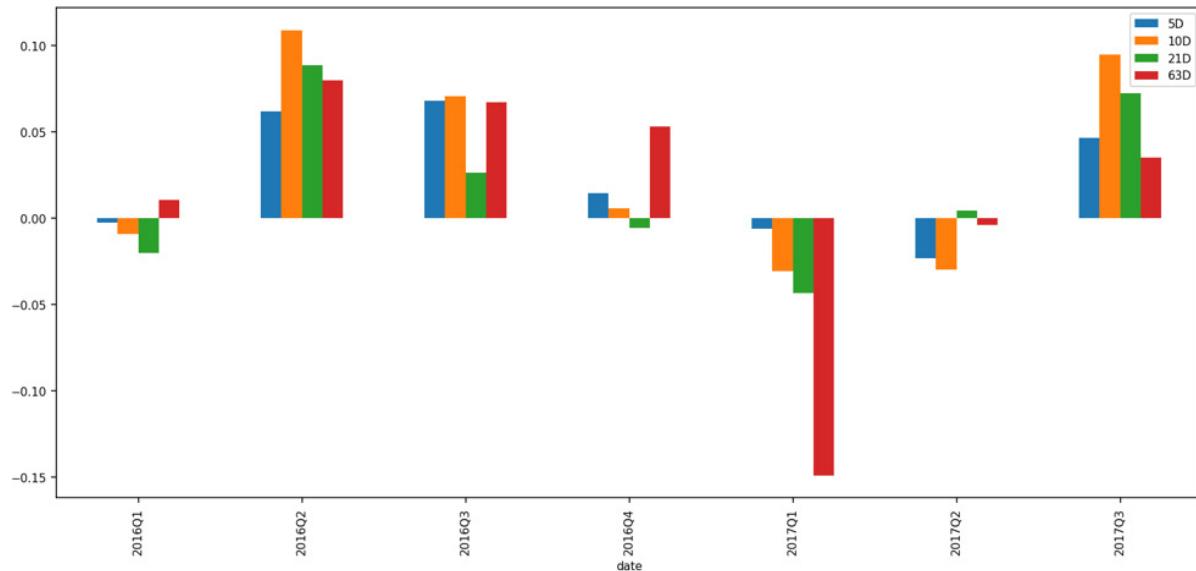


Figure 8.8: Mean IC value per quarter for each forward period

See also

Using the information coefficient is an advanced topic. Here are some resources to get you started:

- Investopedia article describing the information coefficient: <https://www.investopedia.com/terms/i/information-coefficient.asp>
- Details of the Spearman rank correlation coefficient: https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

Examining factor return performance

Factor performance relates to the returns generated by a portfolio constructed explicitly based on specific factor values. These factors can encompass any measurable characteristic or set of characteristics about assets, such as value, momentum, size, or volatility, or in our case, mean reversion. In constructing a factor portfolio, assets with high factor values are held long, while those with low values are held short. This approach seeks to isolate the return attributable solely to the factor under consideration. The primary objective behind computing factor returns is to assess the performance of a factor and determine if it offers a premium over some benchmark return. To ensure that these returns are purely a result of the factor, various adjustments, such as demeaning or group adjustments, are applied.

On the other hand, portfolio returns represent the returns generated by a portfolio based on any chosen investment strategy, which might not necessarily be tied to specific factor values. Such portfolios can be constructed based on criteria like expected returns, risk, fundamental analysis, or technical indicators. The weightings of assets in these portfolios can vary, ranging from equal weighting to market-cap weighting or other schemes. The main aim of evaluating normal portfolio returns is to gauge the efficacy of the investment strategy, which can give us insights into the portfolio's performance relative to benchmarks or other investment alternatives. Unlike factor returns, normal portfolio returns provide a view of the portfolio's performance without the intent to isolate returns due to specific factors.

This recipe will demonstrate the available Alphalens Reloaded functionality to examine factor return performance.

How to do it...

We'll explore how to analyze the return data in the *How it works* section, next.

1. Import additional libraries from Alphalens Reloaded to assess the factor performance:

```
from alphalens.performance import (
    factor_returns,
    factor_cumulative_returns,
    mean_return_by_quantile,
    compute_mean_returns_spread,
```

```
        factor_alpha_beta,  
    )
```

2. Compute the period-wise, returns for the portfolio weighted by the factor values:

```
returns = factor_returns(alphalens_data)
```

The result is the following DataFrame with the portfolio returns for each forward return period:

		5D	10D	21D	63D
	date				
	2016-01-04 00:00:00+00:00	-0.016253	-0.027134	-0.013492	-0.015658
	2016-01-05 00:00:00+00:00	-0.016253	-0.027134	-0.013492	-0.015658
	2016-01-06 00:00:00+00:00	-0.016253	-0.027134	-0.013492	-0.015658
	2016-01-07 00:00:00+00:00	-0.016253	-0.027134	-0.026664	-0.015658
	2016-01-08 00:00:00+00:00	-0.016253	-0.026459	-0.026664	-0.011788

	2017-09-25 00:00:00+00:00	-0.003797	-0.001202	0.000069	0.000902
	2017-09-26 00:00:00+00:00	-0.003797	-0.001202	0.000069	-0.006470
	2017-09-27 00:00:00+00:00	-0.003797	-0.001202	0.000069	-0.006470
	2017-09-28 00:00:00+00:00	-0.003797	-0.001202	0.000069	-0.006470
	2017-09-29 00:00:00+00:00	-0.003797	-0.001202	0.009394	-0.006470

Figure 8.9: DataFrame with factor-weighted portfolio returns

3. Inspect the returns on a per-asset basis:

```
returns = factor_returns(alphalens_data, by_asset=True)
```

The result is the following MultiIndex DataFrame with each asset return weighted by the associated factor:

			5D	10D	21D	63D
	date	asset				
2016-01-04 00:00:00+00:00	AAL	-0.000006	0.000078	0.000058	-0.000064	
	AAPL	-0.001374	-0.001751	-0.001797	0.001162	
	ABBV	0.000862	0.000605	0.000744	-0.000370	
	AET	0.000211	0.000237	0.000316	0.000380	
	AGN	-0.000052	-0.000100	-0.000158	-0.000191	
...
2017-09-29 00:00:00+00:00	ADP	-0.000008	-0.000008	-0.000008	-0.000008	
	COL	0.000130	0.000285	0.000417	0.000417	
	BBY	0.000939	0.001407	0.001407	0.001407	
	EFX	0.000411	0.001081	0.000592	0.000444	
	SBAC	0.000143	0.000635	0.000635	0.000635	

Figure 8.10: MultiIndex DataFrame with per-asset, factor-weighted return

4. Visualize the factor-weighted, cumulative portfolio returns for the 5-day forward return against the equal-weighted portfolio:

```
pd.concat(
    {
        "factor_weighted": factor_cumulative_returns(
            alphalens_data,
            period="5D"
        ),
        "equal_weighted": factor_cumulative_returns(
            alphalens_data,
            period="5D",
            equal_weight=True
        ),
    },
    axis=1,
).plot()
```

The result is the following plot demonstrating the cumulative return of the portfolios:



Figure 8.11: Plot with the cumulative return of the 5-day forward, factor-weighted portfolio against the 5-day forward, equal-weighted portfolio

5. Compute the mean return for factor quantiles across the forward returns:

```
mean_returns, _ = mean_return_by_quantile(alphalens_data)
mean_returns.plot.bar()
```

6. The result is the following bar chart visualizing the mean return for each forward return in each quantile:

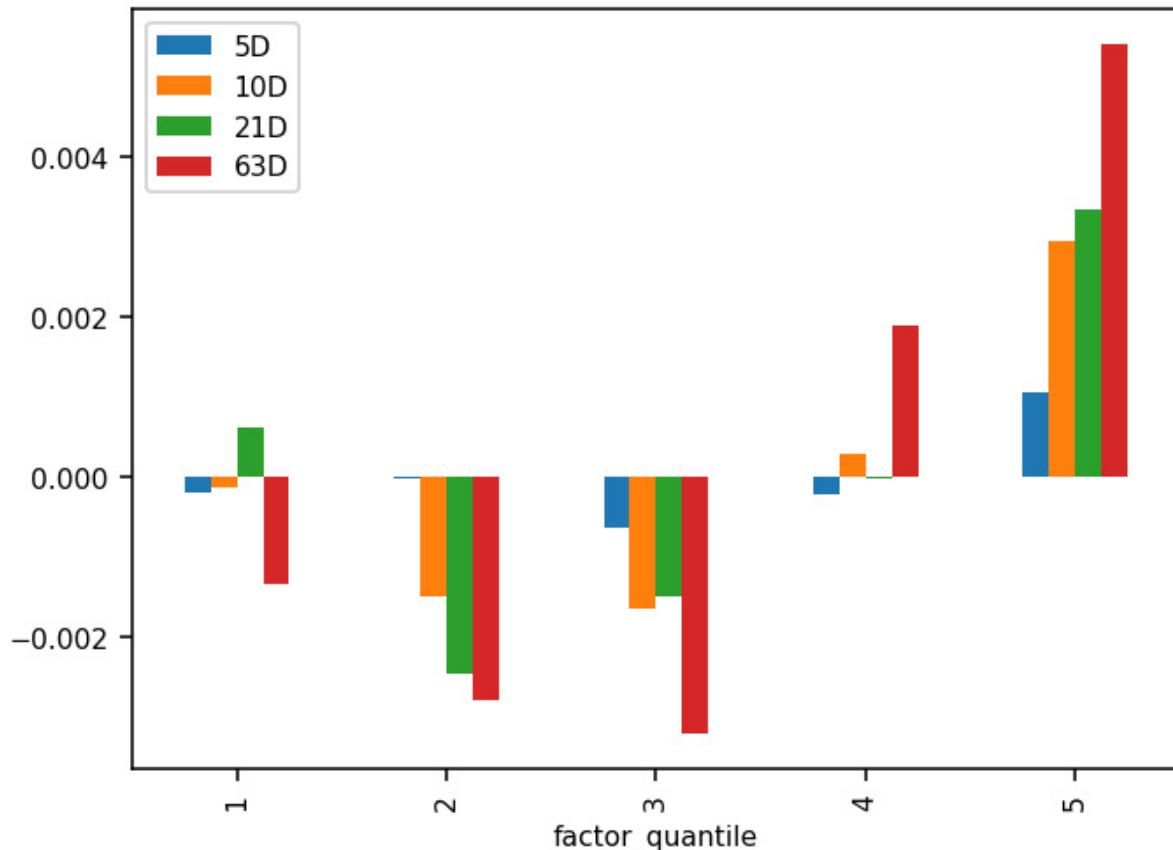


Figure 8.12: Bar chart with the mean factor-weighted portfolio return for each forward return in each quantile

7. Compute the difference in mean returns between the portfolios at the upper and lower quantiles:

```
mean_returns_by_date, _ = mean_return_by_quantile(
    alphalens_data,
    by_date=True
)
mean_return_difference, _ = compute_mean_returns_spread(
    mean_returns=mean_returns_by_date,
    upper_quant=1,
    lower_quant=5,
)
```

8. The result is a DataFrame with the difference between the mean return in the upper and lower quantiles. We can take it a step further and resample the daily values to monthly, take their mean, and plot them:

```
(  
    mean_return_difference[["5D"]]  
    .resample("M")  
    .mean()  
    .to_period("M")  
    .plot  
    .bar()  
)
```

The result is the following plot of the mean spread per month for the 5-day forward return:

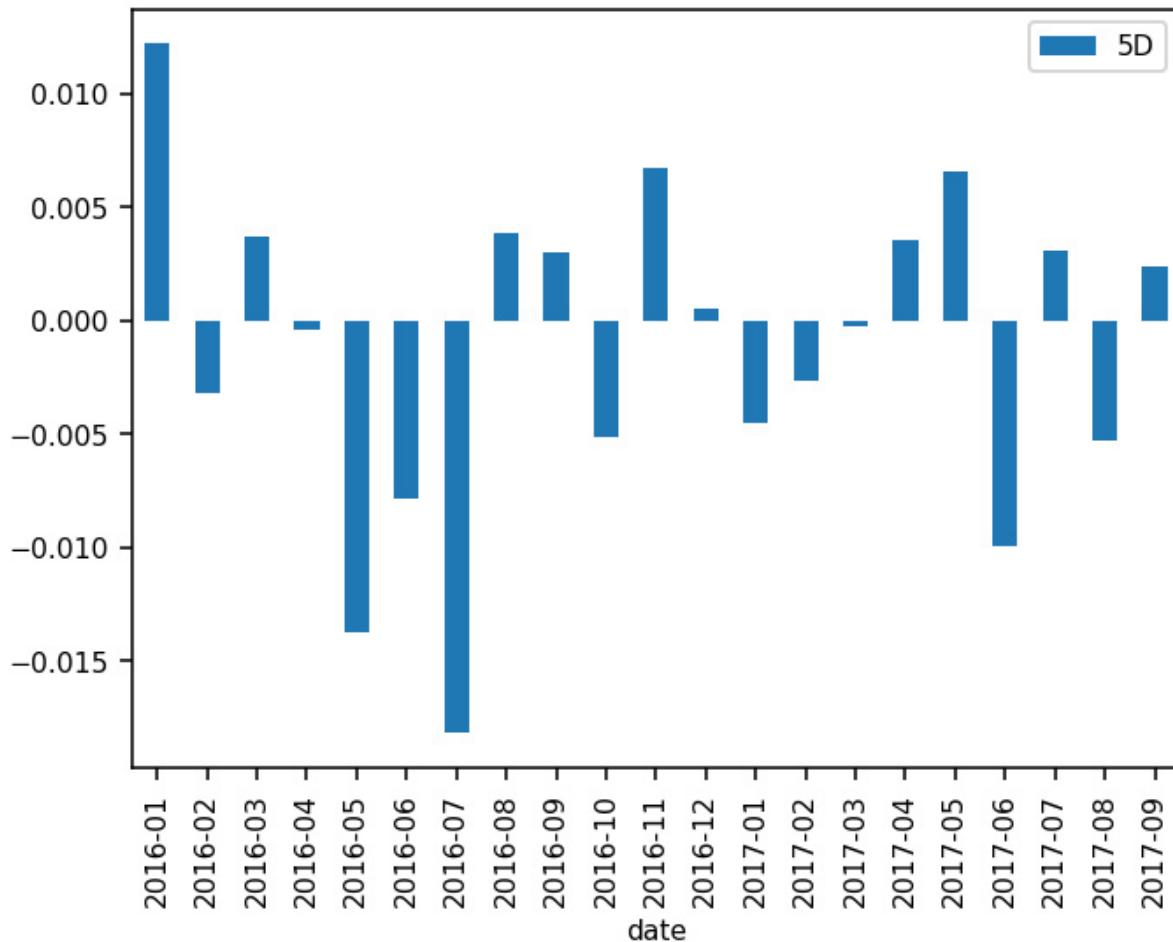


Figure 8.13: Bar chart demonstrating the mean, factor-weighted portfolio returns for the 5-day forward return

How it works...

The `factor_returns()` function is designed to compute the factor-weighted returns. It takes in two primary arguments: `factor_data`, which is a MultiIndex DataFrame that contains factor values, forward returns for each period, and quantiles and `periods`, which is a list of periods for which to compute the returns. For each specified period, it calculates the factor-weighted return by multiplying factor values with forward returns. We can optionally override the factor-weighting and assume an equal-weighted portfolio. This is helpful to compare performance.

The `factor_cumulative_returns()` function is designed to compute the cumulative returns of a factor. The `factor_data` argument is a multi-index DataFrame that contains factor values, forward returns, and group codes. The `period` argument specifies the period for which the cumulative returns are to be calculated, and the `demeaned` argument, when set to `True`, will demean the returns based on the group

codes provided in the `alphalens_data` DataFrame. The function computes the mean returns for each date and then determines the cumulative product of these mean returns. The result is a time-series of cumulative returns indexed by date.

The `mean_return_by_quantile()` function computes the mean returns for factors, grouped by quantiles. This is useful to understand how different quantiles of a factor perform over time. In the context of factor portfolio management, the spread between quantiles, which we look at next, is a way to measure the predictive power of the factor. The function offers flexibility through parameters like `by_date`, `demeaned`, `group_adjust`, and `by_group`, which determine how returns are computed and grouped. The core computation groups data by quantiles, and depending on the flags set, by date or group codes, then calculates the mean returns for each group. The result is a MultiIndex DataFrame with the mean returns for each quantile, and optionally, dates and quantiles.

The `compute_mean_returns_spread()` function is designed to compute the mean returns spread for quantiles over a specified period. The spread between the mean return in different quantiles provides insights into how well an alpha factor can differentiate between high and low performing assets over time or across different groups.

There's more...

Using Alphalens Reloaded we can calculate the factor's alpha (representing excess returns) and its beta (indicating market exposure). To do it, we call `factor_alpha_beta` which performs a regression using the mean return of the factor universe for each period as the independent variable and the average return from a portfolio, weighted by factor values, for each period as the dependent variable.

```
factor_alpha_beta(alphalens_data)
```

The result is a DataFrame with the annualized alpha and beta across each forward return:

	5D	10D	21D	63D
Ann. alpha	0.006204	0.015760	-0.003038	-0.012823
beta	0.136634	0.123627	0.117090	0.147144

Figure 8.14: DataFrame with the results of a regression on the mean return of the factor universe and average portfolio return.

As we learned in [Chapter 5, Build Alpha Factors for Stock Portfolios](#), alpha provides a measure of the factor's performance. A positive alpha indicates that the factor has outperformed the benchmark on a risk-adjusted basis, while a negative alpha suggests underperformance. Beta gives insight into the factor's risk relative to the benchmark. A beta greater than 1 indicates that the factor is more

volatile than the benchmark, while a beta less than 1 suggests it is less volatile. Over the forward returns of 5- and 10-days, our factor seems to generate positive alpha.

See also

Applied **Quantitative Research (AQR)** is a global investment management firm that employs a systematic, research-driven approach to capture investment opportunities across various asset classes. Founded in 1998 by Cliff Asness, David Kabiller, John Liew, and Robert Krail, AQR has become known for its focus on quantitative, data-driven strategies, combining academic research with practical market experience. AQR writes extensively about factor investing. Here's a great paper about measuring factor exposures: <https://www.aqr.com/-/media/AQR/Documents/Insights/Trade-Publications/Measuring-Portfolio-Factor-Exposures-A-Practical-Guide.pdf>

Evaluating factor turnover

In algorithmic trading, factor portfolios form the bedrock of many strategies. As we learned in the last recipe, assets are systematically ranked using the Spearman rank correlation from highest to lowest. Following this ranking, assets in the top quartile are bought and those from the bottom quartile are sold. In this context, turnover quantifies the frequency with which assets are bought or sold to rebalance the designated quartiles.

If a factor exhibits high turnover, it might suggest that the factor's signals are not persistent and could lead to higher trading costs if one were to trade based on this factor. Conversely, a factor with low turnover might indicate more stable signals. By analyzing turnover at the quantile level, we can gain insights into the stability of the factor's rankings across different segments of the asset universe. This can be particularly useful when assessing the viability of a factor for portfolio construction, as it provides a lens into the potential transaction costs and the reliability of the factor's signals.

This recipe will demonstrate the available Alphalens Reloaded functionality to examine turnover.

How to do it...

We'll use Alphalens Reloaded to compute the proportion of assets in a factor quantile that were not in that quantile in the previous period. This is a good way to measure aggregate changes in quartiles across the portfolio.

1. Import additional libraries from Alphalens Reloaded to assess the turnover:

```
from alphalens.performance import (
```

```

        quantile_turnover,
        factor_rank_autocorrelation
    )
from alphalens.plotting import plot_factor_rank_auto_correlation

```

2. Compute the portfolio turnover for the first quantile:

```
turnover = quantile_turnover(alphalens_data, quantile=1)
```

The result is the following Series containing the proportion of assets in the first quantile that were not in that quantile in the previous period:

date	
2016-01-04 00:00:00+00:00	NaN
2016-01-05 00:00:00+00:00	0.000000
2016-01-06 00:00:00+00:00	0.000000
2016-01-07 00:00:00+00:00	0.000000
2016-01-08 00:00:00+00:00	0.000000
	...
2017-09-25 00:00:00+00:00	0.030303
2017-09-26 00:00:00+00:00	0.000000
2017-09-27 00:00:00+00:00	0.000000
2017-09-28 00:00:00+00:00	0.000000
2017-09-29 00:00:00+00:00	0.000000
Freq: C, Name: 1, Length: 440, dtype: float64	

Figure 8.15: Series containing the proportion of assets in the first quantile that were not in that quantile in the previous period

3. Resample the daily values to monthly and plot the mean turnover for the first quantile:

```
turnover.resample("M").mean().to_period("M").plot.bar()
```

The result is the following bar chart with the mean monthly turnover for the first quantile:

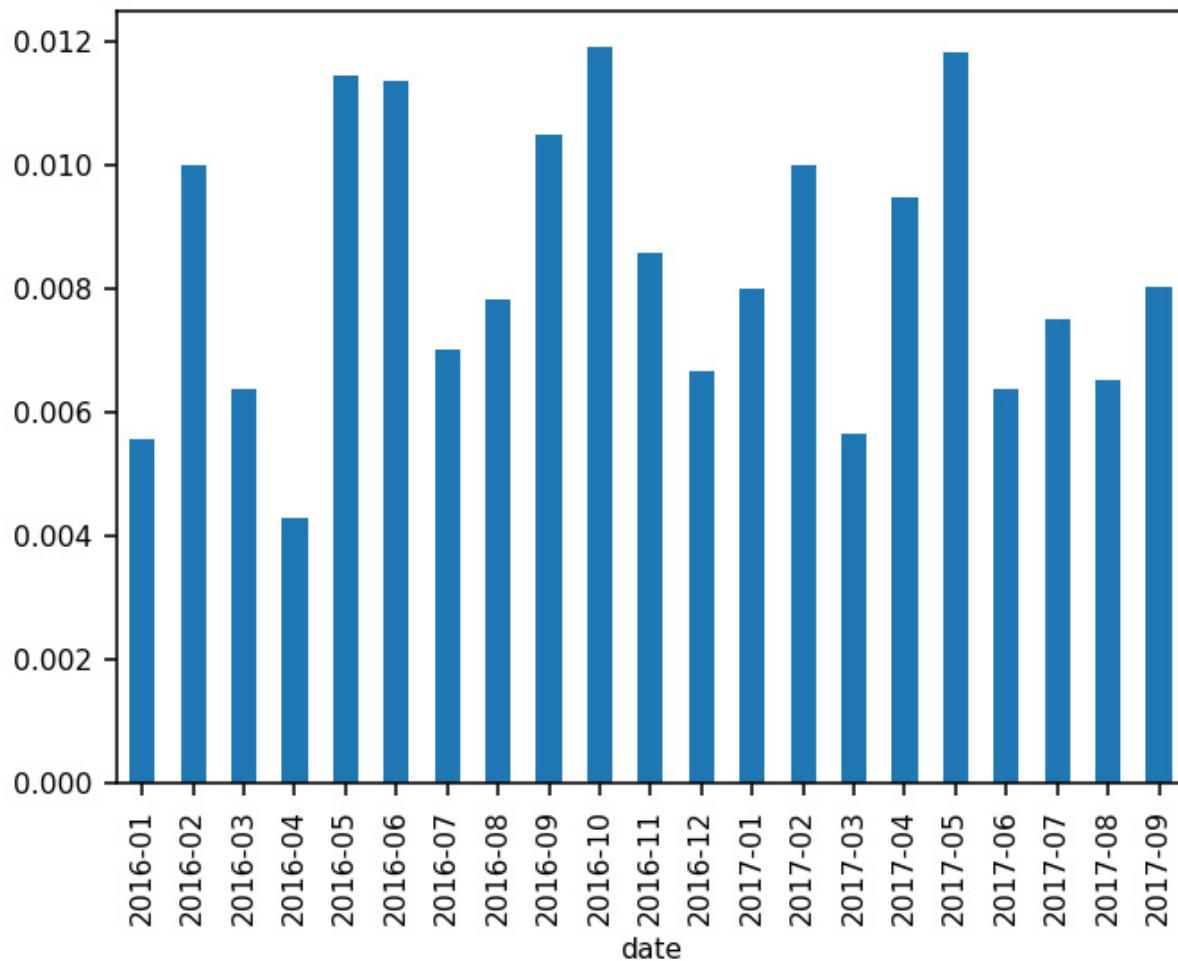


Figure 8.16: Bar chart of the mean monthly turnover for the first quantile

How it works...

The `quantile_turnover()` function calculates the turnover for factor quantiles over a specified period. Turnover, in the context of factor portfolio analysis, refers to the rate at which assets within a portfolio are replaced by new assets. High turnover can indicate frequent trading, which might lead to higher transaction costs, while low turnover suggests a more buy-and-hold strategy. By analyzing turnover in the context of factor quantiles, one can assess how stable the factor rankings are over time.

The function takes in the following parameters:

- `alphalens_data`: A multi-index DataFrame that contains factor values, forward returns for each period, the factor quantile/bin that factor value belongs to, and (optionally) the group the asset belongs to.

- **quantiles**: An integer or sequence of integers. If an integer, it will select the top and bottom **quantiles** from the factor values. If a sequence, it will define the exact quantiles to compute.
- **period**: The period over which to calculate the turnover. It defaults to ‘1D’, meaning daily turnover.

The function begins by filtering the factor data for the specified quantiles. It then groups the data by date and quantile, and for each group, it calculates the set difference between the current and previous day’s assets. This set difference represents the assets that have entered or left the quantile on that day. The function then computes the average turnover for each quantile over the specified period.

There's more...

Another common turnover analysis technique is using the autocorrelation of the factor’s daily Spearman rank correlations. Factor rank autocorrelation is a crucial metric in factor portfolio analysis for several reasons:

- **Stability of Factor Ranks**: The autocorrelation of factor ranks provides insights into the stability of the factor’s rankings over time. A high autocorrelation indicates that the factor’s rankings are relatively stable from one day to the next. This stability can be beneficial for strategies that rely on the persistence of factor rankings.
- **Turnover Implications**: Factors with high rank autocorrelation tend to have lower portfolio turnover. This is because if the ranks of assets based on a factor do not change significantly from one period to the next, the positions in a portfolio constructed based on those ranks would also remain relatively stable. Lower turnover can lead to reduced transaction costs, which can enhance the net returns of a strategy.
- **Factor Consistency**: A consistent factor, one that maintains its rank across assets over time, is generally preferred in quantitative strategies. High rank autocorrelation can be an indicator of such consistency. It suggests that the factor is not frequently changing its opinion about the relative attractiveness of different assets.
- **Risk Management**: Understanding the autocorrelation of factor ranks can also aid in risk management. If a factor’s ranks are highly volatile (low autocorrelation), it might introduce additional risks in a portfolio strategy. Being aware of this can help in designing appropriate risk controls.

We can measure this with the `factor_rank_autocorrelation()` function. The `factor_rank_autocorrelation()` function computes the autocorrelation of the provided factor’s daily ranks. Specifically, it calculates the Spearman rank correlation between the factor values of a particular day and those of the subsequent day. The function takes in the following parameters:

- **alphalens_data**: A multi-index DataFrame that contains factor values, forward returns for each period, and the factor quantile/bin that factor values belong to. The indices are date and asset.
- **period**: The period over which the autocorrelation is computed. It’s the lag between the factor values being compared.

```
factor_ac = factor_rank_autocorrelation(alphalens_data)
```

The result is a Series with the daily autocorrelation.

```

date
2016-01-04 00:00:00+00:00      NaN
2016-01-05 00:00:00+00:00      1.000000
2016-01-06 00:00:00+00:00      1.000000
2016-01-07 00:00:00+00:00      1.000000
2016-01-08 00:00:00+00:00      1.000000
...
2017-09-25 00:00:00+00:00      0.787847
2017-09-26 00:00:00+00:00      1.000000
2017-09-27 00:00:00+00:00      1.000000
2017-09-28 00:00:00+00:00      1.000000
2017-09-29 00:00:00+00:00      1.000000
Freq: C, Name: 1, Length: 440, dtype: float64

```

Figure 8.17: Series with the autocorrelation of the factor's Spearman rank

IMPORTANT

Note in our strategy rebalances on a weekly basis. This means that since there is no opportunity for rebalancing, there will be no turnover except on the weekly rebalance days. We see this reflected in the autocorrelation of 1.

We can use the `plot_factor_rank_auto_correlation()` function to visualize how the autocorrelations change through time.

```
plot_factor_rank_auto_correlation(factor_ac)
```

The result is a time series plot of the autocorrelation.

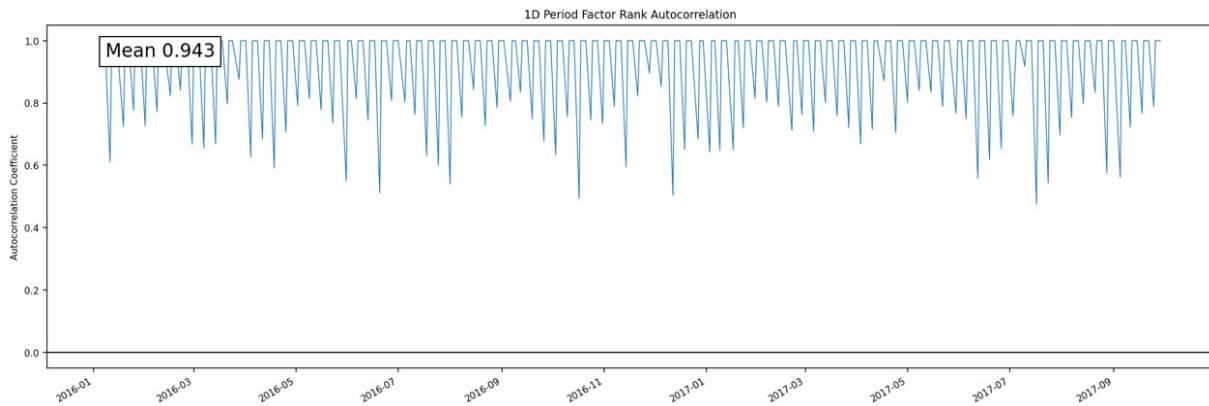


Figure 8.18: Plot visualizing the daily autocorrelation

See also

In case you want to deep dive into the code that computes these metrics, you can do so by visiting the following links:

- Source code for the **quantile_turnover** function: <https://github.com/stefan-jansen/alphalens-reloaded/blob/d4490ba1290f1f135ed398d1b3601569e0e7996b/src/alphalens/performance.py#L575>
- Source code for the **factor_rank_autocorrelation** function: <https://github.com/stefan-jansen/alphalens-reloaded/blob/d4490ba1290f1f135ed398d1b3601569e0e7996b/src/alphalens/performance.py#L620>

Assess Backtest Risk and Performance Metrics with Pyfolio

No single risk or performance metric tells the entire story of how a strategy might perform in live trading. Metrics such as the Sharpe ratio, for instance, focus mainly on returns relative to volatility but neglect other risks such as drawdown or tail risk. Similarly, using only maximum drawdown as a measure ignores the risk-adjusted returns and might discard strategies that are robust but temporarily underperforming. The composite view obtained through multiple metrics provides a more nuanced understanding of how the strategy is likely to behave under varying market conditions. Taking it a step further, visualizing risk and performance metrics over time can capture strategy dynamics over time. A strategy might exhibit robust metrics during a bull market but underperform in terms of risk-adjusted returns during a bear or sideways market.

In this chapter, we introduce **Pyfolio Reloaded (Pyfolio)**, which is a risk and performance analysis library. Pyfolio Reloaded is part of the Zipline Reloaded ecosystem and takes the output of a Zipline Reloaded backtest to build a robust suite of risk and performance metrics. We will walk through the process of using Pyfolio Reloaded to generate risk and performance metrics. Throughout the recipes in this chapter, we'll define the most important metrics to consider when assessing the performance of a backtest.

In this chapter, we present the following recipes:

- Preparing Zipline Reloaded backtest results for Pyfolio Reloaded
- Generating strategy performance and return analytics
- Building a drawdown and rolling risk analysis
- Analyzing strategy holdings, leverage, exposure, and sector allocations
- Breaking down strategy performance to trade level

Preparing Zipline backtest results for Pyfolio Reloaded

In [Chapter 7](#), *Event-Based Backtesting Factor Portfolios with Zipline Reloaded*, we learned how to use Zipline Reloaded to backtest a factor strategy. The output of a Zipline Reloaded backtest includes a DataFrame that details various metrics calculated over the backtest period, such as returns, alpha, beta, the Sharpe ratio, and drawdowns. It also provides transaction logs that capture executed orders, including asset, price, and quantity, giving insights into the trading behavior of the strategy.

Additionally, Zipline Reloaded outputs an asset-wise breakdown of the portfolio, detailing the holdings and their respective values, which can be vital for risk assessment and position sizing in the portfolio.

Before we can use the DataFrame, there is some required data preprocessing. Helpfully, Pyfolio Reloaded comes with helper functions that do most of the work for us. In this recipe, we'll read in the DataFrame and prepare it to use with Pyfolio. We'll also build a symbol-to-sector mapping and acquire data to represent a benchmark.

Getting ready...

We assume you ran the backtest described in [Chapter 7, Event-Based Backtesting Factor Portfolios with Zipline Reloaded](#), and have a file called `mean_reversion.pickle` in the current working directory. The pickle file is a serialized version of the DataFrame Zipline Reloaded generated describing the performance results.

How to do it...

We'll use pandas to read the pickle file and the OpenBB Platform to acquire sector data and benchmark prices.

1. Import the libraries:

```
import pandas as pd
from openbb import obb
import pyfolio as pf
obb.user.preferences.output_type = "dataframe"
```

2. Read in the pickle file using the pandas `read_pickle` method:

```
perf = pd.read_pickle("mean_reversion.pickle")
```

The result is the performance DataFrame from the Zipline backtest.

period_open	period_close	starting_value	ending_value	starting_cash	...	benchmark_period_return	benchmark_volatility	alpha	beta	sharpe
2016-01-04 21:00:00+00:00	2016-01-04 14:31:00+00:00	2016-01-04 21:00:00+00:00	0.00	0.00	100000.00000	...	0.000000	NaN	NaN	NaN
2016-01-05 21:00:00+00:00	2016-01-05 14:31:00+00:00	2016-01-05 21:00:00+00:00	0.00	-1977.80	100000.00000	...	0.002012	0.022588	0.000000	-0.000082
2016-01-06 21:00:00+00:00	2016-01-06 14:31:00+00:00	2016-01-06 21:00:00+00:00	-1977.80	-1986.16	101977.78350	...	-0.011130	0.130408	-0.001603	0.005824
2016-01-07 21:00:00+00:00	2016-01-07 14:31:00+00:00	2016-01-07 21:00:00+00:00	-1986.16	-1983.30	101977.78350	...	-0.034566	0.191144	-0.003779	-0.000142
2016-01-08 21:00:00+00:00	2016-01-08 14:31:00+00:00	2016-01-08 21:00:00+00:00	-1983.30	-1986.38	101977.78350	...	-0.045030	0.166229	-0.004494	-0.000074
...
2017-12-22 21:00:00+00:00	2017-12-22 14:31:00+00:00	2017-12-22 21:00:00+00:00	5747.17	5763.90	95142.41300	...	0.333231	0.103667	0.001123	0.023565
2017-12-26 21:00:00+00:00	2017-12-26 14:31:00+00:00	2017-12-26 21:00:00+00:00	5763.90	5696.27	95142.41300	...	0.331820	0.103569	0.000788	0.023615
2017-12-27 21:00:00+00:00	2017-12-27 14:31:00+00:00	2017-12-27 21:00:00+00:00	5696.27	64.69	95142.41300	...	0.332873	0.103466	0.001109	0.023621
2017-12-28 21:00:00+00:00	2017-12-28 14:31:00+00:00	2017-12-28 21:00:00+00:00	64.69	90.28	100840.72425	...	0.335317	0.103366	0.001211	0.023633
2017-12-29 21:00:00+00:00	2017-12-29 14:31:00+00:00	2017-12-29 21:00:00+00:00	90.28	76.95	100840.72425	...	0.328396	0.103344	0.001203	0.023637

Figure 9.1: Deserialized DataFrame containing the backtest performance metrics

3. Use the Pyfolio helper function to extract returns, positions, and transactions from the DataFrame:

```
returns, positions, transactions = \
    pf.utils.extract_rrets_pos_txn_from_zipline(perf)
```

The result is a pandas Series with strategy returns.

```
2016-01-04 00:00:00+00:00      0.000000e+00
2016-01-05 00:00:00+00:00     -1.650000e-07
2016-01-06 00:00:00+00:00    -8.360001e-05
2016-01-07 00:00:00+00:00     2.860240e-05
2016-01-08 00:00:00+00:00    -3.080170e-05
...
2017-12-22 00:00:00+00:00     1.658249e-04
2017-12-26 00:00:00+00:00    -6.702257e-04
2017-12-27 00:00:00+00:00     6.617624e-04
2017-12-28 00:00:00+00:00     2.536038e-04
2017-12-29 00:00:00+00:00    -1.320704e-04
Name: returns, Length: 503, dtype: float64
```

Figure 9.2: pandas Series with daily strategy returns

4. The **positions** DataFrames contain the Zipline **Equity** objects as column labels. Replace the object with the string representations:

```
positions.columns = [
    col.symbol for col in positions.columns[
        :-1]] + ["cash"]
```

The result is DataFrame with the daily positions, including cash.

	AAL	AAPL	ABBV	ADBE	ADP	...	WDC	WFC	WFM	WMT		cash
index												
2016-01-05 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	...	0.00	0.00	0.0	0.0	101977.78350	
2016-01-06 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	...	0.00	0.00	0.0	0.0	101977.78350	
2016-01-07 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	...	0.00	0.00	0.0	0.0	101977.78350	
2016-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	...	0.00	0.00	0.0	0.0	101977.78350	
2016-01-11 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	...	0.00	0.00	0.0	0.0	101977.78350	
...
2017-12-22 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	...	1936.32	0.00	0.0	0.0	95142.41300	
2017-12-26 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	...	1920.00	0.00	0.0	0.0	95142.41300	
2017-12-27 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	...	0.00	-1950.40	0.0	0.0	100840.72425	
2017-12-28 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	...	0.00	-1961.60	0.0	0.0	100840.72425	
2017-12-29 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	...	0.00	-1941.44	0.0	0.0	100840.72425	

Figure 9.3: DataFrame with the position value each day

5. The **symbol** column in the **transactions** DataFrame also contains the Zipline **Equity** objects. Replace the object with the string representations:

```
transactions.symbol = transactions.symbol.apply(
    lambda s: s.symbol)
```

The result is a DataFrame with the daily transactions:

		sid	symbol	price		order_id	amount	commission		dt	txn_dollars
2016-01-05	21:00:00+00:00	Equity(1228 [GMCR])	GMCR	89.90	6c0bfa6d54d1441baed02c6f0607a864	-22	None	2016-01-05 21:00:00+00:00	1977.80		
2016-01-12	21:00:00+00:00	Equity(1228 [GMCR])	GMCR	90.42	17907b8052f64fe3a5bc66e9ed8b09fd	22	None	2016-01-12 21:00:00+00:00	-1989.24		
2016-01-12	21:00:00+00:00	Equity(3105 [WMT])	WMT	63.62	23ddc217cfdd4a94b491013377386d8f	-31	None	2016-01-12 21:00:00+00:00	1972.22		
2016-01-20	21:00:00+00:00	Equity(3105 [WMT])	WMT	60.84	69aed3b9780e4f5abbd7ecd1ddde5527	31	None	2016-01-20 21:00:00+00:00	-1886.04		
2016-01-20	21:00:00+00:00	Equity(994 [ESRX])	ESRX	71.70	6d826744a2df47f5991903b16516fe45	27	None	2016-01-20 21:00:00+00:00	-1935.90		
...
2017-12-27	21:00:00+00:00	Equity(2331 [PYPL])	PYPL	74.59	9d1d861c73eb4a87ad9880c6ec940a82	27	None	2017-12-27 21:00:00+00:00	-2013.93		
2017-12-27	21:00:00+00:00	Equity(606 [CMCSA])	CMCSA	40.41	2e36bcfdf6484592bf63b14ad9942aa3	-49	None	2017-12-27 21:00:00+00:00	1980.09		
2017-12-27	21:00:00+00:00	Equity(1063 [FDX])	FDX	250.03	ba5adcf1fd4f40db95eebd864b9aead5	-8	None	2017-12-27 21:00:00+00:00	2000.24		
2017-12-27	21:00:00+00:00	Equity(2945 [UNP])	UNP	136.32	7a9cda6bb33a4f3b9ec3ed3e718c4a0b	-14	None	2017-12-27 21:00:00+00:00	1908.48		
2017-12-27	21:00:00+00:00	Equity(3077 [WFC])	WFC	60.95	2ef1d59050db4dd4bc144e9891323cdd	-32	None	2017-12-27 21:00:00+00:00	1950.40		

Figure 9.4: DataFrame with the transactions each day

6. Extract the symbols from the **positions** DataFrame and use the OpenBB Platform screener to download an overview for each one. The overview includes the sector that we'll use to build the symbol-to-sector mapping:

```
symbols = positions.columns[:-1].tolist()
screener_data = obb.equity.profile(
    symbols, provider="yfinance")
```

The result is a DataFrame with summary information for each symbol.

	symbol	name	stock_exchange	long_description	company_url	shares_float	shares_implied_outstanding	shares_short	dividend_yield	beta
0	AAL	American Airlines Group Inc.	NMS	American Airlines Group Inc., through its subs...	https://www.aa.com	6.471347e+08	6.661270e+08	53327236.0	NaN	1.580
1	AAPL	Apple Inc.	NMS	Apple Inc. designs, manufactures, and markets ...	https://www.apple.com	1.530832e+10	1.566390e+10	99287450.0	0.0052	1.264
2	ABBV	AbbVie Inc.	NYQ	AbbVie Inc. discovers, develops, manufactures...	https://www.abbvie.com	1.762248e+09	1.765870e+09	15927790.0	0.0397	0.593
3	ADBE	Adobe Inc.	NMS	Adobe Inc., together with its subsidiaries, op...	https://www.adobe.com	4.467590e+08	4.801530e+08	8007830.0	NaN	1.281
4	ADP	Automatic Data Processing, Inc.	NMS	Automatic Data Processing, Inc. provides cloud...	https://www.adp.com	4.083909e+08	4.092910e+08	4142715.0	0.0232	0.785
...
128	WBA	Walgreens Boots Alliance, Inc.	NMS	Walgreens Boots Alliance, Inc. operates as a h...	https://www.walgreensbootsalliance.com	7.131361e+08	8.627130e+08	43268583.0	0.0650	0.793
129	WDC	Western Digital Corporation	NMS	Western Digital Corporation develops, manufact...	https://www.westerndigital.com	3.249116e+08	3.265250e+08	21805777.0	NaN	1.521
130	WFC	Wells Fargo & Company	NYQ	Wells Fargo & Company, a financial services co...	https://www.wellsfargo.com	3.478506e+09	3.486320e+09	38130739.0	0.0236	1.182
131	WFM	None	YHD	None	None	NaN	NaN	NaN	NaN	NaN
132	WMT	Walmart Inc.	NYQ	Walmart Inc. engages in the operation of retail...	https://corporate.walmart.com	4.332595e+09	8.063520e+08	43289108.0	0.0128	0.494

Figure 9.5: DataFrame with the screener results for the tickers used in our strategy

7. Build a mapping between each **symbol** and **sector**:

```
sector_map = (
    screener_data[["symbol", "sector"]]
    .set_index("symbol")
    .reindex(symbols)
    .fillna("Unknown")
    .to_dict()["sector"]
)
```

The result is a dictionary with symbols as keys and the sector as values. Note for symbols that don't have an associated sector, they're marked as `Unknown`.

```
{'AAL': 'Industrials',
 'AAPL': 'Technology',
 'ABBV': 'Healthcare',
 'ADBE': 'Technology',
 'ADP': 'Industrials',
 'AET': 'Unknown',
 'AGN': 'Unknown',
 'AIG': 'Financial',
 'ALXN': 'Unknown',
 'AMAT': 'Technology',
 'AMGN': 'Healthcare',
 'AMZN': 'Consumer Cyclical',
 'ANTM': 'Unknown',
 'ARIA': 'Unknown',
 'ATVI': 'Communication Services',
 'AVGO': 'Technology',
 'AXP': 'Financial',
 'AZO': 'Consumer Cyclical',
 'BA': 'Industrials',
 'BBY': 'Consumer Cyclical',
 'BCR': 'Unknown',
 'BIDU': 'Communication Services',
 'BIIB': 'Healthcare',
 'BMY': 'Healthcare',
 'BRK_B': 'Unknown',
 'CCE': 'Unknown',
 'CELG': 'Unknown',
 'CHTR': 'Communication Services',
```

Figure 9.6: Dictionary with a symbol-to-sector mapping for the positions in our backtest

8. Use the OpenBB Platform to download price data for the SPY ETF, which we'll use as the benchmark:

```
spy = obb.equity.price.historical(
    "SPY",
    start_date=returns.index.min(),
    end_date=returns.index.max()
)
spy.index = pd.to_datetime(spy.index)
benchmark_returns = spy.close.pct_change()
benchmark_returns.name = "SPY"
benchmark_returns = benchmark_returns.tz_localize(
    "UTC").filter(returns.index)
```

The result is a pandas Series with daily returns for the SPY ETF.

```

date
2016-01-04 00:00:00+00:00      NaN
2016-01-05 00:00:00+00:00   0.001691
2016-01-06 00:00:00+00:00  -0.012614
2016-01-07 00:00:00+00:00  -0.023992
2016-01-08 00:00:00+00:00  -0.010976
...
2017-12-22 00:00:00+00:00  -0.000262
2017-12-26 00:00:00+00:00  -0.001196
2017-12-27 00:00:00+00:00   0.000486
2017-12-28 00:00:00+00:00   0.002058
2017-12-29 00:00:00+00:00  -0.003770
Name: SPY, Length: 503, dtype: float64

```

Figure 9.7: pandas Series with the daily returns for the SPY ETF

How it works...

The `extract_rets_pos_txn_from_zipline` function takes a DataFrame generated by a Zipline backtest as input and extracts daily returns, net position values, and transaction details. These extracted metrics are returned as a tuple of pandas Series and DataFrames, ready for further analysis or visualization with other Pyfolio functions. The function first normalizes the index of the input DataFrame and sets its time zone to UTC, then extracts the `returns` column for daily strategy returns. It iterates through the `positions` and `transactions` items in the DataFrame to construct daily net position values and transaction details, respectively.

Next, we use a list comprehension to replace each of the column labels in the `positions` DataFrame with strings. We use the pandas `apply` method on the `symbol` column in the `transactions` DataFrame to replace the Zipline Reloaded `Equity` objects with their string representations.

After we extract a list of symbols and download the overview using the OpenBB Platform, we build the symbol-to-sector mapping. We start with the OpenBB Platform output that contains columns `symbol` and `sector`, and sets the index to the `symbol` column. The DataFrame is then reindexed based on the list of `symbols`, fills any missing values with `unknown`, and finally converts the `sector` column to a dictionary.

Finally, we download and process price data for the SPY ETF which we use as the benchmark in our analysis. The code loads historical data for the dates in our analysis using the `obb.equity.price.historical` method. It then calculates the percentage change of the adjusted closing prices to generate benchmark returns, sets the name of the Series to `spy`, localizes the time zone to UTC, and filters the Series to match the index of the `returns` DataFrame.

There's more...

You may be wondering why mapping symbols to sectors is important. Incorporating sector information in backtest analysis is crucial for understanding the source of returns and for risk management. Traders often attribute returns to different sectors to identify which segments of the market are driving their portfolio's performance.

This sector-based attribution enables traders to diversify their investments across various sectors, thereby reducing the portfolio's systemic risks associated with any single sector.

Further, by comparing the strategy's returns against a benchmark, like the S&P 500, traders can assess whether the strategy is adding value over and above a passive investment approach. This comparative analysis aids in isolating the strategy's alpha, or risk-adjusted returns, and helps in understanding its behavior relative to the broader market or a specific sector.

See also

Pyfolio offers several helper functions to process Zipline backtest results into a form suitable for further analysis. For more insight into using Pyfolio for risk and performance analysis, see the source code at <https://github.com/stefan-jansen/pyfolio-reloaded/blob/main/src/pyfolio/utils.py>.

In case you want to automate the process of computing risk and performance metrics, **Trade Blotter** is an app that makes performance analytics easy. You can upload your transactions and it does the heavy lifting for you. You can sign up at <https://tradeblotter.io/>. You can also check out the cohort-based course, *Getting Started With Python for Quant Finance*, which covers Pyfolio in detail. The URL is <https://www.pyquantnews.com/getting-started-with-python-for-quant-finance>.

Generating strategy performance and return analytics

Traders use strategy performance and return analysis to evaluate the effectiveness of their trading algorithms. Return analysis, often visualized through equity curves, or return distributions, offers insights into the strategy's profitability over time. Temporal analyses, such as monthly or annual return breakdowns, help identify seasonality or long-term trends that may impact future performance.

By comparing these metrics and analyses against a benchmark, traders can isolate the strategy's alpha, or the excess return over a passive investment approach. This review enables traders to make data-driven modifications to their strategies, enhancing profitability and risk management. In this recipe, we explore Pyfolio Reloaded strategy performance and return analytics.

Getting ready...

We assume the steps in the *Preparing Zipline Reloaded backtest results for Pyfolio Reloaded* recipe were followed. We'll need `returns`, `positions`, `transactions`, and `benchmark_returns` defined for this recipe.

How to do it...

We'll build a series of visualizations using Pyfolio Reloaded that graphically depict strategy performance.

1. Plot the strategy's equity curve against the benchmark:

```
pf.plotting.plot_rolling_returns(  
    returns,  
    factor_returns=benchmark_returns  
)
```

The result is a chart with the strategy's equity curve alongside the cumulative returns of the chosen benchmark.

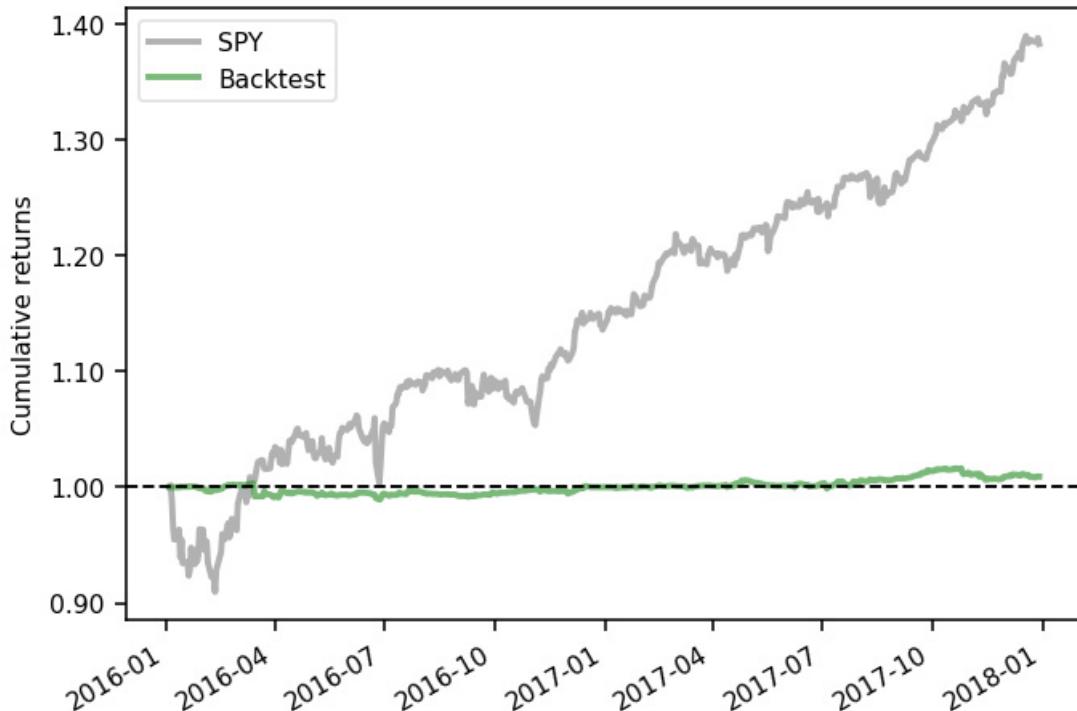


Figure 9.8: Strategy cumulative returns (equity curve) against the benchmark

2. Summarize the distribution of key performance indicators:

```
pf.plotting.plot_perf_stats()
```

```

        returns=returns,
        factor_returns=benchmark_returns,
    )

```

The result is a horizontal box plot that depicts the distribution of performance indicators.

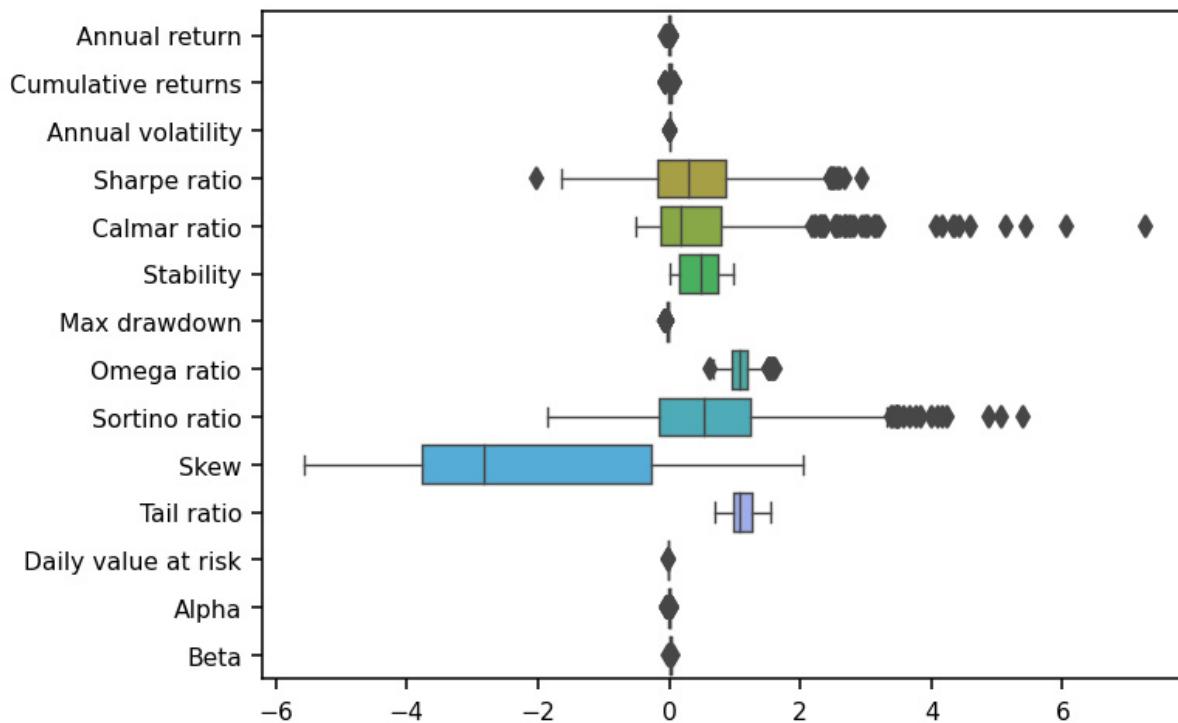


Figure 9.9: Strategy performance metrics

3. Generate a detailed outline of the strategy's performance metrics:

```

pf.plotting.show_perf_stats(
    returns,
    factor_returns=benchmark_returns,
    positions=positions,
    transactions=transactions,
    live_start_date="2017-01-01"
)

```

IMPORTANT NOTE

The `live_start_date` argument in the `show_perf_stats` function separates the backtest and live trading periods. Metrics are calculated separately for the periods before and after this date, which allows for a more nuanced evaluation of the trading strategy's performance. Specifically, the function will display performance statistics for the backtest period up to `live_start_date` and for the live trading period starting from `live_start_date`. Several Pyfolio functions accept this argument.

The result is a pandas DataFrame containing the key performance metrics.

Start date	2016-01-04		
End date	2017-12-29		
In-sample months	12		
Out-of-sample months	11		
	In-sample	Out-of-sample	All
Annual return	-0.005%	0.926%	0.459%
Cumulative returns	-0.005%	0.922%	0.918%
Annual volatility	1.711%	1.302%	1.519%
Sharpe ratio	0.01	0.71	0.31
Calmar ratio	-0.00	0.95	0.33
Stability	0.03	0.69	0.63
Max drawdown	-1.374%	-0.98%	-1.374%
Omega ratio	1.00	1.15	1.07
Sortino ratio	0.01	0.97	0.40
Skew	-3.62	-1.03	-2.92
Kurtosis	42.22	7.57	36.28
Tail ratio	0.98	1.28	1.04
Daily value at risk	-0.215%	-0.16%	-0.19%
Gross leverage	0.06	0.11	0.09
Daily turnover	23.295%	25.598%	24.355%
Alpha	-0.00	0.01	0.00
Beta	0.03	0.01	0.02

Figure 9.10: Strategy performance metrics

4. Generate a heatmap of the strategy's monthly returns:

```
pf.plotting.plot_monthly_returns_heatmap(returns)
```

The result is a heatmap visualizing the strategy returns during the backtest period.

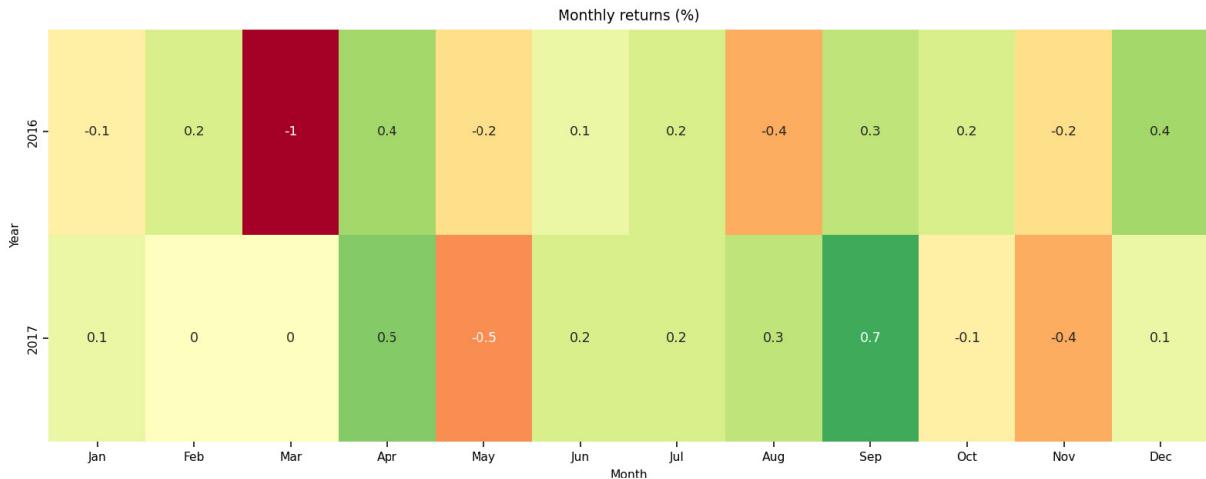


Figure 9.11: Heatmap visualizing the strategy's monthly returns

5. Generate a bar chart of the strategy's annual returns:

```
pf.plotting.plot_annual_returns(returns)
```

The result is a bar chart visualizing the annual returns during the backtest period.

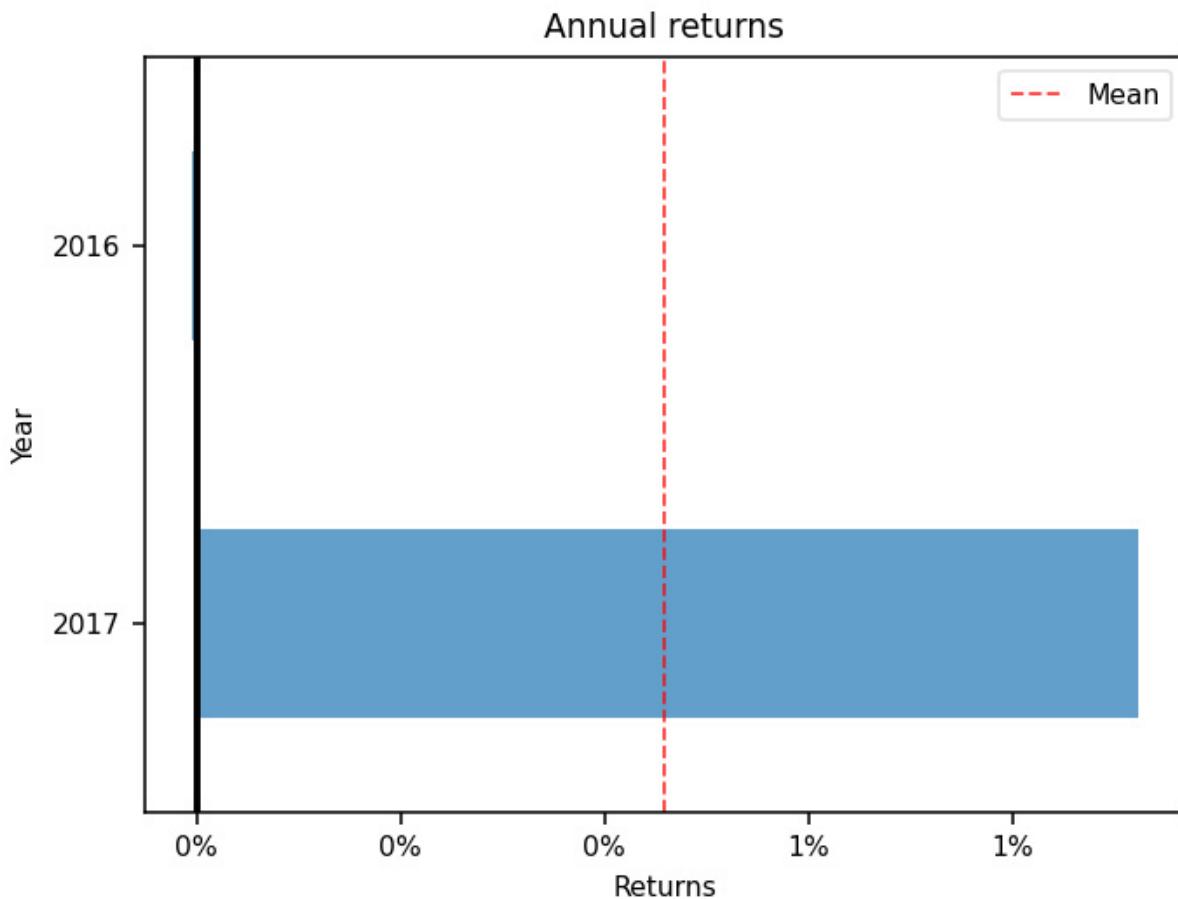


Figure 9.12: Bar chart visualizing the strategy's annual returns

How it works...

Pyfolio does the hard work of parsing the input data and formatting the output charts. Other charts and tables include the following:

- **plot_rolling_returns**: Calculates the rolling returns of a portfolio over a specified window and plots them using Matplotlib.
- **plot_perf_stats**: Iterates through the returns series, extracting each performance metric, and uses Matplotlib's **barch** function to create horizontal bars for each metric.
- **show_perf_stats**: Calculates various performance metrics including annual return, annual volatility, and Sharpe ratio, among others, and then displays these metrics in a formatted table.
- **plot_monthly_returns_heatmap**: Calculates the mean return for each month across years. It then utilizes Matplotlib to generate a heatmap, where the x-axis represents months, the y-axis represents years, and the color intensity indicates the mean return value.
- **plot_annual_returns**: Calculates annual returns by resampling the data to yearly frequency using the mean. It then generates a bar plot of these annual returns using Matplotlib, with the x-axis representing years and the y-axis representing the annual returns.

Now that we covered how the plots are generated, let's review some of the key statistics output by Pyfolio:

- **Calmar ratio**: Calmar ratio is calculated by dividing the **compound annual growth rate (CAGR)** of a trading strategy by the maximum drawdown experienced over a specified period
- **Omega ratio**: The omega ratio is calculated by dividing the sum of positive excess returns by the absolute sum of negative excess returns over a given threshold
- **Skew**: Skew is calculated by taking the third standardized moment of the return series, essentially measuring the asymmetry of the return distribution
- **Kurtosis**: Kurtosis is calculated by taking the fourth central moment of the returns series and dividing it by the square of the variance, effectively measuring the “tailedness” of the distribution
- **Value at risk**: Daily **value at risk (VaR)** is calculated by taking the negative of the quantile of the daily returns at a given confidence level, typically 5% or 1%
- **Gross leverage**: Gross leverage is calculated as the sum of the absolute values of long and short positions divided by the portfolio's net asset value at each time point

There's more...

Pyfolio offers more details about a strategy's returns, which can provide more insight into how the strategy performed during the backtest.

1. Create a distribution of monthly returns:

```
pf.plotting.plot_monthly_returns_dist(returns)
```

The output is a histogram with the frequency of monthly returns.



Figure 9.13: Histogram with the frequency of monthly returns

2. Visualize the strategy's daily returns through time:

```
pf.plotting.plot_returns(  
    returns,  
    live_start_date="2017-01-01"  
)
```

The result is a line plot depicting the daily returns.

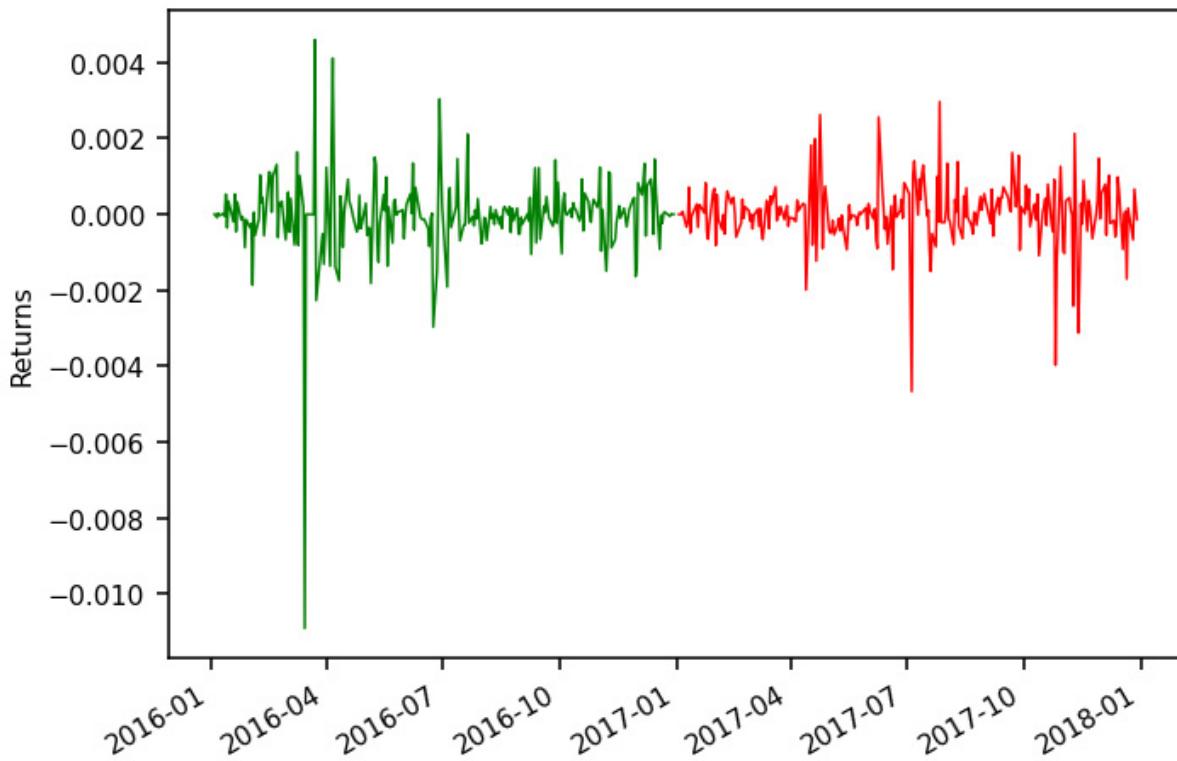


Figure 9.14: Line chart with the daily returns

3. Visualize the return series in quantiles and their cumulative returns for each quantile:

```
pf.plotting.plot_return_quantiles(  
    returns,  
    live_start_date="2017-01-01"  
)
```

The result is a box plot depicting the quantiles of daily, weekly, and monthly returns along with the distribution of cumulative returns.

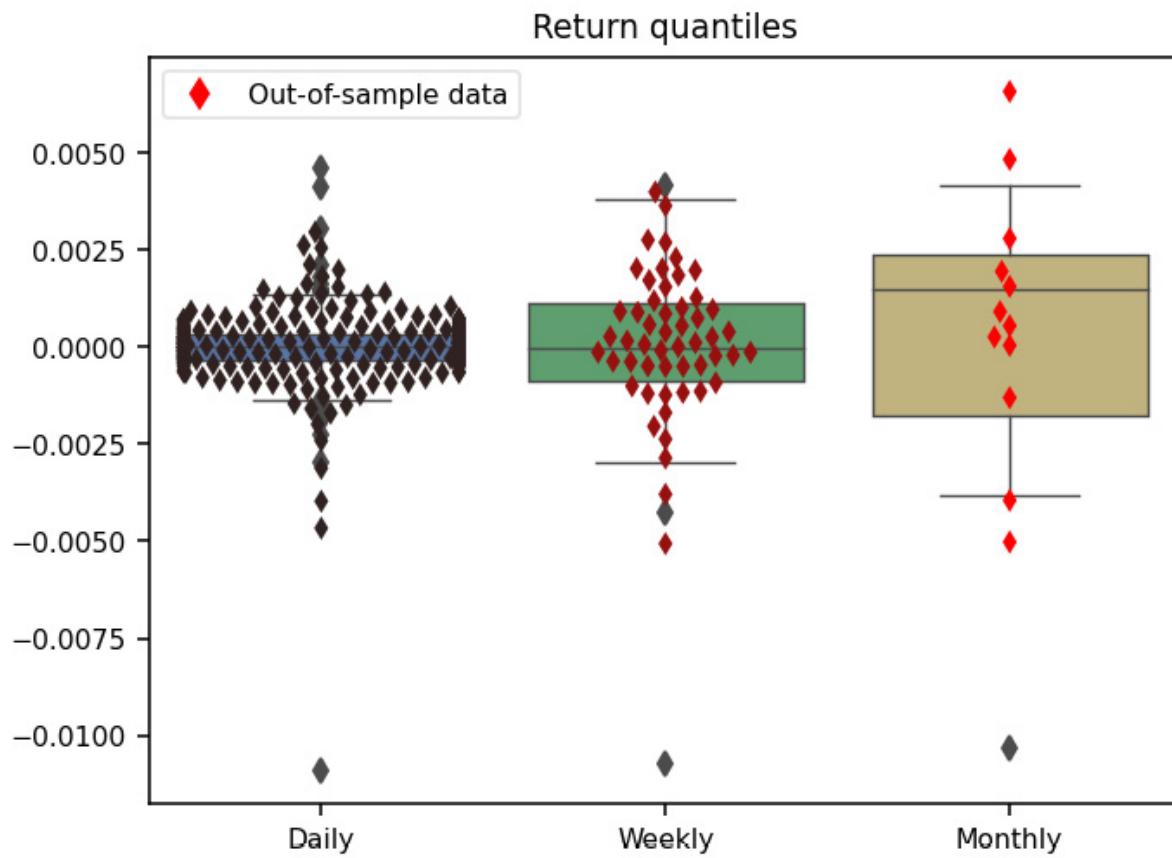


Figure 9.15: Box plot depicting the quantiles of daily, weekly, and monthly returns along with the distribution of cumulative returns

See also

You can dive deeper into Pyfolio's performance and return metrics by reviewing the source code here: <https://github.com/stefan-jansen/pyfolio-reloaded/blob/main/src/pyfolio/plotting.py>

Building a drawdown and rolling risk analysis

A focus only on returns without considering risk is like driving a fast car at high speeds without a seatbelt—it may work for a while, but the consequences can be catastrophic. Risk metrics provide the analytical framework to quantify and manage uncertainty, which lets traders make more informed decisions. These metrics offer insights into the potential volatility, drawdown, and other adverse conditions a strategy might encounter. By incorporating risk analytics into the trading process, traders can better assess the trade-offs between risk and return, optimize their portfolios for maximum risk-adjusted performance, and establish safeguards to mitigate potential losses.

Pyfolio offers several risk metrics to help maintain control of algorithmic trading systems. We'll look at several in this recipe.

Getting ready...

We assume the steps in the *Preparing Zipline Reloaded backtest results for Pyfolio Reloaded* recipe were followed in preparation for this recipe. We'll need `returns`, `positions`, `transactions`, and `benchmark_returns` defined for this recipe.

How to do it...

We'll look at Pyfolio Reloaded drawdown analysis and several rolling risk metrics. Rolling risk metrics help traders understand how strategy risk evolves through time.

1. Graphically depict the top 10 drawdowns over the strategy period:

```
pf.plotting.plot_drawdown_periods(returns, top=10)
```

The result is an equity curve depicting the cumulative strategy returns with vertical shading over the periods the strategy was in drawdown.

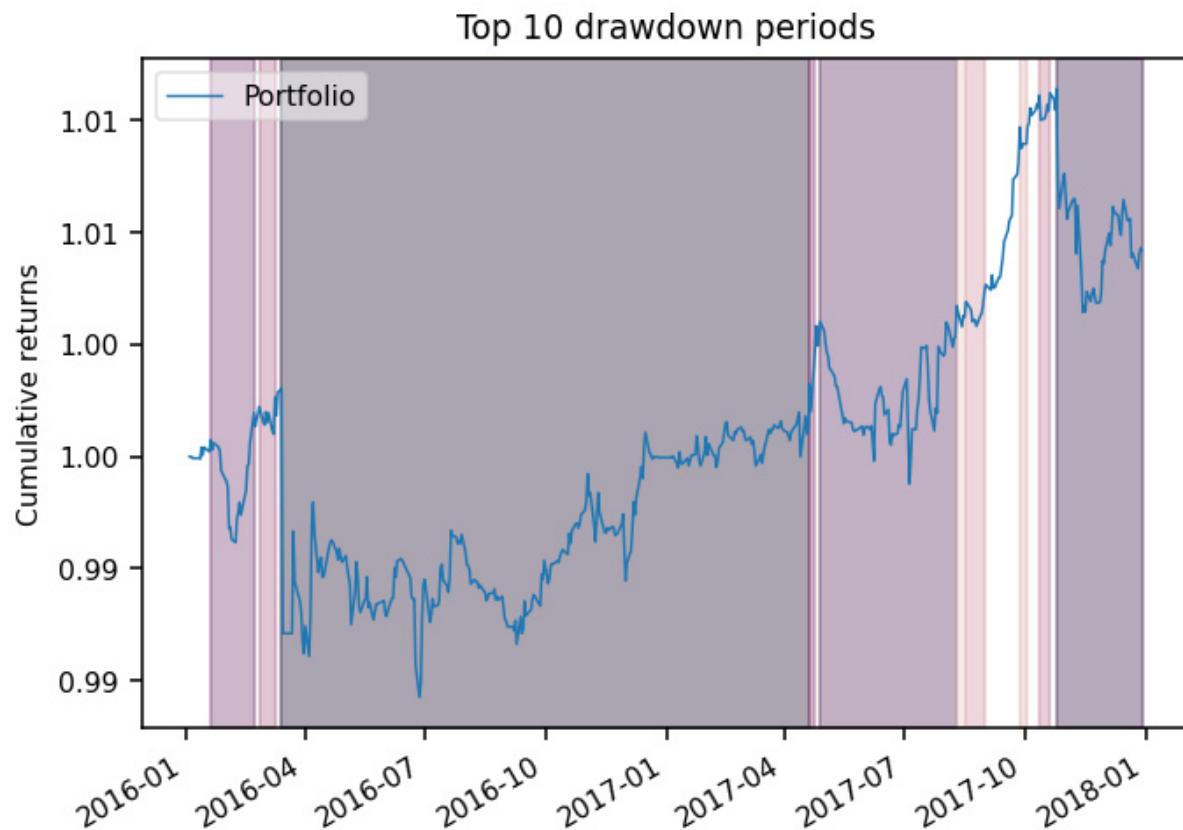


Figure 9.16: Equity curve depicting the cumulative strategy returns with vertical shading over the periods the strategy was in drawdown

2. Visualize the equity drawdown over time:

```
pf.plotting.plot_drawdown_underwater(returns)
```

The result is an **underwater plot** that visualizes the strategy drawdown amounts.

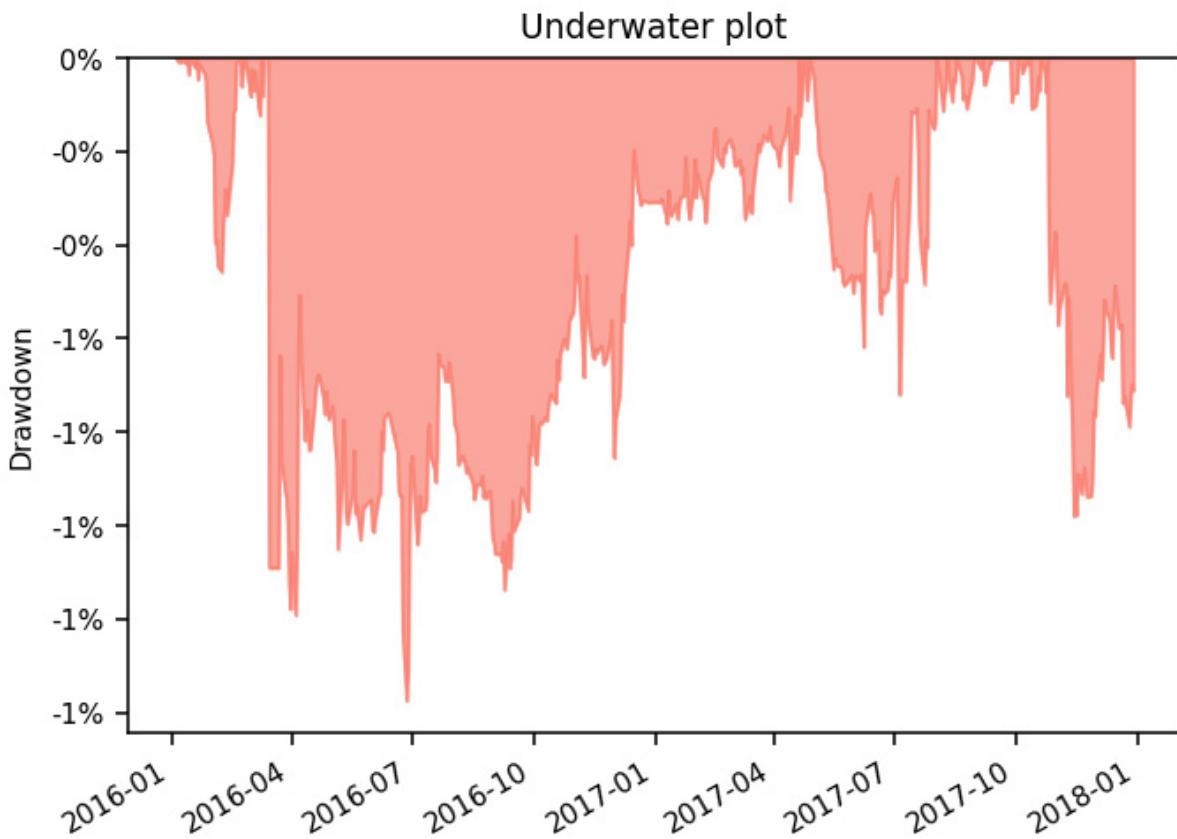


Figure 9.17: Underwater plot visualizing the strategy drawdown amounts

3. Create a table of the worst drawdowns with details of the amount, peak date, valley date, recovery date, and duration:

```
pf.plotting.show_worst_drawdown_periods(returns)
```

The result is a DataFrame detailing the top five drawdown periods.

Worst drawdown periods	Net drawdown in %	Peak date	Valley date	Recovery date	Duration
0	1.37	2016-03-14	2016-06-27	2017-04-20	289
1	0.98	2017-10-25	2017-11-14	NaT	NaN
2	0.72	2017-04-28	2017-07-05	2017-08-10	75
3	0.46	2016-01-20	2016-02-08	2016-02-22	24
4	0.12	2017-04-20	2017-04-21	2017-04-24	3

Figure 9.18: DataFrame detailing the top five drawdown periods

4. Plot the strategy 3-month rolling volatility against the benchmark:

```
pf.plotting.plot_rolling_volatility(
    returns,
    factor_returns=benchmark_returns,
```

```
        rolling_window=66  
    )
```

The result is a chart with the rolling 3-month volatility compared to the benchmark.

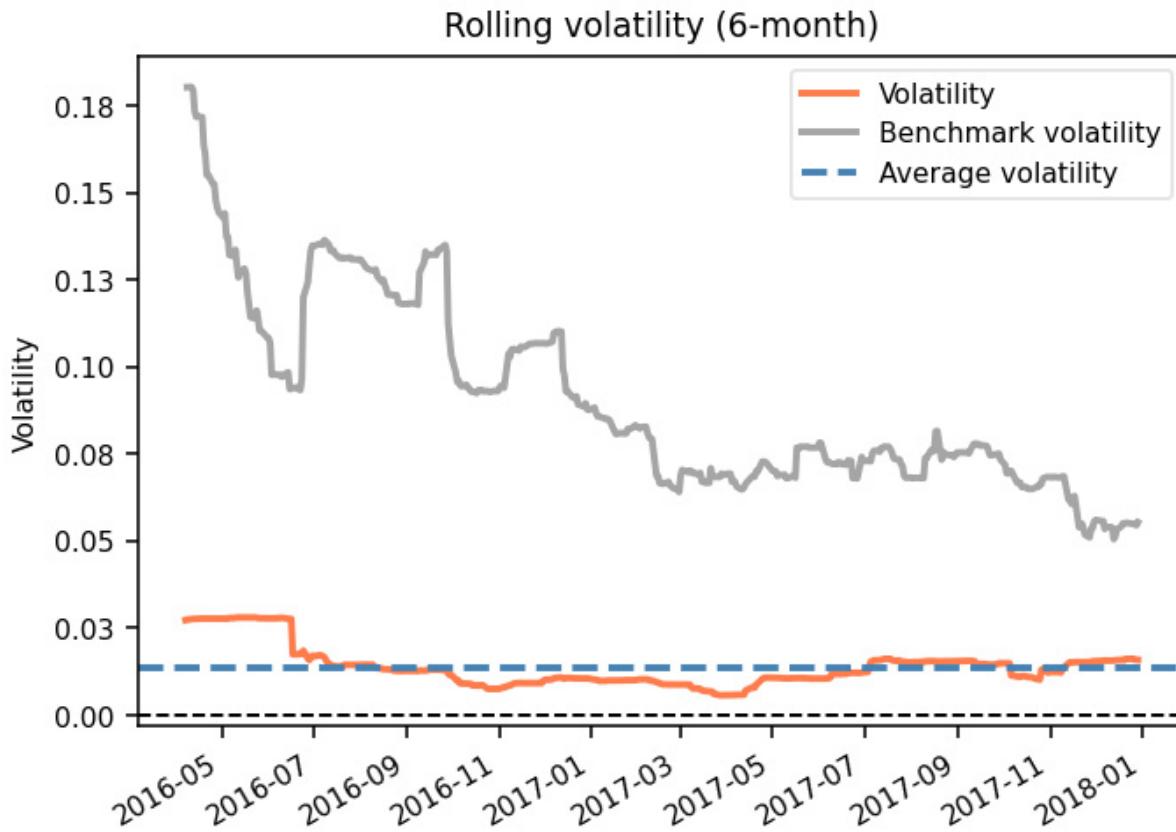


Figure 9.19: Chart with the rolling 3-month volatility compared to the benchmark

5. Plot the strategy 3-month rolling Sharpe ratio, its mean, and the benchmark Sharpe:

```
pf.plotting.plot_rolling_sharpe(  
    returns,  
    factor_returns=benchmark_returns,  
    rolling_window=66  
)
```

The result is a chart with the rolling 3-month Sharpe ratio compared to the benchmark.

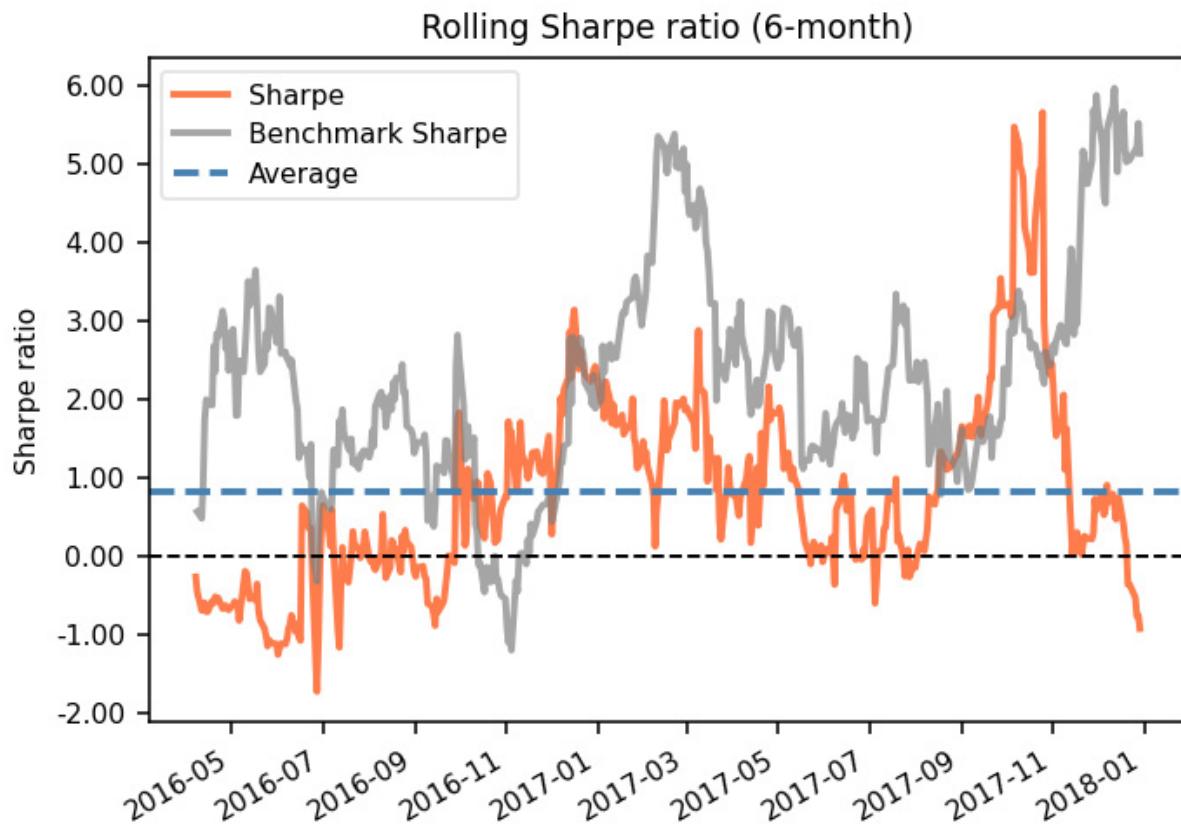


Figure 9.20: Chart with the rolling 3-month Sharpe ratio compared to the benchmark

How it works...

To generate the drawdown and rolling risk metrics, Pyfolio Reloaded uses the same underlying module as the performance and return metrics:

- `plot_drawdown_periods`: Computes the drawdown periods by identifying local maxima and subsequent declines. It then uses Matplotlib to visualize these drawdown periods, highlighting the start, valley, and end of each period on the plot.
- `plot_drawdown_underwater`: Finds the maximum cumulative return up to each point and subtracts the cumulative return from it. It then utilizes Matplotlib to plot the underwater curve, representing the negative drawdowns over time.
- `show_worst_drawdown_periods`: First, this computes the drawdowns, then sorts these drawdown periods and displays the top `n` worst drawdown periods in a pandas DataFrame.
- `plot_rolling_volatility`: Applies a rolling window and computes the standard deviation of returns within each window. It then uses Matplotlib to plot the computed rolling volatility against time.
- `plot_rolling_sharpe`: Applies a rolling window and computes the Sharpe ratio within each window. It then uses Matplotlib to plot the computed rolling Sharpe ratio against time.

Let's cover some of the key risk metrics we covered:

- **Annual volatility:** The annual volatility is calculated by taking the standard deviation of the daily returns and then annualizing it by multiplying it by the square root of the number of trading days (usually 252).
- **Sharpe ratio:** The Sharpe ratio is calculated as the mean of the excess returns divided by the standard deviation of those excess returns.
- **Max drawdown:** Max drawdown is calculated by identifying the maximum difference between a peak and a subsequent trough in a time series of portfolio values.

There's more...

Pyfolio allows for the overlay of specific event timelines, enabling a comparative analysis of portfolio and benchmark performance during these periods. An example would be assessing performance during the market downturn of what is now considered the New Normal. The following steps show you how.

1. Use `extract_interesting_date_ranges` to extract the strategy returns from pre-defined stress events:

```
times = pf.timeseries.extract_interesting_date_ranges(returns)
```

2. Then join with the benchmark returns, compute the cumulative returns, and plot:

```
(  
    times["New Normal"]  
    .to_frame("strategy_returns")  
    .join(benchmark_returns)  
    .add(1)  
    .cumprod()  
    .sub(1)  
    .plot()  
)
```

The result is a plot that depicts the strategy returns compared to the benchmark during the stress event.

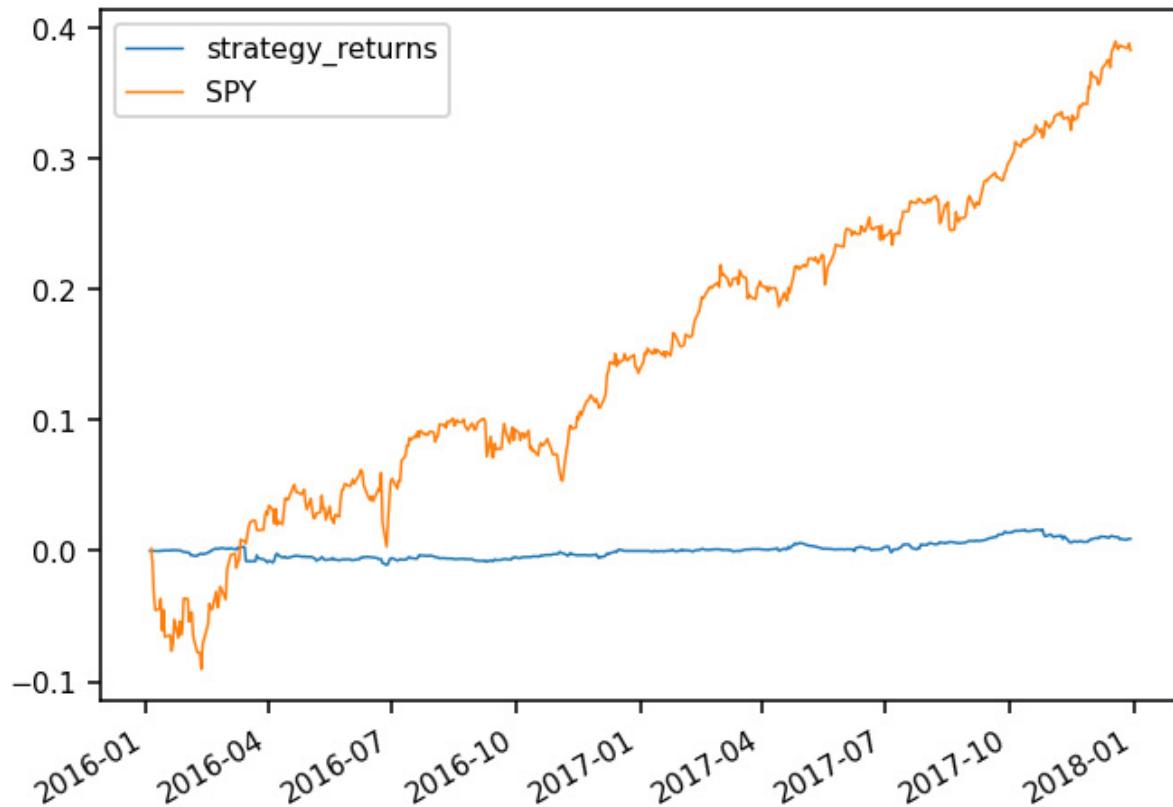


Figure 9.21: Plot that depicts the strategy returns compared to the benchmark during the stress event

See also

To get a list of stress periods and their start and end dates, check into the source code here:

https://github.com/stefan-jansen/pyfolio-reloaded/blob/main/src/pyfolio/interesting_periods.py

Analyzing strategy holdings, leverage, exposure, and sector allocations

We can extend our strategy analysis with Pyfolio by analyzing holdings, leverage, and sector allocations over time. Analyzing holdings over time helps traders understand the diversification and concentration risks within a portfolio. It helps traders identify overexposure to specific assets, which could be detrimental in adverse market conditions. Leverage analysis is equally important, as excessive borrowing can amplify losses, leading to significant drawdowns or even portfolio liquidation. Monitoring leverage levels over time allows traders to adjust their risk exposure in line with their risk tolerance and market outlook.

Sector allocation analysis provides insights into how diversified the portfolio is across different industries. This is important for risk management, as different sectors respond differently to economic cycles and market events. Understanding sector allocations can help traders optimize their portfolios for various market conditions and potentially enhance returns while mitigating risks. In this recipe, we'll look at how Pyfolio is used for portfolio analysis.

Getting ready...

We assume the steps in the *Preparing Zipline Reloaded backtest results for Pyfolio Reloaded* recipe were followed in preparation for this recipe. We'll need `returns`, `positions`, and `sector_map` defined for this recipe.

How to do it...

We'll look at how the strategy holdings, leverage, exposure, and sector allocation evolve throughout the analysis period.

1. Plot the number of daily holdings, average holdings by month, and overall average holdings:

```
pf.plotting.plot_holdings(returns, positions)
```

The result is a plot of the number of daily holdings, average holdings by month, and overall average holdings.

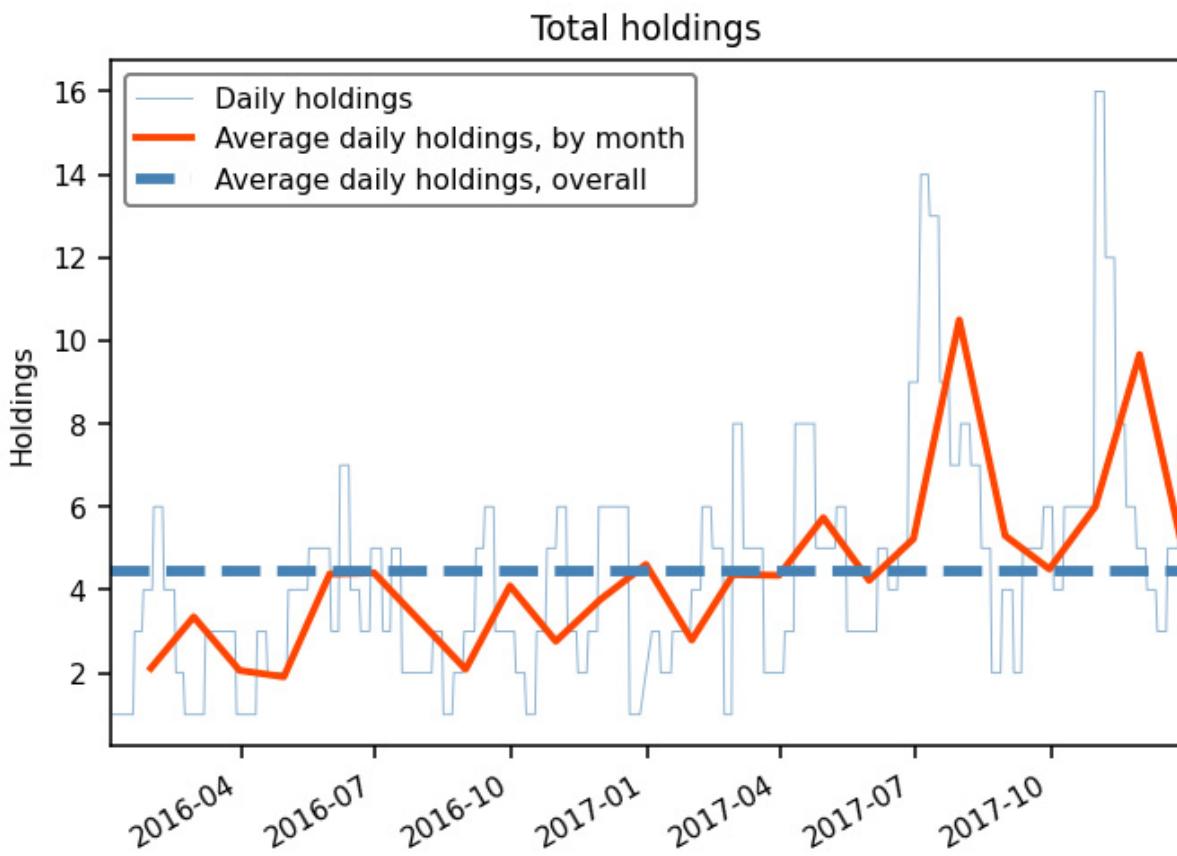


Figure 9.22: Plot of number of daily holdings, average holdings by month, and overall average holdings

2. Plot the number of long and short holdings:

```
pf.plotting.plot_long_short_holdings(
    returns,
    positions
)
```

The result is a plot of the number of long and short holdings.

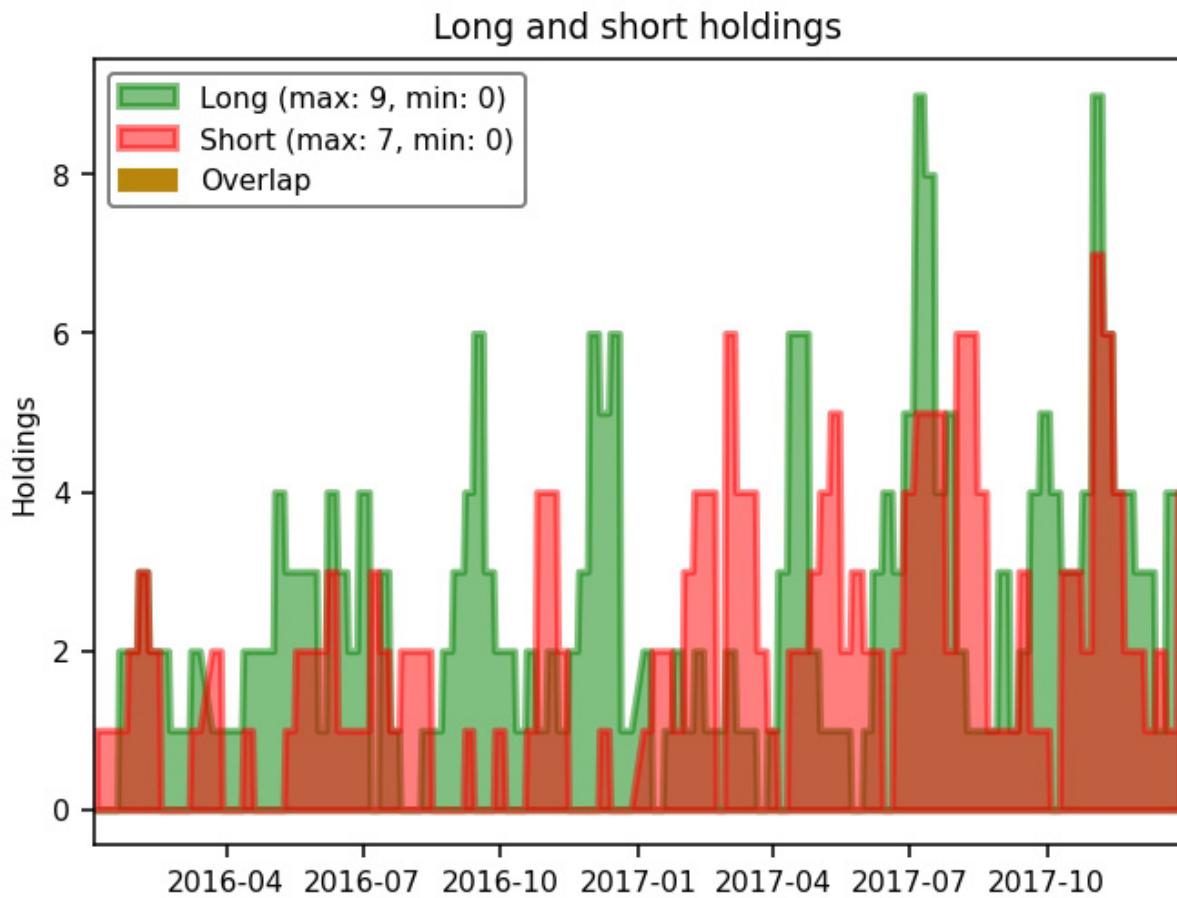


Figure 9.23: Plot of the number of long and short holdings

3. Plot the gross strategy leverage:

```
pf.plotting.plot_gross_leverage(returns, positions)
```

The result is a plot of the gross strategy leverage.

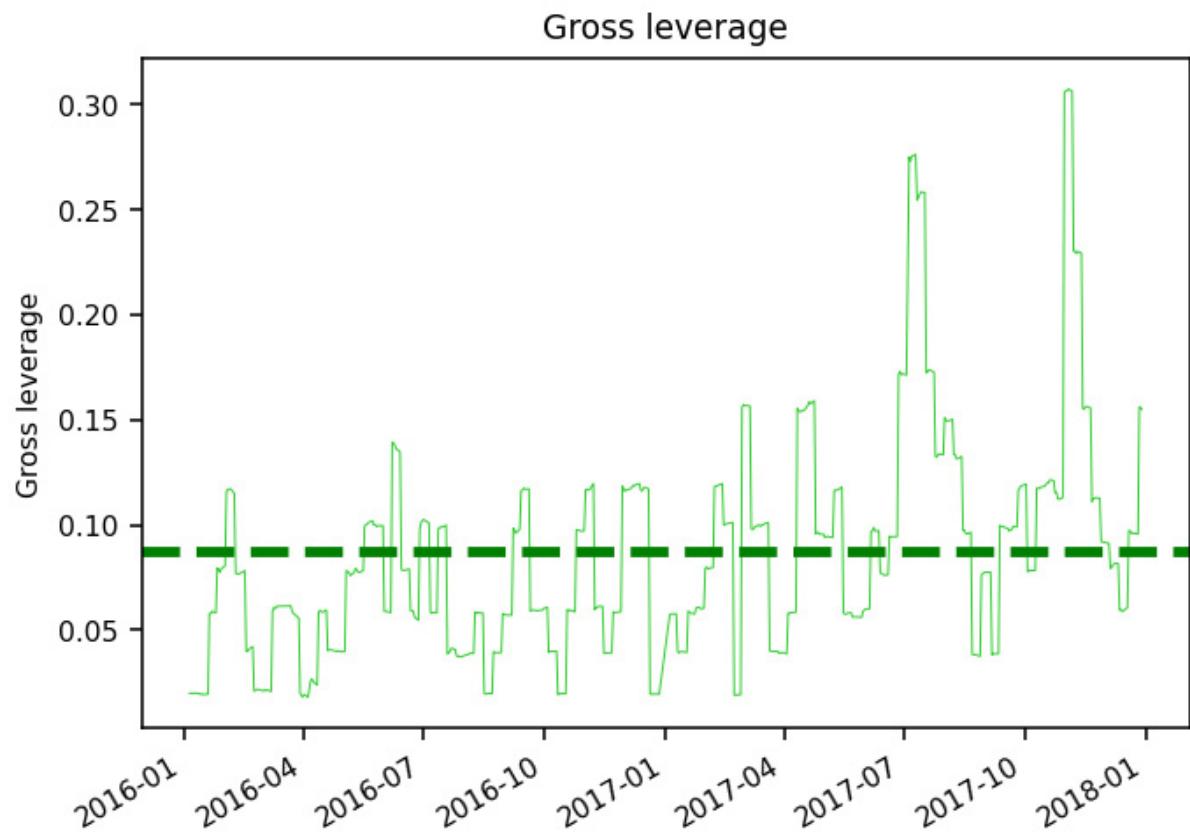


Figure 9.24: Plot of the strategy's gross leverage

4. Plot the long, short, and net exposure:

```
pf.plotting.plot_exposures(returns, positions)
```

The result is a plot of long, short, and net exposure.

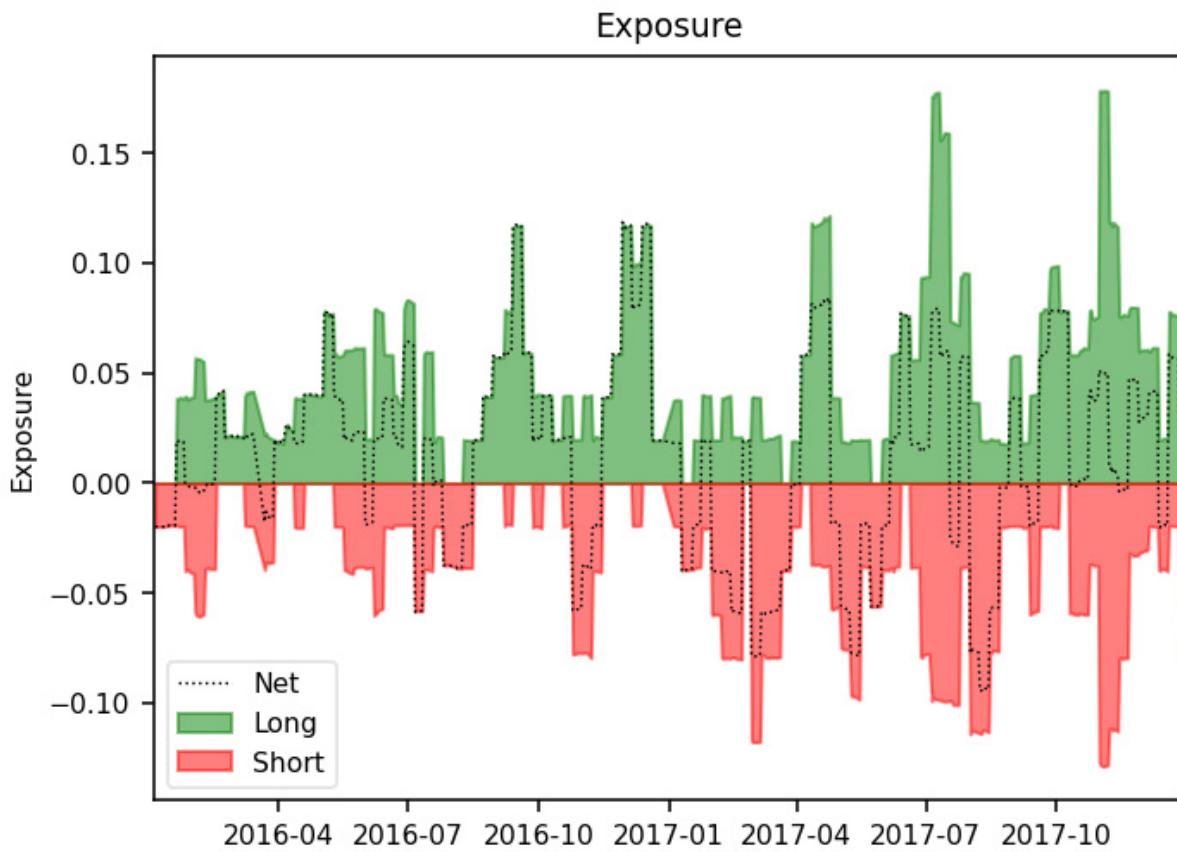


Figure 9.25: Plot of the strategy's long, short, and net exposure

5. Generate a pandas DataFrame with the percentage allocation of each position:

```
positions_alloc = pf.pos.get_percent_alloc(positions)
```

6. Generate a table of the top long, short, and net positions of all time:

```
pf.plotting.show_and_plot_top_positions(
    returns,
    positions_alloc,
    show_and_plot=2
)
```

The result is a pandas DataFrame and chart with the maximum percentage allocation of each of the top 10 strategy holdings.

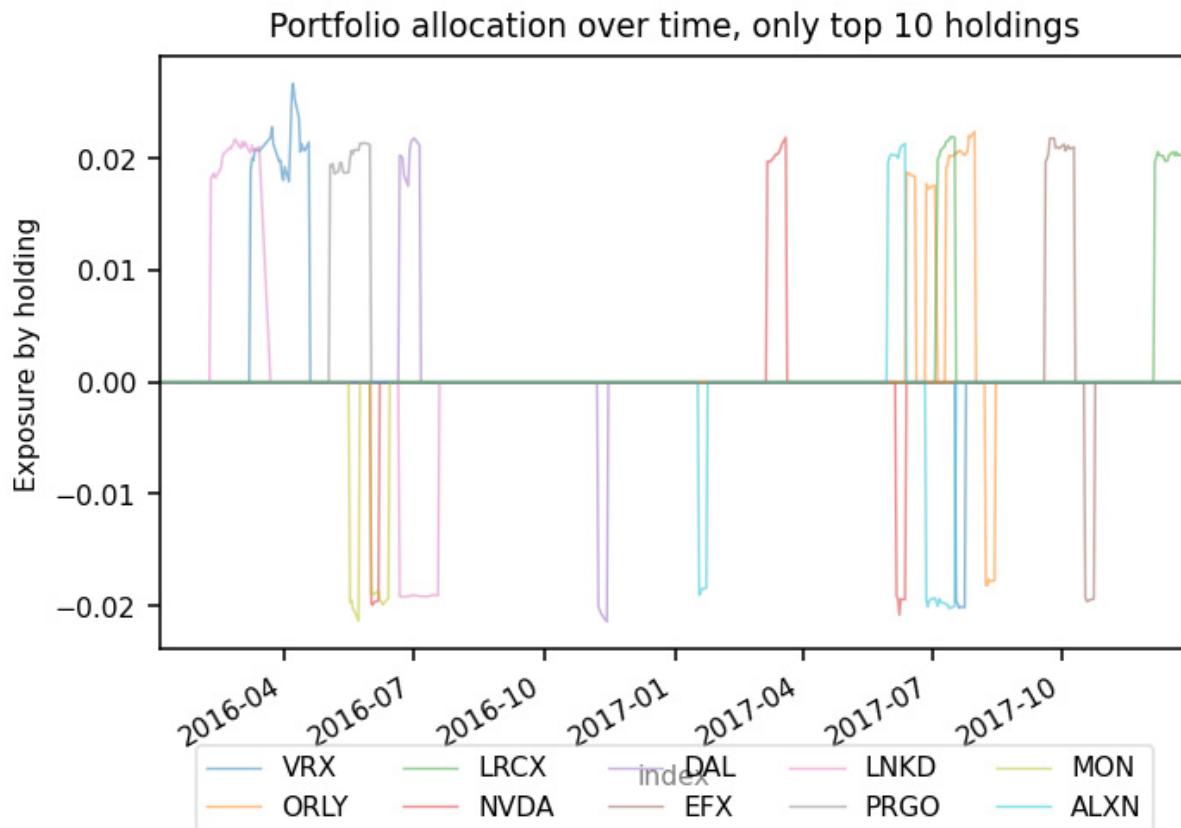


Figure 9.26: pandas DataFrame and chart with the maximum percentage allocation of each of the top 10 strategy holdings

7. Generate the sector allocations based on the positions and sector mapping:

```
sector_alloc = pf.pos.get_sector_exposures(
    positions,
    symbol_sector_map=sector_map
)
```

8. Plot the sector allocation using Pyfolio's `plot_sector_allocations` method:

```
pf.plotting.plot_sector_allocations(
    returns,
    sector_alloc=sector_alloc
)
```

The result is a chart that graphically depicts the strategy's sector allocation, including cash.

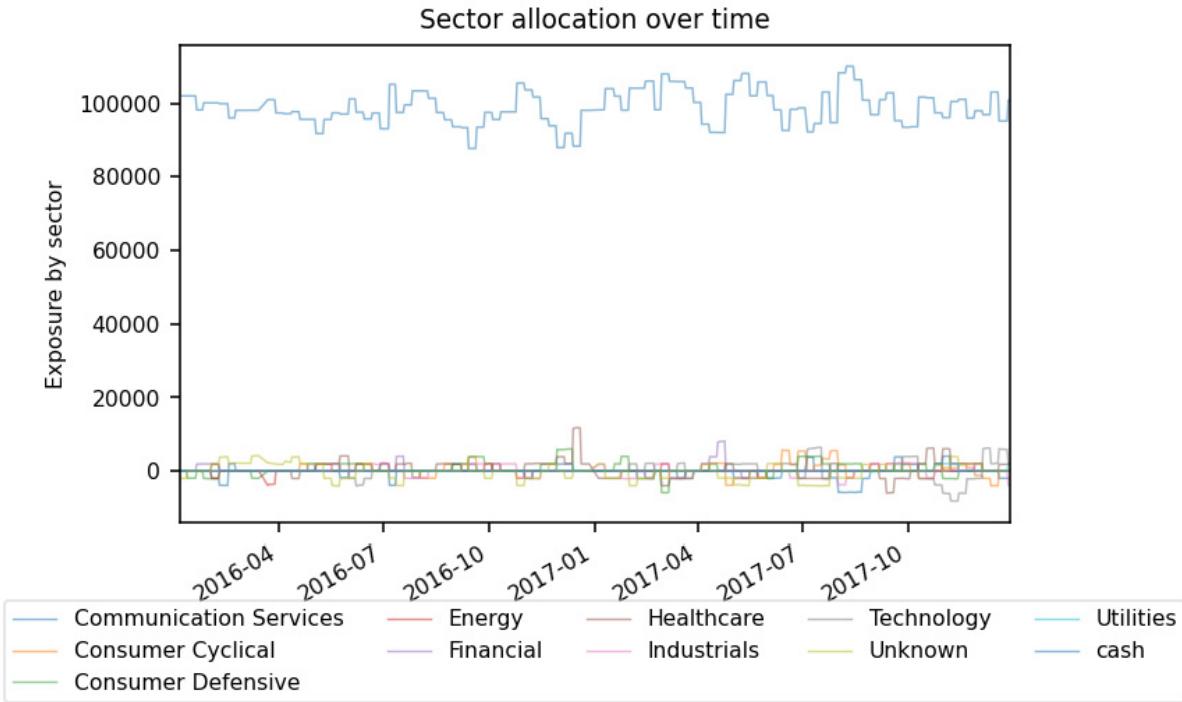


Figure 9.27: Chart of the strategy's sector allocation over time

How it works...

The `plot_holdings` function is designed to visualize the total number of stocks with an active position (either short or long) over time. It takes in a pandas Series of daily returns (`returns`) and a pandas DataFrame of daily net position values (`positions`). The plot gives traders a time-series representation of the strategy's exposure to the market. The daily holdings line gives you an immediate sense of how many positions are held each day, while the monthly and overall averages provide a more smoothed-out perspective, useful for understanding the strategy's typical market exposure.

The `plot_long_short_holdings` takes the same arguments as `plot_holdings`. It filters out cash positions and counts the number of long and short positions for each day. Using Matplotlib, it plots these counts as shaded regions over time, with green indicating long positions and red indicating short positions. Conceptually, the plot serves as a visual representation of the strategy's exposure to long and short positions over time.

`plot_gross_leverage` takes `returns` and `positions` and plots the gross leverage over time using Matplotlib. The output provides a time-series view of the strategy's gross leverage which is the ratio of the absolute sum of the long and short positions to the strategy's net asset value. For traders using

margin, monitoring gross leverage is critical to quantify the level of risk exposure relative to the portfolio's net asset value. Excessive leverage can amplify both gains and losses, potentially leading to rapid depletion of capital or margin calls. Understanding and managing leverage is essential for risk control and ensuring compliance with trading limits or regulatory requirements.

`plot_exposures` helps visualize the long, short, and net exposure across the strategy. Exposure is calculated by summing the values of either long, short, or all positions on each day and dividing by the sum of all positions on the same day. Long exposure represents the proportion of the portfolio invested in long positions, short exposure indicates the proportion in short positions. The net exposure gives an overall exposure level, which can be interpreted as the strategy's directional bias. A positive net exposure would suggest a bullish stance, while a negative value would indicate a bearish outlook.

The `show_and_plot_top_positions` is a snapshot of a strategy's most significant long and short positions. The function takes a pandas DataFrame of positions and a specified number N to identify the top N long and short positions by net market value. It then calculates the mean position for each asset over the analysis period. The function outputs a table displaying these top positions and also generates a bar plot to visualize them over time.

Finally, `plot_sector_allocations` calculates the daily sector allocations by aggregating the positions based on the sector mappings for each stock. The plot is a snapshot of the strategy's average exposure to sectors. It is useful for understanding the portfolio's diversification and risk profile across sectors for hedging purposes.

There's more...

If you're trading intraday strategies, transaction costs play an important part in the profitability of a strategy. Transaction costs can be direct, which are the commissions and fees we pay to our broker and exchanges, and indirect, which include market impact and slippage.

Pyfolio has several tools for measuring these costs:

- `plot_turnover`: Plots turnover, which is the number of shares traded for a period as a fraction of total shares. The output displays the daily total, daily average per month, and all-time daily.
- `plot_slippage_sweep`: Plots equity curves at different per-dollar slippage assumptions.
- `plot_slippage_sensitivity`: Plots curve relating per-dollar slippage to average annual returns.
- `plot_capacity_sweep`: Performs a sweep over different starting portfolio values to assess the impact on the Sharpe ratio, considering transaction costs and market slippage, and then plots the resulting Sharpe ratios against the portfolio values.
- `plot_daily_turnover_hist`: Plots a histogram of daily turnover rates.

- `plot_daily_volume`: Plots the trading volume per day.

Active traders that are concerned about their transaction costs can use these additional tools to analyze costs on their strategy performance.

See also

If you're new to asset allocation and why it matters, you can get a good primer at Investopedia here: <https://www.investopedia.com/terms/a/assetallocation.asp>. A concept close to leverage is margin, which is discussed here: <https://www.investopedia.com/terms/m/margin.asp>. Finally, tools like Trade Blotter can help automate a lot of the risk and performance analytics for you: <https://tradeblotter.io>.

Breaking Down Strategy Performance to Trade Level

So far in this chapter, we've considered risk and performance metrics at the strategy level. This is an important perspective, but not the only one. In this recipe, we'll use Pyfolio Reloaded to look at the strategy at the trade level. Examining strategy risk and performance at the trade level provides more granular insights into how the returns of the strategy are composed. It lets identify specific trades, or classes of trades, that may be contributing disproportionately to risk or returns. This level of examination helps traders optimize strategy features like trade execution, entry and exit criteria, or even asset class. In this recipe, we'll look at trade-level statistics for our strategy.

Getting ready...

We assume the steps in the *Preparing Zipline Reloaded backtest results for Pyfolio Reloaded* recipe were followed in preparation for this recipe. We'll need `transactions` defined for this recipe.

How to do it...

Pyfolio has a utility function to identify a trade from the transaction history. A trade is considered an opening and closing transaction of the same quantity for the same asset.

1. Extract the round trips from the strategy transaction history:

```
round_trips = pf.round_trips.extract_round_trips(
    transactions[["amount", "price", "symbol"]]
)
```

The result is a pandas DataFrame with the profit or loss, return, and duration of all round trips.

	pnl	open_dt	close_dt	long	rt_returns	symbol	duration
0	-29.89	2016-05-17 20:00:00+00:00	2016-05-24 20:00:00+00:00	True	-0.015012	AAL	7 days 00:00:00
1	78.10	2016-07-19 20:00:00+00:00	2016-08-09 20:00:00+00:00	False	0.039433	AAL	21 days 00:00:00
2	57.12	2016-05-03 20:00:00+00:00	2016-05-24 20:00:00+00:00	True	0.028577	AAPL	21 days 00:00:00
3	-16.51	2017-06-20 20:00:00+00:00	2017-06-27 20:00:00+00:00	True	-0.008758	AAPL	7 days 00:00:00
4	35.88	2017-09-26 20:00:00+00:00	2017-10-10 20:00:00+00:00	True	0.018023	AAPL	14 days 00:00:00
...
254	-2.86	2016-01-26 21:00:00+00:00	2016-02-02 21:00:00+00:00	False	-0.044688	WMT	7 days 00:00:00
255	-54.30	2016-01-26 21:00:00+00:00	2016-02-09 21:00:00+00:00	False	-0.028281	WMT	14 days 00:00:00
256	-40.39	2017-01-18 21:00:00+00:00	2017-01-31 21:00:00+00:00	True	-0.019774	WMT	13 days 00:00:00
257	28.62	2017-02-28 21:00:00+00:00	2017-03-07 21:00:00+00:00	False	0.014944	WMT	7 days 00:00:00
258	-86.94	2017-10-31 20:00:00+00:00	2017-11-14 21:00:00+00:00	False	-0.043294	WMT	14 days 01:00:00

Figure 9.28: DataFrame containing the round trips extracted from the transaction history

2. Use the `print_round_trip_stats` function to generate the summary statistics of the strategy trades:

```
pf.round_trips.print_round_trip_stats(
    round_trips.rename(
        columns={"rt_returns": "returns"})
)
```

The result is a series of pandas DataFrames with aggregate and asset-specific statistics.

Summary stats	All trades	Short trades	Long trades
Total number of round_trips	259.00	116.00	143.00
Percent profitable	0.56	0.46	0.64
Winning round_trips	145.00	53.00	92.00
Losing round_trips	114.00	63.00	51.00
Even round_trips	0.00	0.00	0.00
PnL stats	All trades	Short trades	Long trades
Total profit	\$861.37	\$-599.93	\$1461.30
Gross profit	\$8204.94	\$2347.53	\$5857.41
Gross loss	\$-7343.57	\$-2947.46	\$-4396.11
Profit factor	\$1.12	\$0.80	\$1.33
Avg. trade net profit	\$3.33	\$-5.17	\$10.22
Avg. winning trade	\$56.59	\$44.29	\$63.67
Avg. losing trade	\$-64.42	\$-46.79	\$-86.20
Ratio Avg. Win:Avg. Loss	\$0.88	\$0.95	\$0.74
Largest winning trade	\$374.66	\$167.40	\$374.66
Largest losing trade	\$-889.20	\$-258.42	\$-889.20
Duration stats	All trades	Short trades	Long trades
Avg duration	12 days 10:42:51.428571428	12 days 03:47:04.137931034	12 days 16:20:08.391608391
Median duration	8 days 00:00:00	8 days 00:00:00	8 days 00:00:00
Longest duration	35 days 01:00:00	29 days 00:00:00	35 days 01:00:00
Shortest duration	6 days 00:00:00	6 days 00:00:00	6 days 00:00:00
Return stats	All trades	Short trades	Long trades
Avg returns all round_trips	0.26%	-0.46%	0.84%
Avg returns winning	3.18%	2.26%	3.71%
Avg returns losing	-3.46%	-2.74%	-4.35%
Median returns all round_trips	0.31%	-0.21%	0.86%
Median returns winning	2.03%	1.60%	2.50%
Median returns losing	-2.29%	-2.28%	-2.42%
Largest winning trade	19.68%	8.31%	19.68%
Largest losing trade	-46.94%	-15.00%	-46.94%

Figure 9.29: Trade-level summary statistics

3. Plot the duration of each round trip, per asset, over time:

```
pf.plotting.plot_round_trip_lifetimes(round_trips)
```

The result is a plot that visualizes the holding period for each asset traded.

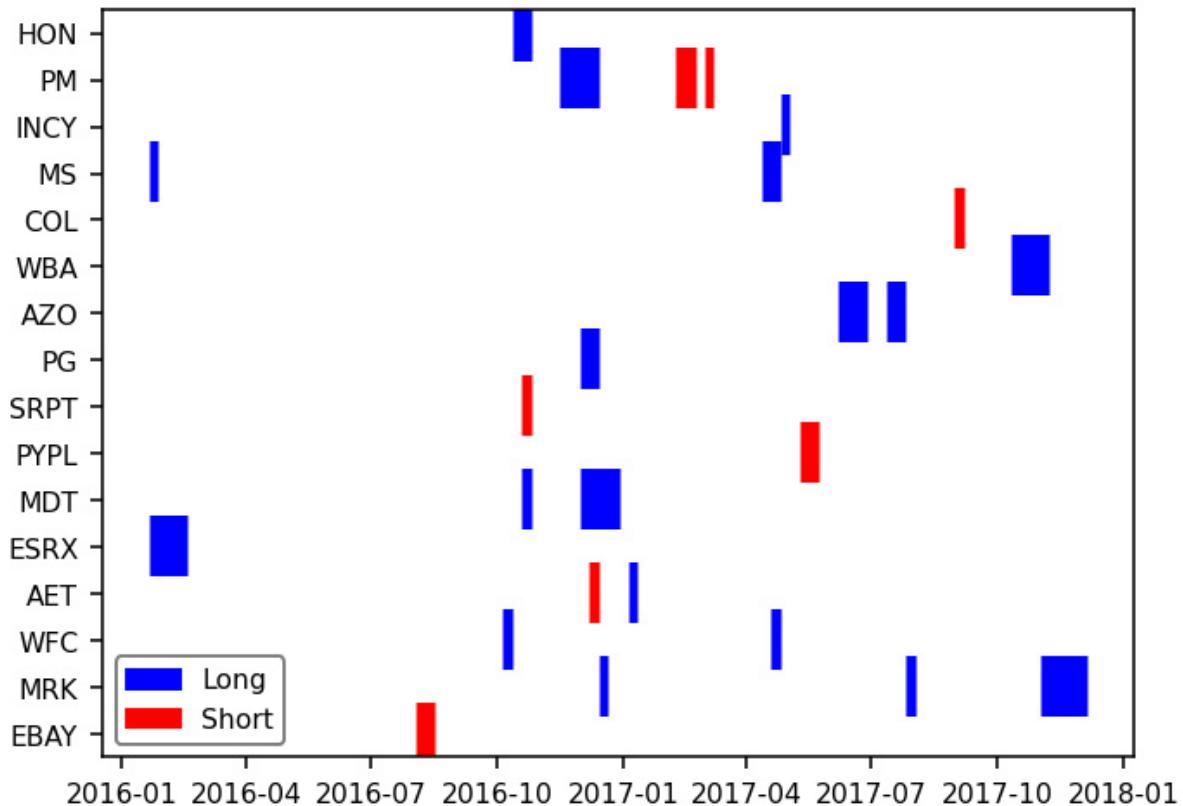


Figure 9.30: Plot visualizing the holding period for each asset traded

How it works...

Round trips are computed by identifying the opening and closing transactions for each asset in a strategy. We start by extracting executed orders from the transactions DataFrame and categorizes them as either “buy” or “sell” based on the sign of the quantity. It then groups these transactions by asset and sorts them by date. For each asset, the algorithm iterates through the sorted transactions to match an opening transaction (buy for long or sell for short) with its corresponding closing transaction (sell for long or buy for short). The time, price, and quantity of both the opening and closing transactions are recorded to compute the round-trip characteristics such as duration, profit and loss, and other relevant metrics.

Conceptually, a round trip is a complete cycle of opening and closing a position in a specific asset, and its analysis is crucial for understanding the effectiveness of our strategy. Metrics derived from round-trip analysis can provide valuable insights into transaction costs, holding periods, and the risk/return profile of individual trades, thereby aiding in strategy optimization and risk management.

Now that we covered how the plots are generated, let's review some of the key statistics output by Pyfolio:

- **Percent profitable:** Calculated by dividing the number of profitable round trips by the total number of round trips
- **Winning round_trips:** Calculated by counting the number of round-trip trades that resulted in a positive profit and loss
- **Losing round_trips:** Calculated by counting the number of round-trip trades that resulted in a negative profit and loss
- **Profit factor:** Calculated by dividing the sum of all profitable trades by the absolute sum of all losing trades
- **Avg. winning trade:** Computed by taking the mean of all profitable trades

There's more...

Depending on our strategy, it's sometimes useful to aggregate the round trip performance statistics by section, instead of by asset. By passing in the `sector_map` dictionary we built in the *Preparing Zipline Reloaded backtest results for Pyfolio Reloaded* recipe, we can aggregate by sector.

1. Apply the sector mapping to the extracted round trips:

```
round_trips_by_sector = pf.round_trips.apply_sector_mappings_to_round_trips(
    round_trips,
    sector_map
)
```

The result is a pandas DataFrame similar to *Figure 9.28* except aggregated by sector, instead of asset symbol.

	pnl	open_dt	close_dt	long	rt_returns	symbol	duration
0	-29.89	2016-05-17 20:00:00+00:00	2016-05-24 20:00:00+00:00	True	-0.015012	Industrials	7 days 00:00:00
1	78.10	2016-07-19 20:00:00+00:00	2016-08-09 20:00:00+00:00	False	0.039433	Industrials	21 days 00:00:00
2	57.12	2016-05-03 20:00:00+00:00	2016-05-24 20:00:00+00:00	True	0.028577	Technology	21 days 00:00:00
3	-16.51	2017-06-20 20:00:00+00:00	2017-06-27 20:00:00+00:00	True	-0.008758	Technology	7 days 00:00:00
4	35.88	2017-09-26 20:00:00+00:00	2017-10-10 20:00:00+00:00	True	0.018023	Technology	14 days 00:00:00
...
254	-2.86	2016-01-26 21:00:00+00:00	2016-02-02 21:00:00+00:00	False	-0.044688	Consumer Defensive	7 days 00:00:00
255	-54.30	2016-01-26 21:00:00+00:00	2016-02-09 21:00:00+00:00	False	-0.028281	Consumer Defensive	14 days 00:00:00
256	-40.39	2017-01-18 21:00:00+00:00	2017-01-31 21:00:00+00:00	True	-0.019774	Consumer Defensive	13 days 00:00:00
257	28.62	2017-02-28 21:00:00+00:00	2017-03-07 21:00:00+00:00	False	0.014944	Consumer Defensive	7 days 00:00:00
258	-86.94	2017-10-31 20:00:00+00:00	2017-11-14 21:00:00+00:00	False	-0.043294	Consumer Defensive	14 days 01:00:00

Figure 9.31: DataFrame with round trips mapped to sector

2. Plot the duration of each round trip, per sector, over time:

```
pf.plotting.plot_round_trip_lifetimes(round_trips)
```

The result is a plot that visualizes the holding period for each sector traded.

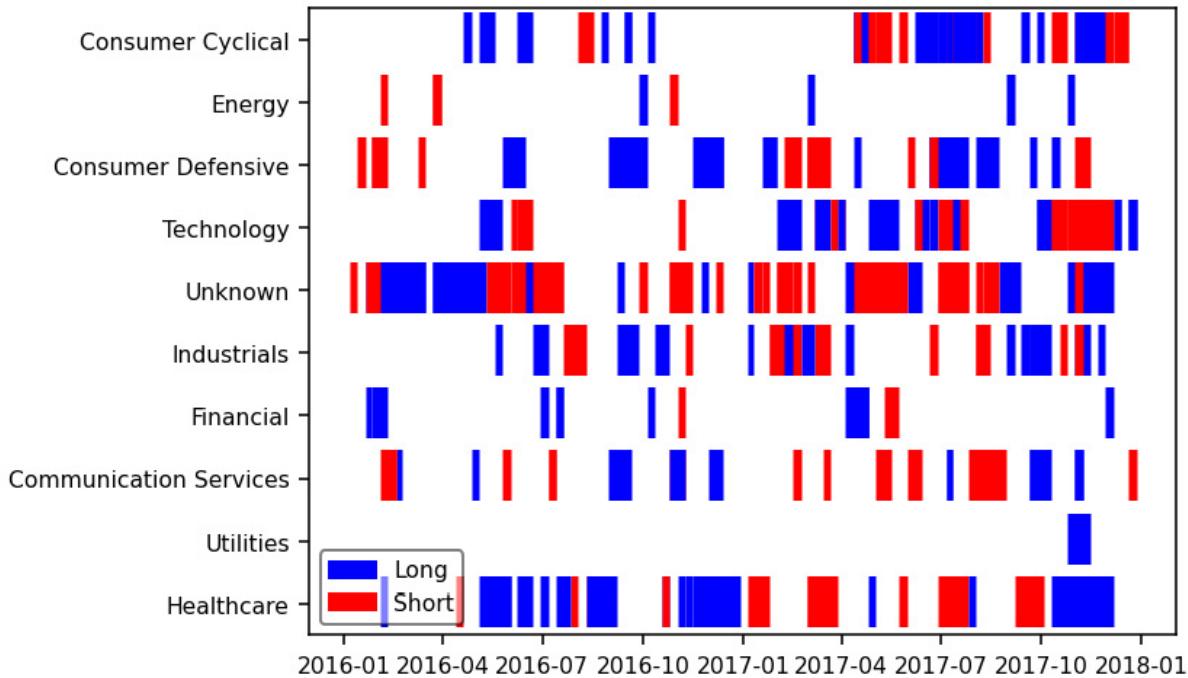


Figure 9.32: Plot visualizing the holding period for each sector traded

See also

As we've said, no single risk or performance metric is enough for a complete picture of your strategy. Pyfolio provides many important risk and performance statistics, but there are more. For a good walkthrough of other important risk and performance metrics, you can refer to blogs online.

Set Up the Interactive Brokers Python API

In *Chapters 1 through 9*, we learned the foundational tools and techniques of algorithmic trading. Now, we'll put this into practice using Python with the **Interactive Brokers (IB) API** and **Trader Workstation (TWS)**. TWS is an advanced trading platform that is used by a wide range of traders, both professional and retail. You may consider alternatives to IB that offer API access. Some other brokers include Alpaca, Think Or Swim, Tasty Trade, and Tradier. When opening an account, make sure to consider market access, account type, commissions, and other factors when selecting a broker. Different brokers and account types may have different costs and restrictions.

TWS is known for its suite of trading tools and features that really enhance the trading experience. TWS is equipped with robust risk management tools so traders can effectively manage and mitigate potential trading risks. IB has unparalleled global market access with the ability to trade across a vast array of financial instruments in 135 markets spanning 33 countries. This feature is useful for traders who want to diversify their portfolios on a global scale.

Further, the paper trading functionality of TWS is an invaluable tool for both novice and experienced traders. It provides a risk-free environment for testing and refining trading strategies using real-time market conditions. This feature is instrumental in helping algorithmic traders develop and hone their algorithms without putting their money at risk. Finally, TWS's API integration is a significant advantage since it exposes all the features of TWS through an API that is accessible with Python. The IB API is especially useful for algorithmic traders since it lets them automate their trading strategies. We will take advantage of the paper trading account and IB API in the next three chapters.

In this chapter, we present the following recipes:

- Building an algorithmic trading app
- Creating a **Contract** object with the IB API
- Creating an **Order** object with the IB API
- Fetching historical market data
- Getting a market data snapshot
- Streaming live tick data
- Storing live tick data in a local SQL database

Building an algorithmic trading app

When using the IB API, there's a lot of code that can be reused across different trading apps. Connecting to TWS, generating orders, and downloading data are done the same way despite the trading strategy. That's why it's a best practice to get the reusable code out of the way first. But before we can start building our algorithmic trading app, we need to install TWS and the IB API.

We'll begin with three important topics when using the IB API:

- The first is the architecture of the IB API, which operates on an asynchronous model. In this model, operations are not executed in a linear, blocking manner. Instead, actions are initiated by requests, and responses to these requests are handled via callbacks.
- The second concept is inheritance, which is common across all computer programming languages. Inheritance is where a new child class acquires the properties and methods of another, parent, class. Our trading app will inherit functionality from two parent classes provided by the IB API (**EClient** and **EWrapper**).
- The third concept is overriding where a child class provides the specific implementation for a method that is defined in a parent class.

We'll learn more about how these three concepts function in the *How it works* section of this recipe.

As we learned in the introduction of this chapter, TWS is IB's trading platform. While running, TWS acts as a server and exposes ports to which the IB API connects. It's through this connection that the IB API operates. In this chapter, we'll install TWS, the IB API, and start building the reusable parts of our algorithmic trading app.

Getting ready...

Before we install the IB API, we need TWS installed on our local computer. You can download TWS for your computer from the IB website (<https://www.interactivebrokers.com/en/trading/tws.php#tws-software>). It's available for Windows, Mac, and Linux.

Once it is installed, you can start TWS.

If you have an account, log in. If not, then you can use the Demo account option on the login screen.

We need to change some settings to allow our Python app to connect to TWS. Navigate to **Trader Workstation Configuration** under **Edit** → **Global Configuration** → **API** → **Settings**. You should see a screen that looks like this:

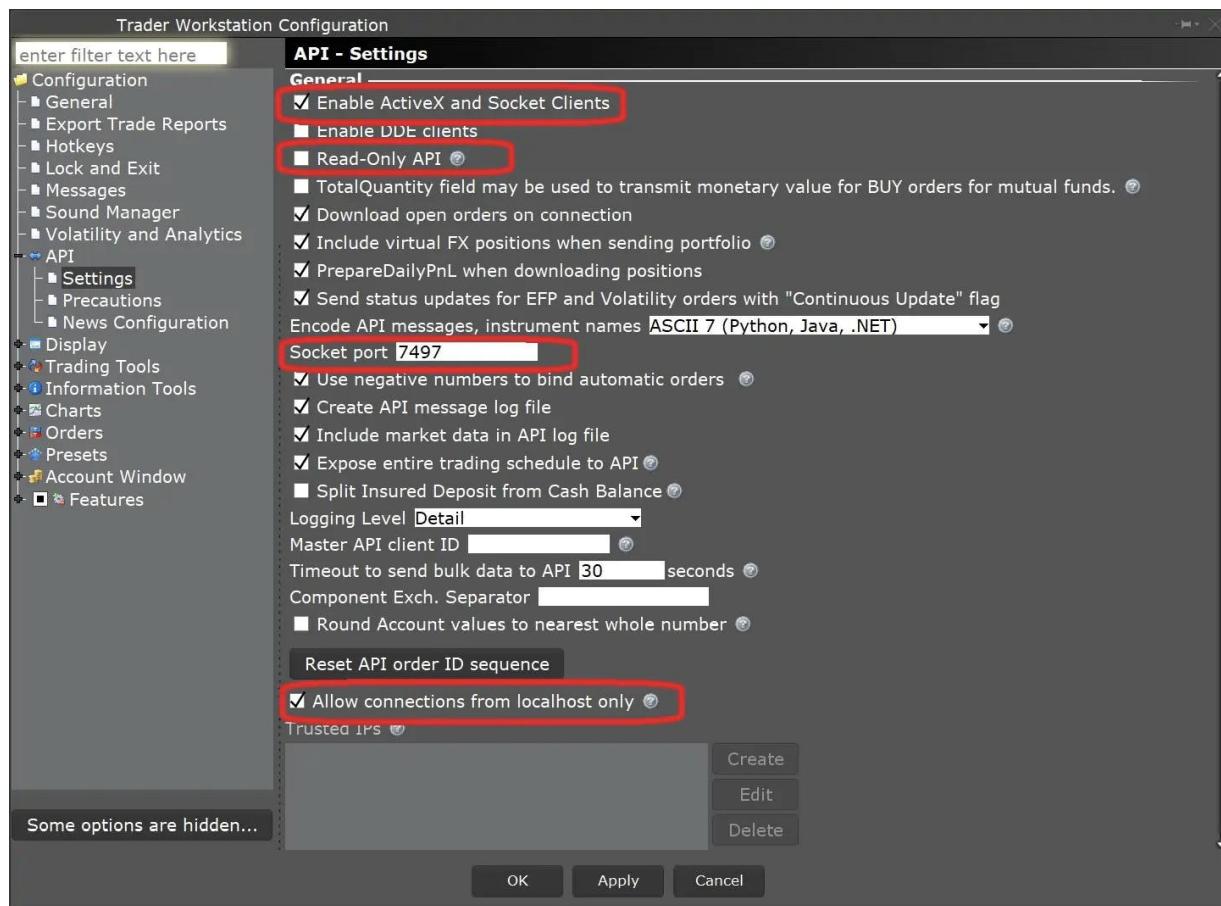


Figure 10.1: Global configuration settings

Make sure to check **Enable ActiveX and Socket Clients**. Check **Read-Only API** if you want extra protection against sending orders to IB. Lastly, check **Allow connections from localhost only** for security.

Make note of the **Socket port**, which you'll need to connect through Python. Depending on whether you logged in with a live or paper trading account, it's either **7497** or **7496**.

IMPORTANT NOTE

To work with the IB API live, you need an IBKR Pro account (not IBRK Lite). If you don't want (or can't) set up an IB account, you can use the free demo account. In the free demo account, you will be unable to download or stream market data.

Near the top-right corner of TWS, you should see a green button that says **DATA**. By clicking on it, you will see the data farms you're connected to. Note the empty section near the bottom titled **API Connections (listening on *:7497)**. Having this section means you've correctly enabled API connections.

Connections

Market Data Connections

Farm Name	Purpose	Status
usfarm	Market Data	connected
ushmds	HMDS	connected
secdefil	Aux Services	connected
usfuture	Market Data	connected
cashfarm	Market Data	connected
usfarm.nj	Market Data	connected

fundfarm	HMDS	connected
euhmds	HMDS	connected
usoft	Market Data	connected
cdc1.ibllc.com	Primary	connected

[More info](#)

API Connections (listening on *:7497)

Peer IP:port	Client ID	Status
--------------	-----------	--------

Reconnect All Farms

Redundant Backup Status

Site	Status
cdc1-hb1.ibllc.com	Accessible
cdc1-hb2.ibllc.com	Accessible

Last Login: Nov 10, 10:33

Close

Figure 10.2: Data connections screen showing empty API connections

Once we have TWS set up, you can install the IB API. Download the Python API from the Interactive Brokers GitHub page here: <http://interactivebrokers.github.io/>. Follow the instructions on the download page to download and install the latest stable version of the API for your operating system.

Once you have the IB API set up, create a new directory called `trading-app`. Inside, create the following Python script files:

- `__init__.py`
- `app.py`
- `client.py`
- `wrapper.py`
- `contract.py`
- `order.py`
- `utils.py`

How to do it...

We'll start by setting up the code we need to connect to TWS through the IB API.

1. Add the following code to the `client.py` file, which imports a base class from the IB API and implements a custom class we'll use to build our trading app:

```
from ibapi.client import EClient
class IBCClient(EClient):
    def __init__(self, wrapper):
        EClient.__init__(self, wrapper)
```

2. Add the following code to the `wrapper.py` file, which imports a base class from the IB API and implements a custom class we'll use to build our trading app:

```
from ibapi.wrapper import EWrapper
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
```

3. Add the following code to the `app.py` file, which implements a custom class that we'll develop into our trading app:

```
import threading
import time
from wrapper import IBWrapper
from client import IBCClient
class IBAppl(IBWrapper, IBCClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBCClient.__init__(self, wrapper=self)
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run, daemon=True)
        thread.start()
        time.sleep(2)
if __name__ == "__main__":
```

```
app = IBApp("127.0.0.1", 7497, client_id=10)
time.sleep(30)
app.disconnect()
```

How it works...

The architecture of the IB API operates on an asynchronous model, which is fundamental to understanding its interaction and data flow mechanisms. In this model, operations are not executed in a linear, blocking manner. Instead, actions are initiated by requests, and responses to these requests are handled via callbacks, making the system highly efficient and responsive.

Request-callback pattern

When we interact with the IB API, we typically start by sending a request for an action, such as retrieving market data or placing an order. This request is made through an instance of `EClient`, a component of the API responsible for initiating communication with the IB servers. The `EClient` sends the request and then continues with other tasks, rather than waiting for a response. This non-blocking pattern gets around the **Global Interpreter Lock (GIL)** of Python and exemplifies the asynchronous nature of the architecture.

The responses to these requests are not received directly by the `EClient`. Instead, they are handled by another component called `EWrapper`. `EWrapper` is essentially a set of callback functions that are triggered when responses or data from the IB server are received. For instance, when market data is requested, `EWrapper` will have a specific method that gets called when that market data arrives. The user implements these methods in `EWrapper` to define how the data or responses should be processed.

This separation of concerns, where `EClient` handles sending requests and `EWrapper` handles receiving responses, allows for a more organized and manageable code structure. It also enables the handling of multiple simultaneous data streams and requests efficiently. The asynchronous model ensures that the application remains responsive and can process incoming data or events as they occur, which is crucial for real-time trading applications where timely response to market changes is critical.

Inheritance and overriding

`EWrapper` is an interface class that defines a set of callback methods, which are intended to be overridden by our custom class that inherits from `EWrapper`. This inheritance allows us to define specific behaviors for each callback method, tailoring the response to various events, such as receiving market data or order updates. Similarly, `EClient` can be extended, or its methods overridden to customize the way requests to the IB server are made, providing flexibility and control over the interaction with the trading system.

Our trading app class

Our main trading app is a custom class that inherits our custom `IBWrapper` and `IBClient` classes (which in turn inherits the IB API's `EWrapper` and `EClient` classes). This class is designed to initialize and manage our connection to the IB API.

In the `__init__` method of `IBApp`, we initialize `IBWrapper` and `IBClient`, using `IBWrapper.__init__(self)` and `IBClient.__init__(self, wrapper=self)`. This sets up the environment and properties inherited from these parent classes. The `IBClient` initialization specifically requires a reference to an `IBWrapper` instance, which is provided by passing `self`.

Next, we call the `connect` method with the parameters `ip`, `port`, and `client_id`. This method is inherited from `IBClient` and is used to establish a connection to TWS. After we connect, a new thread is created and started using Python's `threading` module. This thread runs the `run` method in a daemon mode, which means it runs in the background and will automatically terminate when the main program exits. The `run` method is responsible for processing incoming messages and events from the IB API. Running this in a separate thread allows the `IBApp` to handle API messages asynchronously without blocking the main execution flow of the application.

There's more...

Now that we have the base code established, we can test our connection.

Using your terminal, run the `app.py` script:

```
python app.py
```

You will see a series of error messages printed on the screen. Don't worry, these are not actually errors. They are just messages indicating you've successfully connected to the IB data farms.

```
ERROR -1 2104 Market data farm connection is OK:usfarm.nj
ERROR -1 2104 Market data farm connection is OK:usfuture
ERROR -1 2104 Market data farm connection is OK:cashfarm
ERROR -1 2104 Market data farm connection is OK:usopt
ERROR -1 2104 Market data farm connection is OK:usfarm
ERROR -1 2106 HMDS data farm connection is OK:euhmdu
ERROR -1 2106 HMDS data farm connection is OK:fundfarm
ERROR -1 2106 HMDS data farm connection is OK:ushmdu
ERROR -1 2158 Sec-def data farm connection is OK:secdefil
```

Figure 10.3: Output from the IB API showing we successfully connected

IMPORTANT NOTE

Error code -1 is not actually an error. These are messages indicating successful connections to the IB data farms.

Also note in the TWS data connection screen, we now have a peer IP connected to TWS. That's the Python trading app we built!



Connections



Market Data Connections

Farm Name	Purpose	Status
usfarm	Market Data	connected
ushmds	HMDS	connected
secdefil	Aux Services	connected
usfuture	Market Data	connected
cashfarm	Market Data	connected
usfarm.nj	Market Data	connected

fundfarm	HMDS	connected
euhmds	HMDS	connected
usopt	Market Data	connected
cdc1.ibllc.com	Primary	connected

[More info](#)

API Connections (listening on *:7497)

Peer IP:port	Client ID	Status
127.0.0.1:58436	10	accepted

[Reconnect All Farms](#)

Redundant Backup Status

Site	Status
cdc1-hb1.ibllc.com	Accessible
cdc1-hb2.ibllc.com	Accessible

Last Login: Nov 10, 10:33

[Close](#)

Figure 10.4: TWS data connection screen showing our Python trading app connected to TWS

See also

The IB API has extensive documentation:

- Documentation for the initial setup: https://interactivebrokers.github.io/tws-api/initial_setup.html
- Documentation about the `EClient` and `EWrapper` classes: https://interactivebrokers.github.io/tws-api/client_wrapper.html
- Documentation about connectivity: <https://interactivebrokers.github.io/tws-api/connection.html>

Creating a Contract object with the IB API

When requesting market data or generating orders, we do it using the IB `Contract` object. An IB `Contract` contains all the information required for IB to correctly identify the instrument in question. Using one class, we can represent a broad spectrum of financial instruments, including stocks, options, futures, and more. In this recipe, we'll create an IB `Contract`.

The `Contract` class is used to define the specifications of a financial instrument that we might want to trade or query. The class has all the necessary details that uniquely identify a financial instrument across various asset classes, such as stocks, options, futures, forex, bonds, and more.

A key attribute of the `Contract` class is `conId` (contract ID), which is a unique identifier assigned by IB to each financial instrument. However, in many cases, we do not need to specify this ID directly. Instead, we typically provide other descriptive attributes that the IB system uses to uniquely identify the contract. These attributes typically include the following:

- `symbol`: The ticker symbol of the asset
- `secType`: Specifying the security type (e.g., `STK` for stock, `OPT` for option, or `FUT` for future)
- `expiry`: The expiration date for derivative instruments
- `strike`: The strike price for options
- `right`: Indicating the option type (call or put)
- `multiplier`: Defining the leverage or contract size
- `exchange`: The primary exchange where the asset is traded

For more complex instruments, additional attributes may be specified, such as `currency` for assets that trade in multiple currencies or on international markets. The `lastTradeDateOrContractMonth` attribute is used for futures and options to specify the contract month. For options and futures, `includeExpired` can be set to indicate whether expired contracts should be considered. The

`localSymbol` and `primaryExchange` attributes can provide more specific contract details, especially for instruments where the standard symbol may not be unique.

In this recipe, we'll create custom functions for a future, stock, and option contract.

IMPORTANT NOTE

All financial instruments are considered contracts by IB. This might come as a surprise when considering stocks or FX.

Getting ready...

We assume you've created the `contract.py` file in the `trading-app` directory. If not, do it now.

How to do it...

We'll create custom functions that create instances of different types of contracts, set the properties as relevant to the type of financial instrument, and return it:

Add the following code to the `contract.py` file:

```
from ibapi.contract import Contract
def future(symbol, exchange, contract_month):
    contract = Contract()
    contract.symbol = symbol
    contract.exchange = exchange
    contract.lastTradeDateOrContractMonth = contract_month
    contract.secType = "FUT"
    return contract
def stock(symbol, exchange, currency):
    contract = Contract()
    contract.symbol = symbol
    contract.exchange = exchange
    contract.currency = currency
    contract.secType = "STK"
    return contract
def option(symbol, exchange, contract_month, strike, right):
    contract = Contract()
    contract.symbol = symbol
    contract.exchange = exchange
    contract.lastTradeDateOrContractMonth = contract_month
    contract.strike = strike
    contract.right = right
    contract.secType = "OPT"
    return contract
```

How it works...

The functions are designed to encapsulate the common patterns when creating contracts. The `future` function creates a `contract` object representing a futures contract. The function takes three

parameters: `symbol`, `exchange`, and `contract_month`, which are required to set up the contract. When called, the function creates an instance of the `contract` class, which is a core component of the IB API for defining financial instruments. This instance, stored in the variable `contract`, is then configured with specific properties: the `symbol` of the futures contract, the `exchange` where it is traded, and the `lastTradeDateOrContractMonth`, which defines the expiration date. Additionally, the security type is set to `FUT`, explicitly marking the contract as a futures contract. The `stock` and `option` functions follow the same pattern.

There's more...

Now that we've defined our contract functions, let's use them. Open `app.py` and add a new import directly under the contract import:

```
from contract import stock, future, option
```

Then right under the instantiation of the app, add the following:

```
aapl = stock("AAPL", "SMART", "USD")
gbl = future("GBL", "EUREX", "202403")
ptr = option("PLTR", "BOX", "20240315", 20, "C")
```

The result of the changes is the following code in the `app.py` file:

```
import threading
import time
from wrapper import IBWrapper
from client import IBCient
from contract import stock, future, option
class IBApp(IBWrapper, IBCient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBCient.__init__(self, wrapper=self)
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
                                   daemon=True)
        thread.start()
        time.sleep(2)
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10)
    aapl = stock("AAPL", "SMART", "USD")
    gbl = future("GBL", "EUREX", "202403")
    ptr = option("PLTR", "BOX", "20240315", 20, "C")
    time.sleep(30)
    app.disconnect()
```

Running this code will create a stock, future, and option contract, wait for 30 seconds, and then disconnect.

See also

All financial instruments are considered contracts by the IB API. This includes stocks, ETFs, bonds, options, and futures. To read more about the Contract objects, see this URL:

<https://interactivebrokers.github.io/tws-api/contracts.html>

Creating an Order object with the IB API

Similar to how a `contract` object encapsulates all the information IB needs for financial instruments, `order` objects contain all the information required for IB to correctly place orders in the market. IB is famous for having dozens of order types, from simple market orders to advanced algorithms that slowly execute trades based on time or volume conditions. Key attributes of the `order` class include the following:

- `orderId`: A unique identifier for the order, typically assigned by the client
- `action`: Specifies the action type of the order, such as `BUY` or `SELL`
- `totalQuantity`: The amount of the asset to be bought or sold
- `orderType`: Defines the type of order, such as `LMT` for limit orders or `MKT` for market orders
- `lmtPrice`: The limit price for limit orders
- `auxPrice`: Used for stop or stop-limit orders to specify the stop price
- `tif`: *Time in force*, determining how long the order remains active
- `outsideRTH`: A boolean indicating whether the order can be executed outside regular trading hours
- `account`: Specifies the account the order should be executed from

Additional attributes can be set to further customize the order. These include attributes for advanced order types and conditions, such as `stopPrice` for stop orders, `trailStopPrice` for trailing stop orders, and various attributes for conditional orders. The `order` class also supports attributes for specifying commission, margin, and other trade-related parameters.

Getting ready...

We assume you've created the `order.py` file in the `trading-app` directory. If not, do it now.

How to do it...

We'll create custom functions that create instances of different types of contracts, set the properties relevant to the type of financial instrument, and return it.

Add the following code to the `order.py` file, which implements custom functions to create and return `order` objects using the IB API:

```
from ibapi.order import Order
BUY = "BUY"
SELL = "SELL"
def market(action, quantity):
    order = Order()
    order.action = action
    order.orderType = "MKT"
    order.totalQuantity = quantity
    return order
def limit(action, quantity, limit_price):
    order = Order()
    order.action = action
    order.orderType = "LMT"
    order.totalQuantity = quantity
    order.lmtPrice = limit_price
    return order
def stop(action, quantity, stop_price):
    order = Order()
    order.action = action
    order.orderType = "STP"
    order.auxPrice = stop_price
    order.totalQuantity = quantity
    return order
```

How it works...

A market order is executed immediately at the current market price. It prioritizes speed of execution over price, meaning it will be filled quickly but not necessarily at a specific price point. Market orders are commonly used in situations where the certainty of execution is more important than the exact price of the trade. We can define a market order object using the IB API by first instantiating an `order` object, then assigning either `BUY` or `SELL` to the `action` property, the quantity desired to the `totalQuantity` property, and `MKT` to the `orderType` property.

In similar fashion, we can create a limit order by specifying `LMT` as the `orderType` and a limit price as the `lmtPrice`. A limit order allows traders to specify the maximum price they are willing to pay when buying or the minimum price they are willing to accept when selling. Unlike market orders, limit orders provide price control but do not guarantee execution, as they will only be filled if the market price meets or is better than the limit price.

A stop loss order limits the loss on a position by automatically executing an order when the price reaches a specified stop price. We create a stop loss order by setting `orderType` to `STP` and setting `auxPrice` to the desired stop price.

There's more...

Now that we've defined our order functions, let's use them. Open `app.py` and add a new import directly under the contract import:

```
from order import limit, BUY
```

Then right under the creation of our contracts, add the following:

```
limit_order = limit(BUY, 100, 190.00)
```

The result of the changes is the following code in the `app.py` file:

```
import threading
import time
from wrapper import IBWrapper
from client import IBClient
from contract import stock, future, option
from order import limit, BUY
class IBApp(IBWrapper, IBClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
                                   daemon=True)
        thread.start()
        setattr(self, "thread", thread)
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10)
    aapl = stock("AAPL", "SMART", "USD")
    gbl = future("GBL", "EUREX", "202403")
    pltr = option("PLTR", "BOX", "20240315", 20, "C")
    limit_order = limit(BUY, 100, 190.00)
    time.sleep(30)
    app.disconnect()
```

Running this code will create a limit order object, wait for 30 seconds, and then disconnect. We'll look at how to submit orders in [Chapter 1, Manage Orders, Positions, and Portfolios with the IB API](#).

See also

To read more about the `order` object and the available order types, check the documentation here:

https://interactivebrokers.github.io/tws-api/available_orders.html

Fetching historical market data

Requesting historical market data using the IB API is an asynchronous process, emphasizing non-blocking, event-driven data retrieval. To kick off the process, we send a request for historical data by invoking the `reqHistoricalData` method, specifying parameters such as the financial instrument's identifier, the duration for which data is needed, the bar size, and the type of data required. Once the request is made, the IB API processes it and begins sending back the data. However, instead of

waiting for all data to be received before continuing with other tasks, the API employs a callback mechanism, specifically the `historicalData` method. This method is called asynchronously for each piece of data received from IB. Each invocation of `historicalData` provides a snapshot of market data for a specific time interval, which we can then process or store. In this recipe, we'll set up the code to request and receive historical market data.

Getting ready...

We assume you've created the `app.py`, `client.py`, and `wrapper.py` files in the `trading-app` directory. If not, do it now.

How to do it...

We'll update `app.py`, `client.py`, and `wrapper.py` to request historic market data.

1. Open `client.py` and add the following imports to the top of the file:

```
import time
import pandas as pd
```

2. Create a `list` as a constant after the imports. The strings in this list represent the columns of the DataFrame that we will populate with historical market data:

```
TRADE_BAR_PROPERTIES = ["time", "open", "high", "low",
    "close", "volume"]
```

3. Inside the `IBClient` class in the `client.py` file, add the following method:

```
def get_historical_data(self, request_id, contract, duration, bar_size):
    self.reqHistoricalData(
        reqId=request_id,
        contract=contract,
        endDateTime="",
        durationStr=duration,
        barSizeSetting=bar_size,
        whatToShow="MIDPOINT",
        useRTH=1,
        formatDate=1,
        keepUpToDate=False,
        chartOptions=[],
    )
    time.sleep(5)
    bar_sizes = ["day", "D", "week", "W", "month"]
    if any(x in bar_size for x in bar_sizes):
        fmt = "%Y%m%d"
    else:
        fmt = "%Y%m%d %H:%M:%S %Z"
    data = self.historical_data[request_id]
    df = pd.DataFrame(data, columns=TRADE_BAR_PROPERTIES)
    df.set_index(pd.to_datetime(df.time, format=fmt),
                 inplace=True)
    df.drop("time", axis=1, inplace=True)
```

```

df["symbol"] = contract.symbol
df.request_id = request_id
return df

```

4. Create a method that allows users to request data for more than one contract:

```

def get_historical_data_for_many(self, request_id,
                                 contracts, duration, bar_size,
                                 col_to_use="close"):
    dfs = []
    for contract in contracts:
        data = self.get_historical_data(
            request_id, contract, duration,
            bar_size)
        dfs.append(data)
        request_id += 1
    return (
        pd.concat(dfs)
        .reset_index()
        .pivot(
            index="time",
            columns="symbol",
            values=col_to_use
        )
    )

```

The result of the changes is the following code in `client.py`:

```

import time
import pandas as pd
from ibapi.client import EClient
TRADE_BAR_PROPERTIES = ["time", "open", "high", "low",
                        "close", "volume"]
class IBClient(EClient):
    def __init__(self, wrapper):
        EClient.__init__(self, wrapper)
    def get_historical_data(self, request_id,
                           contract, duration, bar_size):
        self.reqHistoricalData(
            reqId=request_id,
            contract=contract,
            endDateTime="",
            durationStr=duration,
            barSizeSetting=bar_size,
            whatToShow="MIDPOINT",
            useRTH=1,
            formatDate=1,
            keepUpToDate=False,
            chartOptions=[],
        )
        time.sleep(5)
        bar_sizes = ["day", "D", "week", "W",
                     "month"]
        if any(x in bar_size for x in bar_sizes):
            fmt = "%Y%m%d"
        else:
            fmt = "%Y%m%d %H:%M:%S %Z"
        data = self.historical_data[request_id]
        df = pd.DataFrame(data,
                          columns=TRADE_BAR_PROPERTIES)
        df.set_index(pd.to_datetime(df.time,
                                    format=fmt), inplace=True)
        df.drop("time", axis=1, inplace=True)
        df["symbol"] = contract.symbol

```

```

        df.request_id = request_id
        return df
    def get_historical_data_for_many(self,
                                    request_id, contracts, duration, bar_size,
                                    col_to_use="close"):
        dfs = []
        for contract in contracts:
            data = self.get_historical_data(
                request_id, contract, duration,
                bar_size)
            dfs.append(data)
            request_id += 1
        return (pd.concat(dfs)
            .reset_index()
            .pivot(
                index="time",
                columns="symbol",
                values=col_to_use
            )
        )

```

5. Next, open the **wrapper.py** file and add the following line of code to the **__init__** method in the **IBWrapper** class:

```
self.historical_data = {}
```

6. Then add the following method to the **IBWrapper** class:

```

def historicalData(self, request_id, bar):
    bar_data = (
        bar.date,
        bar.open,
        bar.high,
        bar.low,
        bar.close,
        bar.volume,
    )
    if request_id not in self.historical_data.keys():
        self.historical_data[request_id] = []
    self.historical_data[request_id].append(bar_data)

```

The result of the changes is the following code in **wrapper.py**:

```

from ibapi.wrapper import EWrapper
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
        self.historical_data = {}
    def historicalData(self, request_id, bar):
        bar_data = (
            bar.date,
            bar.open,
            bar.high,
            bar.low,
            bar.close,
            bar.volume,
        )
        if request_id not in self.historical_data.keys():
            self.historical_data[request_id] = []
        self.historical_data[request_id].append(
            bar_data)

```

How it works...

Getting historical data through the IB API clearly demonstrates the request-callback architecture. We get historical data by first requesting it, then processing it through a callback.

Requesting historic data

The `get_historical_data` method accepts four parameters: `request_id`, `contract`, `duration`, and `bar_size`. `request_id` is an arbitrary, but unique identifier for the data request, `contract` specifies the financial instrument for which historical data is being requested, `duration` indicates the time span for the historical data, and `bar_size` defines the granularity of the data.

The valid duration strings are as follows:

Unit	Description
S	Seconds
D	Day
W	Week
M	Month
Y	Year

Figure 10.5: Valid duration strings for requesting historical data

The valid bar size strings are as follows:

Size	1 secs	5 secs	10 secs	15 secs	30 secs	1 min	2 mins	3 mins	5 mins	10 mins	15 mins	20 mins	30 mins
1 hour	2 hours	3 hours	4 hours	8 hours									
1 day													
1 week													
1 month													

Figure 10.6: Valid bar sizes for requesting historical data

The method begins by calling `reqHistoricalData`, which is a method provided by the IB API to request historical data. The parameters passed to this function include `request_id`, the `contract` object, an empty string for `endDateTime` (indicating the request is for the most recent data up to the

current date), the `duration` of historical data, `bar_size`, and other settings such as `whatToShow` (indicating the type of data required), `useRTH` (data within regular trading hours), `formatDate` (the format of the returned date), and `keepUpToDate` (indicating no need for live updates). These are hardcoded in our implementation, but you can modify them to allow the user to pass their own options.

The available kinds of historic data type strings (`whatToShow`) are as follows:

Type	Open	High	Low	Close	Volume
<code>TRADES</code>	First traded price	Highest traded price	Lowest traded price	Last traded price	Total traded volume
<code>MIDPOINT</code>	Starting midpoint price	Highest midpoint price	Lowest midpoint price	Last midpoint price	N/A
<code>BID</code>	Starting bid price	Highest bid price	Lowest bid price	Last bid price	N/A
<code>ASK</code>	Starting ask price	Highest ask price	Lowest ask price	Last ask price	N/A
<code>BID_ASK</code>	Time average bid	Max Ask	Min Bid	Time average ask	N/A
<code>ADJUSTED_LAST</code>	Dividend-adjusted first traded price	Dividend-adjusted high trade	Dividend-adjusted low trade	Dividend-adjusted last trade	Total traded volume
<code>HISTORICAL_VOLATILITY</code>	Starting volatility	Highest volatility	Lowest volatility	Last volatility	N/A
<code>OPTION_IMPLIED_VOLATILITY</code>	Starting implied volatility	Highest implied volatility	Lowest implied volatility	Last implied volatility	N/A
<code>FEE_RATE</code>	Starting fee rate	Highest fee rate	Lowest fee rate	Last fee rate	N/A
<code>YIELD_BID</code>	Starting bid yield	Highest bid yield	Lowest bid yield	Last bid yield	N/A
<code>YIELD_ASK</code>	Starting ask yield	Highest ask yield	Lowest ask yield	Last ask yield	N/A
<code>YIELD_BID_ASK</code>	Time average bid yield	Highest ask yield	Lowest bid yield	Time average ask yield	N/A
<code>YIELD_LAST</code>	Starting last yield	Highest last yield	Lowest last yield	Last last yield	N/A
<code>SCHEDULE</code>	N/A	N/A	N/A	N/A	N/A
<code>AGGTRADES</code>	First traded price	Highest traded price	Lowest traded price	Last traded price	Total traded volume

Figure 10.7: Valid data types to return for historical data

After sending the request, the method pauses execution for five seconds to allow time for the data to be retrieved and stored in the `historical_data` dictionary, keyed by `request_id`.

IMPORTANT NOTE

The five-second delay is arbitrary and does not guarantee that all data has been received. You can extend this delay or implement more sophisticated methods to check whether data exists in the `historicData` method in the `IBWrapper` class.

The method then determines how to parse the incoming time string depending on whether the requested data is intraday.

Next, the method retrieves the historical data and creates a pandas DataFrame from it. The DataFrame is structured with columns defined by `TRADE_BAR_PROPERTIES`. The index of the DataFrame is set to the time column, which is formatted, and the original time column is dropped. Additionally, a new `symbol` column is added to the DataFrame, containing the symbol of the contract, and `request_id` is stored as an attribute of the DataFrame.

Receiving historic data

The `historicalData` method is a callback and is automatically invoked for each bar of data received from IB following a call to `reqHistoricalData`. When `reqHistoricalData` is called to request historical market data, the IB API responds by sending back this data in individual bar units, with each bar triggering the `historicalData` method. This method then processes and stores each piece of data.

`historicalData` receives two parameters: `request_id` and `bar`. The `request_id` parameter is the unique identifier associated with the specific request for historical data, which allows the method to differentiate between data from different requests. The `bar` parameter is an object that contains the historical bar data for the instrument. Within the method, the bar data is structured into a tuple that contains the date (`bar.date`), the opening price (`bar.open`), the highest price (`bar.high`), the lowest price (`bar.low`), the closing price (`bar.close`), and the trading volume (`bar.volume`) for that specific time period.

The method then checks whether the `request_id` already exists as a key in the `historical_data` dictionary. If it does not exist the method initializes an empty list under this key. This step ensures that there is a structure in place to store the historical data for each unique request.

Finally, the method appends the `bar_data` tuple to the list associated with the `request_id` in the `historical_data` dictionary. This action effectively stores the historical bar data, allowing the method to accumulate the historical data points for each request over time.

The `get_historical_data_for_many` method loops through each provided contract, requests the historical data, and concatenates their resulting DataFrames together. The method returns a pivoted DataFrame, which moves the data in `col_to_use` into the rows and each requested contract into the columns.

There's more...

We now have the code built to request and receive historical market data. Open the `app.py` file and add the following code block under the definitions of the contracts:

```
data = app.get_historical_data(  
    request_id=99,  
    contract=aapl,  
    duration='2 D',  
    bar_size='30 secs'  
)
```

The result of the changes is the following code in `app.py`:

```
import threading  
import time
```

```

from wrapper import IBWrapper
from client import IBClient
from contract import stock, future, option
class IBApp(IBWrapper, IBClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
                                   daemon=True)
        thread.start()
        time.sleep(2)
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10)
    aapl = stock("AAPL", "SMART", "USD")
    gbl = future("GBL", "EUREX", "202403")
    pltr = option("PLTR", "BOX", "20240315", 20, "C")
    data = app.get_historical_data(
        request_id=99,
        contract=aapl,
        duration='2 D',
        bar_size='30 secs'
    )
    time.sleep(30)
    app.disconnect()

```

After running this code, `data` will contain a pandas DataFrame with the requested historic market data:

		open	high	low	close	volume	symbol
time							
2023-11-14	09:30:00-05:00	187.69	187.73	187.16	187.47	1168171.0	AAPL
2023-11-14	09:30:30-05:00	187.45	187.51	187.28	187.29	210746.0	AAPL
2023-11-14	09:31:00-05:00	187.28	187.48	187.21	187.23	229337.0	AAPL
2023-11-14	09:31:30-05:00	187.24	187.24	186.73	186.73	197331.0	AAPL
2023-11-14	09:32:00-05:00	186.74	186.90	186.69	186.83	163242.0	AAPL
...	
2023-11-15	15:03:30-05:00	188.41	188.44	188.39	188.40	29656.0	AAPL
2023-11-15	15:04:00-05:00	188.40	188.42	188.34	188.34	15333.0	AAPL
2023-11-15	15:04:30-05:00	188.36	188.36	188.30	188.30	16014.0	AAPL
2023-11-15	15:05:00-05:00	188.31	188.34	188.28	188.34	35756.0	AAPL
2023-11-15	15:05:30-05:00	188.33	188.36	188.32	188.32	15280.0	AAPL
[1452 rows x 6 columns]							

Figure 10.8: A pandas DataFrame containing requested historic market data

TIP

If you're using an **Interactive Development Environment (IDE)** such as **PyCharm** or **VSCode**, you can execute the code in debug mode during development. Debug mode lets you pause the execution of the code to inspect the variables.

See also

To learn more about historical market data, see the documentation at this URL:
https://interactivebrokers.github.io/tws-api/historical_data.html. For specifics on the following topics, see the associated documentation:

- Requesting historical market data: https://interactivebrokers.github.io/tws-api/historical_bars.html#hd_request
- Receiving historical market data: https://interactivebrokers.github.io/tws-api/historical_bars.html#hd_receive
- Duration, bar sizes, historical data types available, and available data per instrument: https://interactivebrokers.github.io/tws-api/historical_bars.html#hd_duration

Getting a market data snapshot

In the previous recipe, we learned how to get historical data. In some situations, we may need the current market price. In [Chapter 12, Deploy Strategies to a Live Environment](#), we'll use the current market price to create methods to target specific values or percentage allocations in our portfolio.

The API uses tick types, each representing a specific category of market data, such as last trade price, volume, or bid and ask. These tick types let us access real-time pricing information, which is important for making informed trading decisions. This recipe will show you how to get real-time market data.

Getting ready...

We assume you've created the `app.py`, `client.py`, and `wrapper.py` files in the `trading-app` directory. If not, do it now.

How to do it...

We'll update `app.py`, `client.py`, and `wrapper.py` to request the last price for a contract.

1. Open `client.py` and include the following method in the `IBClient` class after the `get_historical_data_for_many` method:

```
def get_market_data(self, request_id, contract,
                    tick_type=4):
    self.reqMktData(
        reqId=request_id,
        contract=contract,
        genericTickList="",
        snapshot=True,
        regulatorySnapshot=False,
        mktDataOptions=[]
    )
```

```

        time.sleep(5)
        self.cancelMktData(reqId=request_id)
        return self.market_data[request_id].get(tick_type)
    
```

2. Open `wrapper.py` and include the following method in the `IBWrapper` class after the `historicalData` method:

```

def tickPrice(self, request_id, tick_type, price,
             attrib):
    if request_id not in self.market_data.keys():
        self.market_data[request_id] = {}
        self.market_data[request_id][tick_type] =
            float(price)
    
```

3. Add an instance variable in the `__init__` method under `self.historical_data = {}`:

```
self.market_data = {}
```

How it works...

The `get_market_data` method fetches a specific tick type from the IB API. It initiates a market data request for a given financial contract, which we identify by `request_id`, and requests all available ticks by using an empty string for the `genericTickList` argument. The function opts for a single data snapshot, rather than a continuous stream and pauses for five seconds to allow data reception and processing. After the pause, it cancels the data request to free up the request ID. Finally, the function retrieves and returns the specific market data from a dictionary using the request ID and tick type as keys. We default to tick type **4**, which is the last traded price.

In the `tickPrice` callback, it first checks whether the `request_id` exists in the `market_data` dictionary and if not, it initializes an empty dictionary for that ID. It then updates the dictionary, setting the `tick_type` key to the received `price`, converted to a float.

The `tickPrice` method will typically retrieve seven different tick types:

Code	Tick Name	Description
1	Bid Price	Highest priced bid for the contract.
2	Ask Price	Lowest price offer on the contract.
4	Last Price	Last price at which the contract traded.
6	High	High price for the day.
7	Low	Low price for the day.
9	Close Price	The last available closing price for the previous day.
14	Open Tick	Current session's opening price .

Figure 10.9: Different tick types returned by the IB API

There's more...

To get the last closing price for AAPL, add the following code to `app.py` after we define our AAPL contract:

```
data = app.get_market_data(  
    request_id=99,  
    contract=aapl  
)
```

The `data` variable contains a float representing the last traded price of AAPL.

See also

To learn more about how to request and receive market data using the IB API, see the following resources:

- A list of different tick types: https://interactivebrokers.github.io/tws-api/tick_types.html
- Requesting market data: https://interactivebrokers.github.io/tws-api/md_request.html
- Receiving market data: https://interactivebrokers.github.io/tws-api/md_receive.html

Streaming live market data

Requesting tick-by-tick data involves a real-time, granular approach to market data acquisition. This process begins by invoking the `reqTickByTickData` method, where we specify the unique request identifier, the financial instrument's contract details, and the type of tick data we are interested in (such as `BidAsk`). Upon this request, the IB API starts transmitting data for each individual market **tick**, which is a single change or update in market data.

Unlike bulk historical data retrieval, this method provides data almost instantaneously as market events occur, which lets us build near real-time algorithmic trading applications. The received data is handled through a callback function that is triggered for each tick, capturing detailed information such as price, size, and the time of the tick. By the end of this receipe, you'll be able to stream near-real-time market data from the IB API.

Getting ready...

We assume you've created the `app.py`, `client.py`, and `wrapper.py` files in the `trading-app` directory. If not, do it now.

How to do it...

We'll update `app.py`, `client.py`, and `wrapper.py` to request historic market data.

1. Open `client.py` and include the following import at the top of the file:

```
from dataclasses import dataclass, field
```

2. Then add a `dataclass` to represent each price tick below the `TRADE_BAR_PROPERTIES` constant:

```
@dataclass
class Tick:
    time: int
    bid_price: float
    ask_price: float
    bid_size: float
    ask_size: float
    timestamp_: pd.Timestamp = field(init=False)
    def __post_init__(self):
        self.timestamp_ = pd.to_datetime(self.time,
                                         unit="s")
        self.bid_price = float(self.bid_price)
        self.ask_price = float(self.ask_price)
        self.bid_size = int(self.bid_size)
        self.ask_size = int(self.ask_size)
```

3. Inside the `IBClient` class in the `client.py` file, add the functions that will start and stop the streaming data:

```
def get_streaming_data(self, request_id, contract):
    self.reqTickByTickData(
        reqId=request_id,
        contract=contract,
        tickType="BidAsk",
        numberOfTicks=0,
        ignoreSize=True
    )
    time.sleep(10)
    while True:
        if self.stream_event.is_set():
            yield Tick(
                *self.streaming_data[request_id])
            self.stream_event.clear()
def stop_streaming_data(self, request_id):
    self.cancelTickByTickData(reqId=request_id)
```

The result of the changes is the following code in `client.py`:

```
import time
import pandas as pd
from dataclasses import dataclass, field
from ibapi.client import EClient
TRADE_BAR_PROPERTIES = ["time", "open", "high", "low",
                        "close", "volume"]
@dataclass
class Tick:
    time: int
    bid_price: float
    ask_price: float
    bid_size: float
    ask_size: float
    timestamp_: pd.Timestamp = field(init=False)
    def __post_init__(self):
        self.timestamp_ = pd.to_datetime(self.time,
                                         unit="s")
```

```

        self.bid_price = float(self.bid_price)
        self.ask_price = float(self.ask_price)
        self.bid_size = int(self.bid_size)
        self.ask_size = int(self.ask_size)
    class IBClient(EClient):
        def __init__(self, wrapper):
            EClient.__init__(self, wrapper)
        <snip>
        def get_streaming_data(self, request_id, contract):
            self.reqTickByTickData(
                reqId=request_id,
                contract=contract,
                tickType="BidAsk",
                numberofticks=0,
                ignoreSize=True
            )
            time.sleep(10)
            while True:
                if self.stream_event.is_set():
                    yield Tick(
                        *self.streaming_data[request_id])
                    self.stream_event.clear()
        def stop_streaming_data(self, request_id):
            self.cancelTickByTickData(reqId=request_id)

```

4. Open **wrapper.py** and include the following import at the top of the file:

```
import threading
```

5. Add the following lines of code to the **__init__** method in the **IBWrapper** class:

```

    self.streaming_data = {}
    self.stream_event = threading.Event()

```

6. Next, add the following method to the **IBWrapper** class:

```

    def tickByTickBidAsk(
        self,
        request_id,
        time,
        bid_price,
        ask_price,
        bid_size,
        ask_size,
        tick_attrib_last
    ):
        tick_data = (
            time,
            bid_price,
            ask_price,
            bid_size,
            ask_size,
        )
        self.streaming_data[request_id] = tick_data
        self.stream_event.set()

```

The result of the changes is the following code in **wrapper.py**:

```

import threading
from ibapi.wrapper import EWrapper
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)

```

```

        self.nextValidOrderId = None
        self.historical_data = {}
        self.streaming_data = {}
        self.stream_event = threading.Event()
    <snip>
    def tickByTickBidAsk(
            self,
            request_id,
            time,
            bid_price,
            ask_price,
            bid_size,
            ask_size,
            tick_attrib_last
    ):
        tick_data = (
            time,
            bid_price,
            ask_price,
            bid_size,
            ask_size,
        )
        self.streaming_data[request_id] = tick_data
        self.stream_event.set()

```

How it works...

Getting streaming data through the IB API requires two steps: first we request it, then we process it through a callback. The tick-by-tick data we stream is the data that corresponds to the data shown in the TWS Time and Sales window.

IMPORTANT NOTE

Time and Sales refers to a real-time data feed that provides detailed information about executed trades for a specific financial instrument. This feed includes the exact time of each trade, the price at which it was executed, and the size of the trade. Time and Sales data is useful for traders who want to analyze order book dynamics in real time.

Requesting streaming data

We start by defining a Python **dataclass** called `tick`. A Python **dataclass** is a decorator that automatically generates special methods such as `__init__`, `__repr__`, and `__eq__` for classes, primarily used for storing data, simplifying, and reducing boilerplate code.

The `tick` **dataclass** represents market data ticks with attributes for time, bid and ask prices, and sizes. An additional attribute, `timestamp_`, is defined as a pandas `Timestamp` and excluded from automatic `__init__` method generation using `field(init=False)`. The `__post_init__` method of the **dataclass** converts the `time` attribute from a Unix timestamp to a more readable pandas `Timestamp` format, using `to_datetime`.

The `get_streaming_data` method requires a request ID and a contract to start streaming data. The method requests and yields real-time bid and ask tick data for a specified financial instrument, represented by the `contract` parameter.

The method begins by calling `reqTickByTickData`, a function of the IB API that requests real-time tick-by-tick data. The parameters passed to this function include the following:

- `reqId`: A unique identifier for the data request.
- `contract`: Specifies the financial instrument for which data is being requested.
- `tickType`: Specifies the type of data to stream. Available options are `Last`, `AllLast`, `BidAsk`, and `MidPoint`.
- `numberOfTicks`: A flag to request a specific set of historic ticks or to stream until canceled.
- `ignoreSize`: Whether updates to bid and ask sizes are required.

Following the data request, we delay execution for 10 seconds to ensure that some data is received and buffered before the method starts yielding data.

The method then enters an infinite loop, continuously checking if the thread event `stream_event` is set and when it is, the method yields the `tick` object with the latest data for the given `request_id`, and then clears the event, readying it for the next batch of data.

IMPORTANT NOTE

A Python generator is a special type of function that returns an iterator, allowing for the generation of items on the fly rather than storing them all in memory at once. When a generator function is called, it doesn't execute its code immediately. Instead, it returns a generator object that can be iterated over. Each iteration over a generator object resumes the function's execution from where it last yielded a value, using the `yield` statement, until it either encounters another `yield` or reaches the end of the function, at which point it raises a `StopIteration` exception.

Receiving streaming data

Depending on the `tickType` requested in the `reqTickByTickData`, IB will use a different callback function. They are as follows:

tickType	Callback
<code>Last</code>	<code>tickByTickAllLast</code>
<code>AllLast</code>	<code>tickByTickAllLast</code>
<code>BidAsk</code>	<code>tickByTickBidAsk</code>
<code>MidPoint</code>	<code>tickByTickMidPoint</code>

Figure 10.10: Callbacks used for each tick type requested

IMPORTANT NOTE

We only implement `tickByTickBidAsk` to respond to the `BidAsk` tick type. You can read more about the type of data that is returned in each of the callbacks at this URL: https://interactivebrokers.github.io/tws-api/tick_data.html

We override the `tickByTickBidAsk` provided by the IB API. This callback method is invoked for every tick. Upon invocation, the method first constructs a tuple, `tick_data`, comprising the time, bid price, ask price, bid size, and ask size.

After receiving a tick, the method constructs a `tick_data` tuple with the bid and ask prices and respective volumes. The method then updates a dictionary `streaming_data`, keyed by `request_id`, with this tick data and sets a `stream_event` threading event, signaling that new data has been received and is ready for processing in the `get_streaming_data` method.

There's more...

We now have the code built to request and receive streaming tick data. Open the `app.py` file and add the following code block under the definitions of the contracts:

```
eur = future("EUR", "CME", "202312")
for tick in app.get_streaming_data(99, es):
    print(tick)
```

The result of the changes is the following code in `app.py`:

```
import threading
import time
from wrapper import IBWrapper
from client import IBCClient
from contract import future
class IBApp(IBWrapper, IBCClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBCClient.__init__(self, wrapper=self)
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
                                  daemon=True)
        thread.start()
        time.sleep(2)
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10)
    eur = future("EUR", "CME", "202312")
    for tick in app.get_streaming_data(99, eur):
        print(tick)
    time.sleep(30)
    app.disconnect()
```

After running this code, you'll see a list of `Tick` objects printed to the console with the details of each tick:

```
Tick(time=1700522344, bid_price=1.0956, ask_price=1.09565, bid_size=Decimal('2'), ask_size=Decimal('38'), timestamp_=Timestamp('2023-11-20 23:19:04'))
Tick(time=1700522353, bid_price=1.09555, ask_price=1.09565, bid_size=Decimal('65'), ask_size=Decimal('22'), timestamp_=Timestamp('2023-11-20 23:19:13'))
Tick(time=1700522355, bid_price=1.0956, ask_price=1.09565, bid_size=Decimal('2'), ask_size=Decimal('35'), timestamp_=Timestamp('2023-11-20 23:19:15'))
Tick(time=1700522361, bid_price=1.09555, ask_price=1.09565, bid_size=Decimal('55'), ask_size=Decimal('32'), timestamp_=Timestamp('2023-11-20 23:19:21'))
Tick(time=1700522361, bid_price=1.0956, ask_price=1.09565, bid_size=Decimal('2'), ask_size=Decimal('32'), timestamp_=Timestamp('2023-11-20 23:19:21'))
Tick(time=1700522400, bid_price=1.09555, ask_price=1.09565, bid_size=Decimal('57'), ask_size=Decimal('26'), timestamp_=Timestamp('2023-11-20 23:20:00'))
Tick(time=1700522403, bid_price=1.0956, ask_price=1.09565, bid_size=Decimal('4'), ask_size=Decimal('29'), timestamp_=Timestamp('2023-11-20 23:20:03'))
Tick(time=1700522410, bid_price=1.09555, ask_price=1.09565, bid_size=Decimal('71'), ask_size=Decimal('32'), timestamp_=Timestamp('2023-11-20 23:20:10'))
Tick(time=1700522416, bid_price=1.0956, ask_price=1.09565, bid_size=Decimal('4'), ask_size=Decimal('33'), timestamp_=Timestamp('2023-11-20 23:20:16'))
```

Figure 10.11: Streaming tick data as Tick objects

See also

To learn more about the available types of streaming data, check the following URL:

https://interactivebrokers.github.io/tws-api/market_data.html. For specifics, check the following URLs:

- Level 1 (top of book) streaming and snapshot data. This is another option for streaming real-time data that was not covered in this recipe: https://interactivebrokers.github.io/tws-api/md_request.html.
- Streaming tick data covered in this recipe: https://interactivebrokers.github.io/tws-api/tick_data.html.

Storing live tick data in a local SQL database

We discussed storing financial market data in [Chapter 4, Store Financial Market Data on Your Computer](#). In this chapter, we learned several ways of downloading different types of market data for free and storing it in a variety of formats. In [Chapter 13, Advanced Recipes for Market Data and Strategy Management](#), we'll look at how to use the cutting-edge ArcticDB library to store petabytes of market data. For now, we'll extend the recipes in this chapter and the recipes in [Chapter 4](#) to store the streaming tick data in a local SQLite database.

Getting ready...

We assume you've read the *Storing Data On-Disk with SQLite* recipe in [Chapter 4, Store Financial Market Data on Your Computer](#). If you haven't, please review it now. We also assume you've created the `app.py`, `client.py`, `wrapper.py`, and `utils.py` files in the `trading-app` directory. If not, do it now.

We'll also move some of the code currently in `client.py`, to the `utils.py` file. Copy the following code from `client.py` and paste it into `utils.py`:

```
import pandas as pd
from dataclasses import dataclass, field
TRADE_BAR_PROPERTIES = ["time", "open", "high", "low",
    "close", "volume"]
@dataclass
class Tick:
    time: int
    bid_price: float
    ask_price: float
    bid_size: float
    ask_size: float
    timestamp_: pd.Timestamp = field(init=False)
    def __post_init__(self):
        self.timestamp_ = pd.to_datetime(self.time,
            unit="s")
```

```

        self.bid_price = float(self.bid_price)
        self.ask_price = float(self.ask_price)
        self.bid_size = int(self.bid_size)
        self.ask_size = int(self.ask_size)

```

Now, remove the code from the `client.py` file and import it from `utils.py`. The result of the changes is the following code in `client.py`:

```

import time
import pandas as pd
from utils import Tick, TRADE_BAR_PROPERTIES
from ibapi.client import EClient
class IBClient(EClient):
    <snip>

```

How to do it...

We'll add methods to our `IBApp` class that will create the SQLite database, generate a connection to our database, and insert tick data into the table.

1. Add the `connection` method to the `IBApp` class in `app.py` and decorate it with the `property` decorator:

```

@property
def connection(self):
    return sqlite3.connect("tick_data.sqlite",
                           isolation_level=None)

```

2. Add the `create_table` method to the `IBApp` class:

```

def create_table(self):
    cursor = self.connection.cursor()
    cursor.execute(
        "CREATE TABLE IF NOT EXISTS bid_ask_data (
            timestamp datetime, symbol string,
            bid_price real, ask_price real,
            bid_size integer, ask_size integer)")

```

3. Add the `stream_to_sqlite` method to the `IBApp` class. Don't worry, we'll go through it in detail next:

```

def stream_to_sqlite(self, request_id, contract,
                     run_for_in_seconds=23400):
    cursor = self.connection.cursor()
    end_time = time.time() + run_for_in_seconds + 10
    for tick in app.get_streaming_data(request_id,
                                       contract):
        query = "INSERT INTO bid_ask_data (
            timestamp, symbol, bid_price,
            ask_price, bid_size, ask_size)
            VALUES (?, ?, ?, ?, ?, ?)"
        values = (
            tick.timestamp_.strftime(
                "%Y-%m-%d %H:%M:%S"),
            contract.symbol,
            tick.bid_price,
            tick.ask_price,
            tick.bid_size,
            tick.ask_size

```

```

        )
        cursor.execute(query, values)
        if time.time() >= end_time:
            break
    self.stop_streaming_data(request_id)

```

4. In the `__init__` method of the `IBApp` class, add the following line under the calls to `IBWrapper` and `IBClient`:

```
    self.create_table()
```

The result of the changes is the following code in `IBApp`:

```

class IBApp(IBWrapper, IBClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.create_table()
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
                                   daemon=True)
        thread.start()
        time.sleep(2)
    @property
    def connection(self):
        return sqlite3.connect("tick_data.sqlite",
                               isolation_level=None)
    def create_table(self):
        cursor = self.connection.cursor()
        cursor.execute(
            "CREATE TABLE IF NOT EXISTS bid_ask_data (
                timestamp datetime, symbol string,
                bid_price real, ask_price real,
                bid_size integer, ask_size integer)")
    def stream_to_sqlite(self, request_id, contract,
                        run_for_in_seconds=23400):
        cursor = self.connection.cursor()
        end_time = time.time() + run_for_in_seconds + 10
        for tick in app.get_streaming_data(request_id,
                                           contract):
            query = "INSERT INTO bid_ask_data (
                timestamp, symbol, bid_price,
                ask_price, bid_size, ask_size) VALUES (
                ?, ?, ?, ?, ?, ?)"
            values = (
                tick.timestamp_.strftime(
                    "%Y-%m-%d %H:%M:%S"),
                contract.symbol,
                tick.bid_price,
                tick.ask_price,
                tick.bid_size,
                tick.ask_size
            )
            cursor.execute(query, values)
            if time.time() >= end_time:
                break
        self.stop_streaming_data(request_id)

```

How it works...

We first create a class property called `connection`, which establishes and returns a connection to our SQLite database. The `@property` decorator is used to create a property, allowing the `connection` method to be accessed like an attribute without the need to call it as a method.

When the `connection` property is accessed, it creates a connection to an SQLite database file named `tick_data.sqlite`. The `isolation_level=None` parameter sets the transaction isolation level to `None`, which means that the `autocommit` mode is enabled. In `autocommit` mode, changes to the database are committed immediately after each statement, without needing to explicitly call `commit` after each database operation.

The `create_table` method creates a database table named `bid_ask_data`. When this method is called, it first establishes a database cursor, which is an intermediary for executing database commands. The code then executes a `create` SQL command using the `execute` method, which creates the table if it does not exist. When `IBApp` is initialized, we call `self.create_table`, which creates the table if it does not exist.

The `stream_to_sqlite` method stores the tick data in our SQLite database for a specified duration. It starts by creating a database cursor from the established SQLite connection. The method calculates an `end_time` value by adding the specified `run_for_in_seconds` duration (defaulting to 23,400 seconds, or 6.5 hours) to the current time, plus an additional 10 second buffer which makes up for the time we wait for the tick stream to start. It then enters a loop, retrieving streaming data from the `get_streaming_data` method, which yields `Tick` objects containing market data for the given `request_id` and `contract`. For each tick received, the method constructs a SQL insert query to add the tick data into the `bid_ask_data` table, formatting the timestamp and including relevant data such as the contract's symbol, bid price, ask price, bid size, and ask size. The loop continues until the current time exceeds the calculated `end_time`, at which point it breaks, limiting the data streaming to the specified duration.

There's more...

We now have the code built to store streaming tick data. Add the following code block under the definitions of the contracts:

```
es = future("ES", "CME", "202312")
app.stream_to_sqlite(99, es, run_for_in_seconds=30)
```

The result of the changes is the following code in `app.py`:

```
<snip>
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10)
```

```

es = future("ES", "CME", "202312")
app.stream_to_sqlite(99, es, run_for_in_seconds=30)
app.disconnect()

```

We can inspect the data stored in our SQLite database with DB Browser for SQLite. It's a free tool that gives us a graphical interface to inspect data in SQLite databases. You can download it here: <https://sqlitebrowser.org/>. Once installed, open up the `tick_data.sqlite` file and navigate to the **Browse Data** tab and you'll see the tick data.

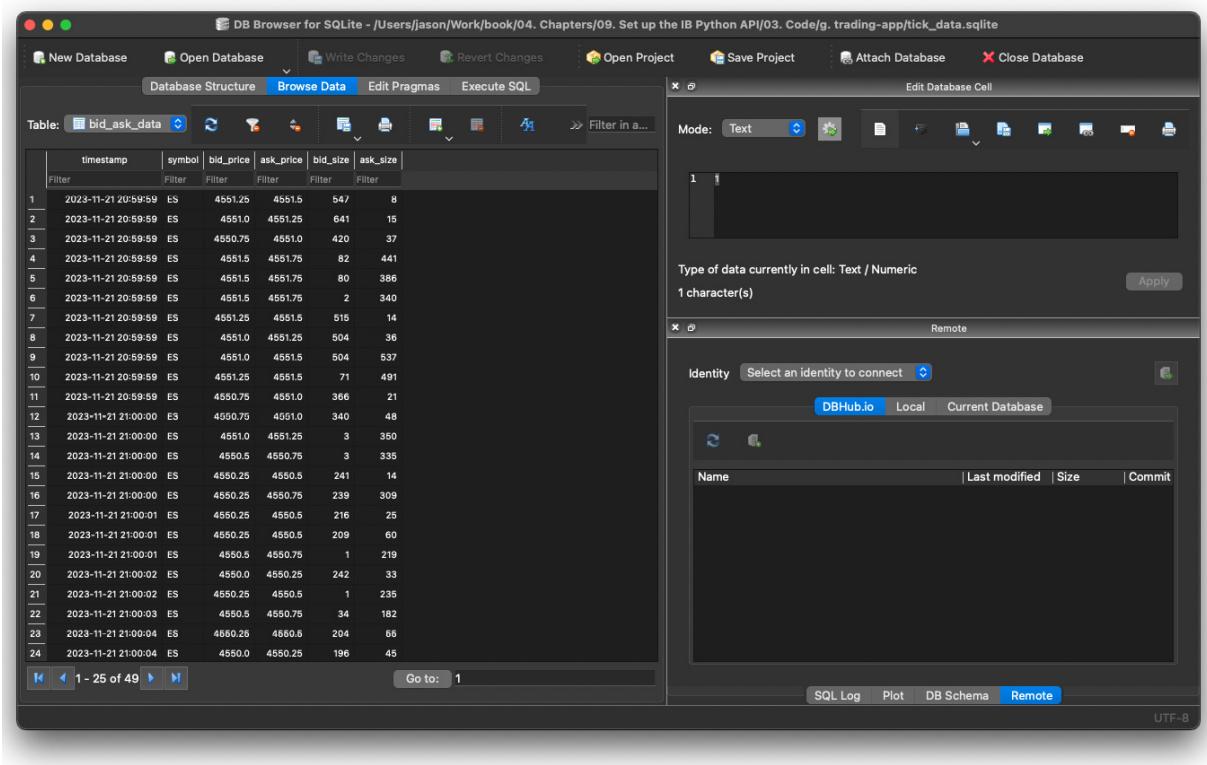


Figure 10.12: Browsing bid ask tick data in our SQLite database

See also

The SQLite documentation was covered in [Chapter 4, Store Financial Market Data on Your Computer](#), but if you're interested in more details about the `Connection` and `Cursor` objects, you can check the following URLs:

- Documentation for the SQLite `connect` method: <https://docs.python.org/3/library/sqlite3.html#sqlite3.connect>
- Documentation for the SQLite `Cursor` object: <https://docs.python.org/3/library/sqlite3.html#cursor-objects>

Manage Orders, Positions, and Portfolios with the IB API

In algorithmic trading, efficient management of orders, positions, and portfolio data is critical. Luckily for us, we can do it all using Python. Managing orders encompasses a range of activities, including executing new trades, canceling existing orders, and updating orders to adapt to changing market conditions or shifts in trading strategies. Managing positions involves monitoring and analyzing live position data to track **profit and loss (PnL)** in real time. This immediate insight into the performance of individual trades enables traders to make informed decisions on whether to hold, sell, or adjust positions. Further, real-time (or near real-time) portfolio data can generate real-time (or near real-time) risk statistics to improve overall risk management. Portfolio data management involves a comprehensive analysis of the portfolio to assess its performance, understand risk exposure, and make strategic adjustments for optimizing returns. This is especially true when trading stocks on margin or futures where there is a financial cost and opportunity cost in holding losing positions.

As we've seen, the IB API uses a consistent request-callback pattern that can be used in several aspects of our trading app, including order management, position management, and accessing portfolio details. This pattern allows for the initiation of a request, such as placing or modifying an order, retrieving current position data, or gathering portfolio information, followed by a callback function that handles the response. We'll use this pattern throughout the chapter. By the end of this chapter, we'll be able to submit and modify orders to IB through Python, obtain key portfolio details including information on liquidity, margin requirements, and open positions, and compute our portfolio's profit or loss.

In this chapter, we present the following recipes:

- Executing orders with the IB API
- Managing orders once they're placed
- Getting details about your portfolio
- Inspecting positions and position details
- Computing portfolio profit and loss

Executing orders with the IB API

In [Chapter 10](#), Set Up the Interactive Brokers Python API, we created `contract` and `order` objects. Using these, we can use the IB API to execute trades. But before we can execute trades, we have to understand the concept of the next order ID.

The next order ID (`nextValidOrderId`) is a unique identifier for each order. Since up to 32 instances of a trading app can run in parallel, this identifier makes sure individual orders are traceable within the trading system. `nextValidOrderId` is used to preserve order integrity and prevent overlap between multiple orders submitted simultaneously or in rapid succession. When our trading app connects to the IB API, it receives an integer variable called `nextValidOrderId` from the server that is unique to each client connection to TWS. This ID must be used for the first order submission. Subsequently, we are responsible for incrementing this identifier for each new order.

Getting ready

We assume you've created the `client.py` and `wrapper.py` files in the `trading-app` directory. If not, do it now.

How to do it...

First, we'll add the code to deal with the integer `nextValidOrderId` in the `wrapper.py` file and print out the details of the order's execution:

1. Add an instance variable, `nextValidOrderId`, in the `__init__` method of our `IBWrapper` class:

```
self.nextValidOrderId = None
```

2. Add an implementation of the `nextValidId` method overriding the method from the inherited `EWrapper` class in our `IBWrapper` class:

```
def nextValidId(self, order_id):  
    super().nextValidId(order_id)  
    self.nextValidOrderId = order_id
```

3. Add an implementation of the `orderStatus` method overriding the method from the inherited `EWrapper` class:

```
def orderStatus(  
    self,  
    order_id,  
    status,  
    filled,  
    remaining,  
    avg_fill_price,  
    perm_id,  
    parent_id,  
    last_fill_price,  
    client_id,  
    why_held,
```

```

        mkt_cap_price,
    ) :
        print(
            "orderStatus - orderid:",
            order_id,
            "status:",
            status,
            "filled",
            filled,
            "remaining",
            remaining,
            "lastFillPrice",
            last_fill_price,
        )

```

4. Add an implementation of the **openOrder** method overriding the method from the inherited **EWrapper** class:

```

def openOrder(self, order_id, contract, order,
             order_state):
    print(
        "openOrder id:",
        order_id,
        contract.symbol,
        contract.secType,
        "@",
        contract.exchange,
        ":",
        order.action,
        order.orderType,
        order.totalQuantity,
        order_state.status,
    )

```

5. Add an implementation of the **execDetails** method overriding the method from the inherited **EWrapper** class:

```

def execDetails(self, request_id, contract, execution):
    print(
        "Order Executed: ",
        request_id,
        contract.symbol,
        contract.secType,
        contract.currency,
        execution.execId,
        execution.orderId,
        execution.shares,
        execution.lastLiquidity,
    )

```

6. Now, open the **client.py** file. Here, we'll add a custom method, **send_order**, under the **__init__** method that accepts a **contract** object and **order** object, increments the **nextValidOrderId** variable, and sends the order to the exchange:

```

def send_order(self, contract, order):
    order_id = self.wrapper.nextValidOrderId
    self.placeOrder(orderId=order_id,
                    contract=contract, order=order)
    self.reqIds(-1)
    return order_id

```

The result of the changes is the following code in the **client.py** file:

```

import time
import pandas as pd
from utils import Tick, TRADE_BAR_PROPERTIES

```

```

from ibapi.client import EClient
class IBClient(EClient):
    def __init__(self, wrapper):
        EClient.__init__(self, wrapper)
    def send_order(self, contract, order):
        order_id = self.wrapper.nextValidOrderId
        self.placeOrder(orderId=order_id,
                        contract=contract, order=order)
        self.reqIds(-1)
        return order_id
<snip>

```

How it works...

First, we'll cover the overridden methods in the `IBWrapper` class.

When orders are submitted through the `placeOrder` method from the `IBClient` class, `orderStatus`, `openOrder`, `execDetails` from the `IBWrapper` class are invoked depending on the life cycle of the order. They accomplish the following:

- `orderStatus` receives updates about the status of submitted orders
- `openOrder` provides information about orders that have been submitted but not fully executed
- `execDetails` provides detailed information about the execution of an order

Each of these callbacks accepts parameters that are passed from the IB API. In this recipe, we just print out the information. In more sophisticated applications, we can use the events to trigger risk analysis, portfolio updates, or alerts.

When `send_order` is called, it takes a `contract` object and an `order` object. The `contract` object represents the financial instrument to be ordered, while the `order` object contains the details of the order. The first line within the method, `order_id = self.wrapper.nextValidOrderId`, retrieves the next valid order ID from the `IBWrapper` object. The next line, `self.placeOrder(orderId=order_id, contract=contract, order=order)`, is a call to the `placeOrder` method of the IB API. This method is responsible for placing the order with the given `order_id`, `contract`, and `order` details on the IB server. The subsequent call, `reqIds(-1)`, is a request to the server to increment the internal counter for the next valid order ID.

There's more...

We now have everything in place to send orders to IB through the API using Python. To send an order for execution, import our `order` type and the `BUY` constant at the top of the `app.py` file:

```
from order import limit, BUY
```

Then, add the following code after the line that defines the trading app:

```
limit_order = limit(BUY, 100, 190.00)
app.send_order(aapl, limit_order)
```

This uses the `contract` object we set up in the *Creating a Contract object with the IB API* and the `order` object we set up in the *Creating an Order object with the IB API* recipes from [Chapter 9](#).

The result of the changes is the following code in the `app.py` file:

```
import threading
import time
import sqlite3
from wrapper import IBWrapper
from client import IBClient
from contract import stock, future, option
from order import limit, BUY
class IBApp(IBWrapper, IBClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.create_table()
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
                                   daemon=True)
        thread.start()
        time.sleep(2)
    <snip>
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=11)
    aapl = stock("AAPL", "SMART", "USD")
    gbl = future("GBL", "EUREX", "202403")
    pltr = option("PLTR", "BOX", "20240315", 20, "C")
    limit_order = limit(BUY, 100, 190.00)
    app.send_order(aapl, limit_order)
    time.sleep(30)
    app.disconnect()
```

After running this code, you will see a series of messages in the terminal. These messages are a result of the `orderStatus`, `openOrder`, and `execDetails` callbacks being called after the `placeOrder` method was invoked:

```
openOrder id: 9 AAPL STK @ SMART : BUY LMT 100 PreSubmitted
orderStatus - orderid: 9 status: PreSubmitted filled 0 remaining 100 lastFillPrice 0.0
Order Executed: -1 AAPL STK USD 0000e0d5.6554a46e.01.01 9 100 2
openOrder id: 9 AAPL STK @ SMART : BUY LMT 100 Filled
orderStatus - orderid: 9 status: Filled filled 100 remaining 0 lastFillPrice 188.66
openOrder id: 9 AAPL STK @ SMART : BUY LMT 100 Filled
orderStatus - orderid: 9 status: Filled filled 100 remaining 0 lastFillPrice 188.66
```

Figure 11.1: Message indicating the limit order was executed

IMPORTANT NOTE

In this example, we enter a buy limit order at a price above the current ask, which, at the time of writing, was \$188.66. This means that even though the order is a limit order, it will be executed immediately. This is done only for demonstration and testing purposes. In most scenarios, we'd enter a limit order under the best bid to wait for our desired price.

See also

To learn more about the famous `nextValidOrderId` and details about order execution, see the documentation here: https://interactivebrokers.github.io/tws-api/order_submission.html. This URL details the `openOrder` and `orderStatus` callbacks.

Managing orders once they're placed

Effective order management is important and typically involves canceling or updating existing orders. Canceling orders is straightforward: we may enter a limit or stop loss order that we no longer want. Market conditions change or our strategy indicates a different entry or exit position. In these cases, we'll use the IB API to completely cancel the order.

On the other hand, we may want the order to remain in the order book but with different attributes. Traders frequently update orders to change the quantity being traded, which allows them to scale into positions in response to market analysis or risk management requirements. Adjusting limit prices is another common update, which lets us set a new maximum purchase price or minimum sale price, depending on market conditions. Similarly, modifying stop prices is a strategic move to manage potential losses or lock in profits, especially in volatile markets.

We can update existing orders using the IB API by calling the `placeOrder` method with the same fields as the open order, except with the parameter to modify. This includes the order's ID, which must match the existing open order. IB recommends only changing order price, size, and **time in force**. Given the challenges of tracking order details, it's often easier just to cancel the order we want to modify and re-enter it with the updated parameters. That's the approach we'll take in this recipe.

Getting ready

We assume you've created the `client.py` and `app.py` files in the `trading-app` directory. If not, do it now.

How to do it...

We'll add three new methods to our `client.py` file to manage orders:

1. Add the `cancel_all_orders` method to our `IBClient` class directly under the `__init__` method:

```
def cancel_all_orders(self):
    self.reqGlobalCancel()
```

2. Add the `cancel_order_by_id` method next:

```
def cancel_order_by_id(self, order_id):
    self.cancelOrder(orderId=order_id,
                     manualCancelOrderTime="")
```

3. Finally, add `update_order`:

```
def update_order(self, contract, order, order_id):
    self.cancel_order_by_id(order_id)
    return self.send_order(contract, order)
```

4. The result of the changes is the following code in the `client.py` file:

```
<snip>
class IBClient(EClient):
    def __init__(self, wrapper):
        EClient.__init__(self, wrapper)
    def cancel_all_orders(self):
        self.reqGlobalCancel()
    def cancel_order_by_id(self, order_id):
        self.cancelOrder(orderId=order_id,
                         manualCancelOrderTime="")
    def update_order(self, contract, order, order_id):
        self.cancel_order_by_id(order_id)
        return self.send_order(contract, order)
<snip>
```

How it works...

We start by creating a function to cancel all open orders. The `cancel_all_orders` method executes the `reqGlobalCancel` method, which is a command to cancel all open orders placed through the current session, ensuring that no pending orders remain active in the trading system.

IMPORTANT NOTE

Calling `cancel_all_orders` will cancel all open orders, regardless of how they were originally placed. That means it will cancel orders manually entered through TWS in addition to those entered through the IB API.

To cancel a single order, we use `cancel_order_by_id`, which cancels a specific order identified by its integer `order_id`. When invoked, it calls the `cancelOrder` method, passing the integer `order_id` as an argument, along with an empty string for `manualCancelOrderTime`. The `send_order` method returns the `order_id` that is used to cancel the order.

To update orders, we combine two methods. The `update_order` method first cancels an existing order identified by `order_id` using the `cancel_order_by_id` method. It then creates and sends a new order with the specified `contract` and `order` details using the `send_order` method, effectively updating the

original order by replacing it with a new one. This is the recommended method of updating orders using the IB API.

There's more...

Let's test out our new method. In the `app.py` file, add the following code after the line that defines our AAPL (jas rev. 2024-07-03):

```
order_1 = limit(BUY, 10, 185.0)
order_1_id = app.send_order(aapl, order_1)
```

Once you run this code, you'll see an order in the **Orders** section of TWS:

Orders						ALL	?	⚙️	🔗	▼
	Actn	Type	Details	Quantity	Fill Px					
■ AAPL	BUY	LMT	LMT 185.00 ▾	0/10	-					
						Cancel				

Figure 11.2: Our AAPL limit order safely resting off the market

To cancel the order, run the following code:

```
app.cancel_order_by_id(order_1_id)
```

You'll now see our order canceled:

Orders						ALL	?	⚙️	🔗	▼
	Actn	Type	Details	Quantity	Fill Px					
● AAPL	BUY	LMT	LMT 185.00 ▾	0/10	-					
						Cancel				

Figure 11.3: Our AAPL limit order is canceled

Let's reenter the order, create a second order with a different limit price, and update it:

```
order_2 = limit(BUY, 10, 187.50)
app.update_order(aapl, order_2, order_1_id)
```

Our original order is canceled, and the new one is pending:

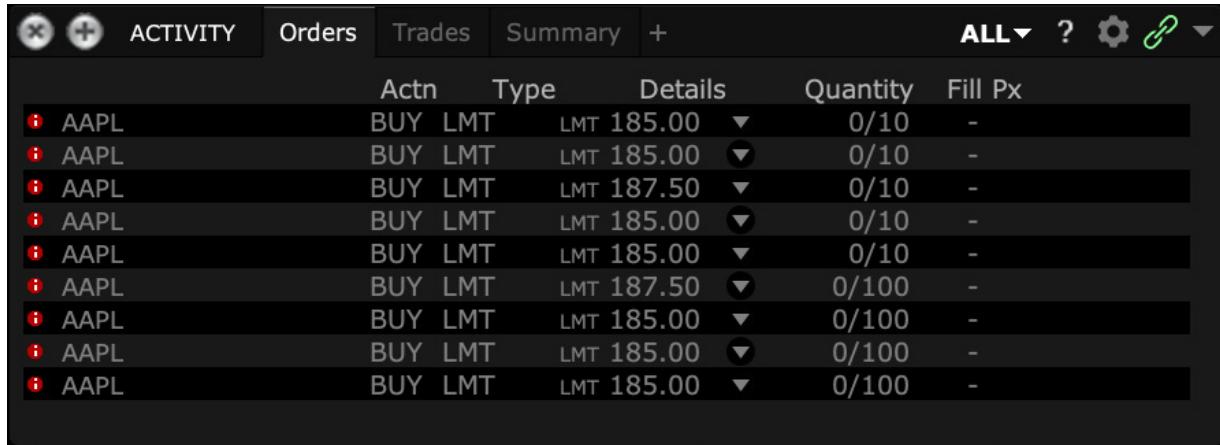
Orders						ALL	?	⚙️	🔗	▼
	Actn	Type	Details	Quantity	Fill Px					
■ AAPL	BUY	LMT	LMT 187.50 ▾	0/10	-	Cancel				
● AAPL	BUY	LMT	LMT 185.00 ▾	0/10	-					

Figure 11.4: Our original AAPL order is canceled and our new AAPL order is entered

Finally, cancel all open orders:

```
app.cancel_all_orders()
```

As a result, all open orders are canceled:



The screenshot shows the Interactive Brokers TWS application interface. The top navigation bar includes tabs for ACTIVITY, Orders (which is selected), Trades, Summary, and a plus sign. On the right side of the header are buttons for ALL, ?, Settings, and a link icon. The main content area displays a table of orders. The columns are labeled: Actn, Type, Details, Quantity, and Fill Px. There are ten rows, each representing a canceled order for AAPL. The 'Actn' column shows a red circular icon with a question mark. The 'Type' column shows 'BUY LMT'. The 'Details' column shows 'LMT 185.00'. The 'Quantity' column shows '0/10' or '0/100'. The 'Fill Px' column shows '-'.

Actn	Type	Details	Quantity	Fill Px
?	BUY LMT	LMT 185.00	0/10	-
?	BUY LMT	LMT 185.00	0/10	-
?	BUY LMT	LMT 187.50	0/10	-
?	BUY LMT	LMT 185.00	0/10	-
?	BUY LMT	LMT 185.00	0/10	-
?	BUY LMT	LMT 187.50	0/100	-
?	BUY LMT	LMT 185.00	0/100	-
?	BUY LMT	LMT 185.00	0/100	-
?	BUY LMT	LMT 185.00	0/100	-

Figure 11.5: All open orders are canceled

See also

Read more about modifying orders here: https://interactivebrokers.github.io/tws-api/modifying_orders.html.

Getting details about your portfolio

The IB API offers a comprehensive snapshot of portfolio data, returning 157 different portfolio values through a single API call. This data provides a detailed view of our portfolios, encompassing a wide range of metrics and data points. Account values delivered via `updateAccountValue` can be classified in the following way:

- **Commodities:** Suffix by `-C`
- **Securities:** Suffix by `-S`
- **Totals:** No suffix

In this recipe, we'll build the code to get those data points.

Getting ready

We assume you've created the `client.py`, `wrapper.py`, and `app.py` files in the `trading-app` directory. If not, do it now.

How to do it...

The first step is to incorporate the account number into our `IBApp` class. While an account number is optional for requesting account-level data in a single account structure, it's best practice to specify it in the case of multiple accounts. Then, we'll add the callback to the `IBWrapper` class and add the request method to the `IBClient` class:

1. Modify the `__init__` method of our `IBApp` class by adding `account` as an argument:

```
def __init__(self, ip, port, client_id, account):
```

2. Add the following dictionary to the `__ini__` method in the `IBWrapper` class to store our account details:

```
self.account_values = {}
```

3. Then, add the callback function that responds to the TWS messages:

```
def updateAccountValue(self, key, val, currency,
                      account):
    try:
        val_ = float(val)
    except:
        val_ = val
    self.account_values[key] = (val_, currency)
```

The result of the changes is the following code in the `wrapper.py` file:

```
import threading
from ibapi.wrapper import EWrapper
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
        self.nextValidOrderId = None
        self.historical_data = {}
        self.market_data = {}
        self.streaming_data = {}
        self.stream_event = threading.Event()
        self.account_values = {}
    <snip>
    def updateAccountValue(self, key, val, currency,
                          account):
        try:
            val_ = float(val)
        except:
            val_ = val
        self.account_values[key] = (
            val_, currency)
```

4. At the end of the `IBClient` class, add the following method:

```
def get_account_values(self, key=None):
    self.reqAccountUpdates(True, self.account)
    time.sleep(2)
    if key:
        return self.account_values[key]
    return self.account_values
```

How it works...

By now, we're settling into a common pattern: request something to happen in the `IBClient` class and add the results of the call to a dictionary in the `IBWrapper` class. The `updateAccountValue` method is a callback that accepts a key (representing a specific account attribute), a value, a currency, and an account identifier, then stores the value and currency as a tuple in the `account_values` dictionary, keyed by the provided attribute key.

The `get_account_values` method requests account updates, `reqAccountUpdates`, then pauses execution for two seconds to allow time for the account data to be updated. If a specific key is provided, the method returns the value associated with that key from the `account_values` dictionary. If no key is specified, it returns the entire `account_values` dictionary, providing a snapshot of all account-related values.

The IB API returns 157 different account values. Some of the more common are as follows:

- `AvailableFunds`
- `BuyingPower`
- `CashBalance`
- `Currency`
- `EquityWithLoanValue`
- `FullAvailableFunds`
- `NetLiquidation`
- `RealizedPnL`
- `TotalCashBalance`
- `UnrealizedPnL`

There's more...

Let's get the net liquidation value of the account. In the `app.py` file, replace the code after the `IBApp` class with the following:

```
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10,
                account="DU7129120")
    account_values = app.get_account_values()
    net_liquidation = app.get_account_values(
        "NetLiquidation")
    app.disconnect()
```

The result is the `account_values` dictionary, which contains all 157 account values. Here's a sample:

```
{
    'AccountReady': ('true', ''),
    'AccountType': ('INDIVIDUAL', ''),
    'AccruedCash': (0.0, 'BASE'),
    'AccruedCash-C': (0.0, 'USD'),
    'AccruedCash-P': (0.0, 'USD'),
    'AccruedCash-S': (0.0, 'USD'),
    'AccruedDividend': (0.0, 'USD'),
    'AccruedDividend-C': (0.0, 'USD'),
    'AccruedDividend-P': (0.0, 'USD'),
    'AccruedDividend-S': (0.0, 'USD'),
    'AvailableFunds': (3195.34, 'USD'),
    'AvailableFunds-C': (0.0, 'USD'),
    'AvailableFunds-P': (0.0, 'USD'),
    'AvailableFunds-S': (3195.34, 'USD'),
    ...
    'SMA': (2312528.09, 'USD'),
    'SMA-S': (2312528.09, 'USD'),
    'SegmentTitle-C': ('US Commodities', ''),
    'SegmentTitle-P': ('Crypto at Paxos', ''),
    'SegmentTitle-S': ('US Securities', ''),
    'StockMarketValue': (-769.01, 'BASE'),
    'TBillValue': (0.0, 'BASE'),
    'TBondValue': (0.0, 'BASE'),
    'TotalCashBalance': (5175.9625, 'BASE'),
    'TotalCashValue': (5175.96, 'USD'),
    'TotalCashValue-C': (0.0, 'USD'),
    'TotalCashValue-P': (0.0, 'USD'),
    'TotalCashValue-S': (5175.96, 'USD'),
    'TotalDebitCardPendingCharges': (0.0, 'USD'),
    'TotalDebitCardPendingCharges-C': (0.0, 'USD'),
    'TotalDebitCardPendingCharges-P': (0.0, 'USD'),
    'TotalDebitCardPendingCharges-S': (0.0, 'USD'),
    'TradingType-S': ('STKNOPT', ''),
    'UnrealizedPnL': (-12.9, 'BASE'),
    'WarrantValue': (0.0, 'BASE'),
    'WhatIfPMEnabled': ('true', '')
}
```

Figure 11.6: A sample of the data provided by `get_account_values`

The result is the `net_liquidation` tuple, which contains the account's net liquidation value and the currency the value is denominated in. Here's what it looks like:

<code>net_liquidation</code>
(4408.76, 'USD')

Figure 11.7: Net liquidation value of the account

See also

For a complete list of account values and their descriptions, see the following URL:

https://interactivebrokers.github.io/tws-api/interfaceIBApi_1_IERWrapper.html#ae15a34084d9f26f279abd0bdeab1b9b5.

Inspecting positions and position details

To get position-level details from the IB API, including data such as position size, market price, value, average cost, and PnL, we can utilize specific API calls. These calls request detailed information for each position held in an account, with the API responding with the requested data for each position. This enables us to have a comprehensive view of our holdings. In this recipe, we introduce how to get the position data. In the next chapter, we'll make use of it to build a trading strategy.

Getting ready

We assume you've created the `client.py`, `wrapper.py`, and `app.py` files in the `trading-app` directory. If not, do it now.

How to do it...

In the previous recipe, *Getting details about your portfolio*, we used the `reqAccountUpdates` request method in the `IBClient` class to request account details. Calling `reqAccountUpdates` triggers two callbacks. The first is `updateAccountValue`, which we overrode in the `IBWrapper` method. This method returns details about the account. `reqAccountUpdates` also triggers the `updatePortfolio` callback, which returns details about the positions in the account. We'll use the `updatePortfolio` method to get account details using the same `reqAccountUpdates` method:

1. Add a dictionary to the end of the `__init__` method in the `IBWrapper` class to store position details:

```
self.positions = {}
```

2. Add the following method to the end of the `IBWrapper` class:

```
def updatePortfolio(  
    self,  
    contract,  
    position,
```

```

        market_price,
        market_value,
        average_cost,
        unrealized_pnl,
        realized_pnl,
        account_name
    ) :
    portfolio_data = {
        "contract": contract,
        "symbol": contract.symbol,
        "position": position,
        "market_price": market_price,
        "market_value": market_value,
        "average_cost": average_cost,
        "unrealized_pnl": unrealized_pnl,
        "realized_pnl": realized_pnl,
    }
    self.positions[contract.symbol] = portfolio_data

```

3. The result of the changes is the following code in the **wrapper.py** file:

```

<snip>
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
        self.nextValidOrderId = None
        self.historical_data = {}
        self.streaming_data = {}
        self.stream_event = threading.Event()
        self.account_values = {}
        self.positions = {}

    <snip>
    def updatePortfolio(
            self,
            contract,
            position,
            market_price,
            market_value,
            average_cost,
            unrealized_pnl,
            realized_pnl,
            account_name
        ) :
        portfolio_data = {
            "contract": contract,
            "symbol": contract.symbol,
            "position": position,
            "market_price": market_price,
            "market_value": market_value,
            "average_cost": average_cost,
            "unrealized_pnl": unrealized_pnl,
            "realized_pnl": realized_pnl,
        }
        self.positions[contract.symbol] = portfolio_data

```

4. Now, add the following method to the end of the **IBClient** class:

```

def get_positions(self):
    self.reqAccountUpdates(True, self.account)
    time.sleep(2)
    return self.positions

```

How it works...

When `reqAccountUpdates` is called, the `updatePortfolio` callback is triggered for every position in the account. TWS passes details about the position to the method, which are captured in the `positions` dictionary, and keyed by the contract's symbol.

The `get_positions` method triggers the callback through the `reqAccountsUpdates` method, waits two seconds, and returns the dictionary containing the positions.

There's more...

To get the current positions in the account, add the following code to the end of the `app.py` file:

```
positions = app.get_positions()
```

The result is a dictionary with position details:

```
{"AAPL": {"contract": 4966399376: 265598,AAPL,STK,,0,0,,,NASDAQ,USD,AAPL,NMS,False,,,,combo:,  
'symbol': 'AAPL',  
'position': Decimal('10'),  
'market_price': 191.30000305,  
'market_value': 1913.0,  
'average_cost': 191.66,  
'unrealized_pnl': -3.6,  
'realized_pnl': 0.0},  
'NVDA': {"contract": 4966396112: 4815747,NVDA,STK,,0,0,,,NASDAQ,USD,NVDA,NMS,False,,,,combo:,  
'symbol': 'NVDA',  
'position': Decimal('-10'),  
'market_price': 487.3200073,  
'market_value': -4873.2,  
'average_cost': 487.27595595,  
'unrealized_pnl': -0.44,  
'realized_pnl': 0.0}}
```

Figure 11.8: Position details of the account

See also

Read more about getting position data using the `reqAccountsUpdates` method at this URL:

https://interactivebrokers.github.io/tws-api/account_updates.html.

Computing portfolio profit and loss

To get portfolio PnL details from the IB API, we can utilize specific API calls. These calls request the aggregate daily profit or loss, total unrealized profit or loss, and total realized PnL, with the API responding with the requested data. Getting PnL allows us to compute periodic portfolio returns, which, in turn, unlock a suite of risk metrics based on portfolio returns. We'll make use of the dollar profit or loss to compute periodic portfolio returns in the next chapter. In this recipe, we focus on requesting and receiving the portfolio profit or loss.

Getting ready

We assume you've created the `client.py`, `wrapper.py`, and `app.py` files in the `trading-app` directory. If not, do it now.

How to do it...

We'll follow the same pattern in this recipe as in the last few:

1. Add a dictionary to the end of the `__init__` method in the `IBWrapper` class to store position details:

```
self.account_pnl = {}
```

2. Add the following method to the end of the `IBWrapper` class:

```
def pnl(self, request_id, daily_pnl, unrealized_pnl,
        realized_pnl):
    pnl_data = {
        "daily_pnl": daily_pnl,
        "unrealized_pnl": unrealized_pnl,
        "realized_pnl": realized_pnl
    }
    self.account_pnl[request_id] = pnl_data
```

3. The result of the changes is the following code in the `wrapper.py` file:

```
<snip>
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
        self.nextValidOrderId = None
        self.historical_data = {}
        self.streaming_data = {}
        self.stream_event = threading.Event()
        self.account_values = {}
        self.positions = {}
        self.account_pnl = {}

    def pnl(self, request_id, daily_pnl,
            unrealized_pnl, realized_pnl):
        pnl_data = {
            "daily_pnl": daily_pnl,
            "unrealized_pnl": unrealized_pnl,
            "realized_pnl": realized_pnl
        }
```

```
        self.account_pnl[request_id] = pnl_data
<snip>
```

4. Now, add the following method to the end of the `IBClient` class:

```
def get_pnl(self, request_id):
    self.reqPnL(request_id, self.account, "")
    time.sleep(2)
    return self.account_pnl
```

How it works...

The `pnl` method is a callback function to handle PnL data. When called, it receives an integer `request_id` along with three types of PnL data: `daily_pnl`, `unrealized_pnl`, and `realized_pnl`. The method then constructs a `pnl_data` dictionary with these values and stores it in the `account_pnl` dictionary, keyed by `request_id`, effectively updating the account's PnL information associated with that specific request.

The `get_pnl` method requests PnL data for a specific account. It calls `reqPnL` to send a request to the IB API for PnL information associated with the given `request_id` and account, then pauses for two seconds to allow time for the data to be received and processed, before returning the updated PnL data stored in `account_pnl`. Unrealized profit or loss is the potential financial gain or loss on an investment that has not yet been sold for cash. Realized profit or loss is the gain or loss on the asset once it's been sold.

There's more...

At the bottom of the `app.py` file, add the following code to get the account's PnL:

```
pnl = app.get_pnl(request_id=99)
```

The result is a dictionary with the daily, unrealized, and realized PnL:

```
{99: {'daily_pnl': 5.859764794922285,
      'unrealized_pnl': 5.859724294922216,
      'realized_pnL': 0.0}}
```

Figure 11.9: The result of calling `get_pnl` is a dictionary with the account PnL

See also

The IB API can retrieve PnL data from two different sources: the TWS Account window and the TWS Portfolio window. These sources have different update times, which may result in differing

portfolio PnL values. The method described in this recipe requests PnL data from the TWS Portfolio window with a reset schedule specified in the global configuration.

For more details on how portfolio PnL is calculated, see the following URL:

<https://interactivebrokers.github.io/tws-api/pnl.html>.

Deploy Strategies to a Live Environment

In [Chapter 10](#), *Set up the Interactive Brokers Python API*, and [Chapter 11](#), *Manage Orders, Positions, and Portfolios with the IB API*, we set the stage to begin deploying algorithmic trading strategies into a live (or paper trading) environment. Before we get there, we need two more critical pieces of the algorithmic trading puzzle: risk and performance metrics and more sophisticated order strategies that allow us to build and rebalance asset portfolios. For risk and performance metrics, we will introduce the **Empirical Reloaded** library, which generates statistics based on portfolio returns. `empirical-reloaded` is the library that provides the performance and risk analytics behind **Pyfolio Reloaded**, which we learned about in [Chapter 9](#), *Assess Backtest Risk and Performance Metrics with Pyfolio*. In this chapter, we'll use `empirical-reloaded` to calculate key performance indicators such as the Sharpe ratio, Sortino ratio, and the maximum drawdown, among others, using real-time portfolio return data. To allow us to compute real-time return data while executing trades or other code in our trading app, we'll learn how to run code asynchronously on a thread.

In addition to calculating risk and performance metrics, we will finalize our position management code by introducing methods to submit orders based on a target number of contracts, monetary value, or percentage allocation. This extended functionality unlocks portfolio-based strategies as opposed to only single-asset strategies. Finally, we will introduce three algorithmic trading strategies that you can deploy right away: a monthly factor-based strategy using the `zipline-reloaded` pipeline API that we learned about in [Chapter 7](#), *Event-Based Backtesting FactorPortfolios with Zipline Reloaded*, an options combo strategy, and an intraday multi-asset mean-reversion strategy. These strategies take advantage of the code we've built throughout this book.

This chapter contains the following recipes:

- Calculating real-time key performance and risk indicators
- Sending orders based on portfolio targets
- Deploying a monthly factor portfolio strategy
- Deploying an options combo strategy
- Deploying an intraday multi-asset mean reversion strategy

Calculating real-time key performance and risk indicators

Real-time performance and risk metrics are important for maintaining robust trading strategies. They allow us to compare real-life performance to the performance of our backtests. They provide immediate feedback on the effectiveness of our trading algorithms and let us make adjustments in response to market volatility or unexpected events. By continuously monitoring risk metrics such as drawdowns, volatility, and value at risk, we can effectively manage exposure and mitigate potential losses. Most professional algorithmic traders spend their time analyzing and explaining deviations from the performance that they expect in their backtests to the performance that they observe during live trading. This recipe will introduce the tools we need to do the same.

Getting ready

We'll use `empirical-reloaded` to compute performance and risk statistics. To install it, use `pip`:

```
pip install empirical-reloaded
```

At the top of `app.py`, import `Empirical Reloaded`:

```
import empirical as ep
```

To calculate real-time performance and risk indicators, we need periodic portfolio returns.

Unfortunately, the IB API does not provide a way to retrieve portfolio returns, so we'll need to build our own method. We'll do this by periodically requesting account PnL and then using it to compute returns. To allow the rest of our trading app to run independently of the PnL calculations, we'll run the method on its own thread.

Add a new instance variable to `wrapper.py`, which we'll use to capture account PnL. At the end of the `__init__` method in the `IBWrapper` class, add the following:

```
self.portfolio_returns = None
```

Add a method to `client.py` that continuously requests account PnL from the `get_pnl` method we built in [Chapter 9](#):

```
def get_streaming_pnl(self, request_id, interval=60,
                      pnl_type="unrealized_pnl"):
    interval = max(interval, 5) - 2
    while True:
        pnl = self.get_pnl(request_id=request_id)
        yield {"date": pd.Timestamp.now(),
                "pnl": pnl[request_id].get(pnl_type)}
        time.sleep(interval)
```

The method first ensures that the interval between data retrievals is at least three seconds. This accounts for the three-second update time from IB and the two-second wait time in `get_pnl`. In an

infinite loop, it retrieves the PnL data and then yields a dictionary containing the current timestamp and the specified type of PnL before pausing for the defined interval duration.

Add a method to stream portfolio PnL and compute the associated returns:

```
def get_streaming_returns(self, request_id, interval,
                           pnl_type):
    returns = pd.Series(dtype=float)
    for snapshot in self.get_streaming_pnl(
        request_id=request_id,
        interval=interval,
        pnl_type=pnl_type
    ):
        returns.loc[snapshot["date"]] = snapshot["pnl"]
        if len(returns) > 1:
            self.portfolio_returns = (
                returns
                .pct_change()
                .dropna()
            )
```

The `get_streaming_returns` method calculates the periodic PnL. It initializes a pandas Series to store PnL values, then iterates over PnL data snapshots obtained from `get_streaming_pnl`, adding each PnL value to the `returns` Series indexed by the snapshot's timestamp. If the Series has more than one entry, it calculates the percentage change in PnL, updating the `portfolio_returns` attribute with these calculated returns after dropping the `NaN` values.

In `wrapper.py`, add the following instance variable at the end of the `__init__` method:

```
self.portfolio_returns = None
```

In `app.py`, update the `__init__` method so it resembles the following:

```
def __init__(self, ip, port, client_id, account,
             interval=5):
    IBWrapper.__init__(self)
    IBCClient.__init__(self, wrapper=self)
    self.account = account
    self.create_table()
    self.connect(ip, port, client_id)
    threading.Thread(
        target=self.run, daemon=True).start()
    time.sleep(2)
    threading.Thread(
        target=self.get_streaming_returns,
        args=(99, interval, "unrealized_pnl"),
        daemon=True
    ).start()
```

Upon instantiation of our `IBApp` class, we start two separate threads: one to continuously run the main event loop of the API client and another to stream unrealized PnL returns every five seconds, both beginning their execution after a two-second pause to ensure that the connection is established. This allows us to continue to access the periodic portfolio returns while executing other code.

How to do it...

We'll implement six methods (decorated as properties) to compute performance and risk metrics in our `IBApp` class using `empirical-reloaded`. You should add these methods after the `stream_to_sql` method we added in [Chapter 11, Manage Orders, Positions, and Portfolios with the IB API](#).

1. Add a method to compute cumulative returns:

```
@property
def cumulative_returns(self):
    return ep.cum_returns(self.portfolio_returns, 1)
```

2. Add a method to compute the maximum drawdown:

```
@property
def max_drawdown(self):
    return ep.max_drawdown(self.portfolio_returns)
```

3. Add a method to compute the volatility of returns:

```
@property
def volatility(self):
    return self.portfolio_returns.std(ddof=1)
```

4. Add a method to compute the Omega ratio:

```
@property
def omega_ratio(self):
    return ep.omega_ratio(self.portfolio_returns,
                           annualization=1)
```

5. Add a method to compute the Sharpe ratio:

```
@property
def sharpe_ratio(self):
    return self.portfolio_returns.mean() / self.portfolio_returns.std(ddof=1)
```

6. Add a method to compute the percentage and dollar conditional value at risk:

```
@property
def cvar(self):
    net_liquidation = self.get_account_values(
        "NetLiquidation")[0]
    cvar_ = ep.conditional_value_at_risk(
        self.portfolio_returns)
    return (
        cvar_,
        cvar_ * net_liquidation
    )
```

How it works...

We'll cover the details of each method in detail in the following sub-sections.

Cumulative returns

To get the cumulative returns of our portfolio, we add the following in the `app.py` file after defining our trading app:

```
app.cumulative_returns
```

The `cumulative_returns` method computes the cumulative returns of our portfolio returns. The first argument, `returns`, is a pandas Series representing the periodic returns of our portfolio. The second argument, `1`, specifies the starting value for the cumulative returns calculation. This function effectively calculates the compounded return at each point in time, assuming that the initial investment value is `1`. We can use this method to generate the following intraday equity curve by plotting the data from the resulting Series.

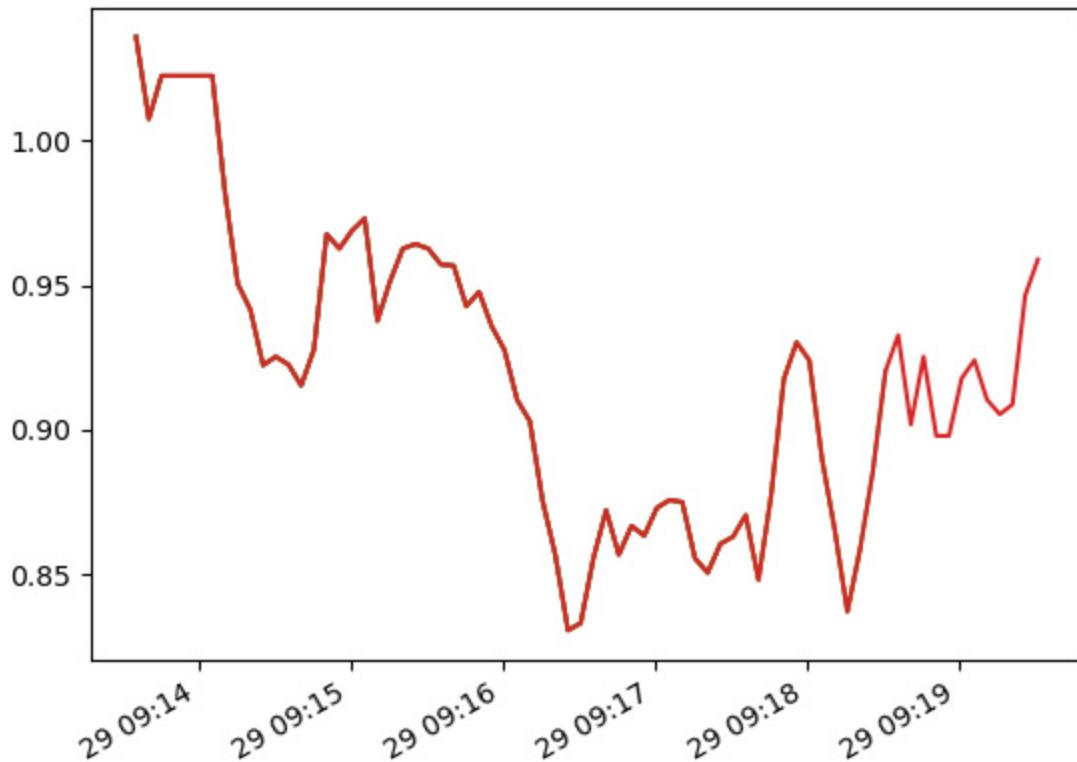


Figure 12.1: An intraday equity curve computed every five seconds based on cumulative returns

Max drawdown

To get the max drawdown of our portfolio, we add the following in the `app.py` file after defining our trading app:

```
app.max_drawdown
```

The `max_drawdown` method computes the maximum drawdown of our portfolio. The maximum drawdown is calculated by determining the largest decrease in value from a portfolio's peak to its lowest point before it reaches a new peak. It's typically expressed as a negative percentage. This metric is crucial in risk management and performance evaluation in algorithmic trading, as it provides insight into the potential downside risk of an investment strategy.

Volatility

To get the volatility of our portfolio, we add the following in the `app.py` file after defining our trading app:

```
app.volatility
```

The `volatility` method calculates the sample standard deviation of portfolio returns. This measure provides insight into the volatility or risk associated with the portfolio over the time period covered by the data. By setting `ddof` to `1`, the calculation adjusts for bias in the standard deviation estimate, making it more accurate for a sample (such as a subset of returns over a specific period) rather than the entire population of returns.

IMPORTANT NOTE

We don't use the `empirical-reloaded` method in this example because it assumes daily returns. When we compute volatility using `empirical-reloaded`, it annualizes the volatility metric, which is not appropriate for intraday values.

Omega ratio

To get the Omega ratio of our portfolio, we add the following in the `app.py` file after defining our trading app:

```
app.omega_ratio
```

The `omega_ratio` method calculates the Omega ratio for our portfolio returns. The Omega ratio is a risk-adjusted return measure of an investment asset, portfolio, or strategy, comparing the probability of achieving a threshold return level to the probability of falling below it. It's calculated by dividing the sum of excess returns above a threshold (`0` in our case) by the absolute value of the sum of returns below that threshold, where excess returns are annualized if an `annualization` factor is provided (which it is not in our case). In trading, this ratio is particularly useful as it provides a more comprehensive view of the risk-reward tradeoff than traditional measures such as the Sharpe ratio, especially in portfolios with non-normal return distributions.

Sharpe ratio

To get the Sharpe ratio of our portfolio, we add the following in the `app.py` file after defining our trading app:

```
app.sharpe_ratio
```

The `sharpe_ratio` method computes the Sharpe ratio for our portfolio returns. Typically, the `sharpe_ratio` method is computed by first calculating the difference between the returns of the portfolio and the risk-free rate. This excess return is then annualized by multiplying it by the square root of the `annualization` parameter. The annualized excess return is then divided by the standard deviation of the portfolio's returns, which is a measure of the portfolio's risk. Since we are using intraday returns, we won't use `empirical-reloaded` because it assumes daily returns.

Conditional value at risk

To get the conditional value at risk of our portfolio, we add the following in the `app.py` file after defining our trading app:

```
app.cvar
```

The **Conditional Value at Risk (CVaR)**, also known as the **Expected Shortfall (ES)**, is a risk assessment metric that estimates the expected loss in our portfolio under extreme market conditions, focusing on the tail end of the distribution of potential losses. The method first retrieves the `NetLiquidation` value of the account, which represents the total value of the portfolio if all positions were liquidated at current market prices. It then calculates the CVaR of the account's returns using `empirical-reloaded`'s `conditional_value_at_risk` function, which computes CVaR as the average of the worst-performing returns (those in the tail of the distribution) below a 5% percentile. The method returns two values: the CVaR as a percentage and the CVaR in terms of the actual monetary value, calculated by multiplying the CVaR percentage by the portfolio's net liquidation value.

There's more...

There are more than 40 risk and performance metrics available in `Empyrcal Reloaded`, including rolling metrics and those that compare portfolio returns to a benchmark. Here are a few more you can explore:

- `calmar_ratio`: Ratio of annualized return over maximum drawdown, assessing returns relative to downside risks
- `downside_risk`: Calculates the risk of negative returns, aiding in understanding a strategy's downside volatility
- `sortino_ratio`: Similar to Sharpe, but this focuses only on downside risk and is useful for strategies with asymmetric risk
- `stability_of_timeseries`: Measures consistency of returns; important for assessing strategy reliability over time
- `tail_ratio`: Compares the right (positive) versus left (negative) tail of a distribution, indicating extreme outcome frequency

- **value_at_risk**: Estimates the maximum potential loss over a specified time frame and is crucial for risk assessment
- **excess_sharpe**: Calculates the difference in Sharpe ratio relative to a benchmark; useful for strategy comparison

See also

To learn more about `empirical-reloaded`, check out the documentation here:

<https://empirical.ml4trading.io/>

Sending orders based on portfolio targets

We now have the components of our trading app built to add flexibility to the way we submit orders. By combining real-time position data and portfolio net liquidation value, we can build more sophisticated order techniques. For example, now that we can access current positions, we're able to dynamically adjust our positions to align with quantity or value targets. Similarly, with live net liquidation value, we can calculate order sizes as a percentage of the portfolio. Building orders based on portfolio percentage targets unlocks advanced portfolio and risk management capabilities. This integration results in a more responsive trading system that is capable of adapting to market changes swiftly and executing orders that are consistently in tune with our overall risk management and investment objectives. In this recipe, we'll implement methods to submit orders based on target values, quantities, and percentage allocations.

Getting ready

We will assume that you've created the `client.py` and `app.py` files in the `trading-app` directory. If you have not, do it now.

We'll start by adding a default market data request ID to the `utils.py` file. Since we cancel market data requests immediately after requesting the data, we can use the same ID.

Open `utils.py` and add the following code after the `TABLE_BAR_PROPERTIES` list:

```
DEFAULT_MARKET_DATA_ID = 55
```

How to do it...

We'll create five separate order methods and three associated helper methods. The order methods will generate the number of contracts to order so that we can achieve the desired portfolio allocation. The

helper methods will do the work of calculating the quantities. Add all of the following code under the `send_order` method in the `rbclient` class in the `client.py` file.

1. Add the method that computes the quantity and places an order for a fixed amount of money:

```
def order_value(self, contract, order_type, value, **kwargs):
    quantity = self._calculate_order_value_quantity(
        contract, value)
    order = order_type(quantity=quantity, **kwargs)
    return self.send_order(contract, order)
```

2. Add the method to place an order to adjust a position to the target number of contracts:

```
def order_target_quantity(self, contract, order_type,
                         target, **kwargs):
    quantity = self._calculate_order_target_quantity(contract,
                                                    target)
    order = order_type(
        action=SELL if quantity < 0 else BUY,
        quantity=abs(quantity),
        **kwargs
    )
    return self.send_order(contract, order)
```

3. Add the helper method that loops through the positions and computes the target number of contracts to order:

```
def _calculate_order_target_quantity(self, contract,
                                     target):
    positions = self.get_positions()
    if contract.symbol in positions.keys():
        current_position = positions[
            contract.symbol]["position"]
        target -= current_position
    return int(target)
```

4. Add the method to order the specified asset according to the percent of the current portfolio value:

```
def order_percent(self, contract, order_type, percent,
                  **kwargs):
    quantity = self._calculate_order_percent_quantity(
        contract, percent)
    order = order_type(quantity=quantity,
                       **kwargs)
    return self.send_order(contract, order)
```

5. Add the associated helper function that computes the percentage of the portfolio based on the net liquidation value and the number of contracts to order:

```
def _calculate_order_percent_quantity(self, contract, percent):
    net_liquidation_value = self.get_account_values(
        key="NetLiquidation")[0]
    value = net_liquidation_value * percent
    return self._calculate_order_value_quantity(contract, value)
```

6. Add the method to adjust a position to a target dollar value. If the position doesn't already exist, the code sends a new order. If the position exists, the code sends an order for the difference between the target value and the current position value:

```
def order_target_value(self, contract, order_type,
                      target, **kwargs):
    target_quantity = self._calculate_order_value_quantity(
        contract, target)
```

```

        quantity = self._calculate_order_target_quantity(contract,
            target_quantity)
    order = order_type(
        action=SELL if quantity < 0 else BUY,
        quantity=abs(quantity),
        **kwargs
    )
    return self.send_order(contract, order)

```

7. Add the associated helper method:

```

def _calculate_order_value_quantity(self, contract,
    value):
    last_price = self.get_market_data(
        request_id=DEFAULT_MARKET_DATA_ID,
        contract=contract, tick_type=4)
    multiplier = contract.multiplier if contract.multiplier
    != "" else 1
    return int(value / (last_price * multiplier))

```

8. Add the method to send an order to update a position to a percent of the portfolio value. If the position doesn't already exist, the code sends a new order. If the position exists, the code sends an order for the difference between the target percent and the current portfolio percent:

```

def order_target_percent(self, contract, order_type,
    target, **kwargs):
    quantity = self._calculate_order_target_percent_quantity(
        contract, target)
    order = order_type(
        action=SELL if quantity < 0 else BUY,
        quantity=abs(quantity),
        **kwargs
    )
    return self.send_order(contract, order)

```

9. Add the associated helper method:

```

def _calculate_order_target_percent_quantity(self,
    contract, target):
    target_quantity = self._calculate_order_percent_quantity(
        contract, target)
    return self._calculate_order_target_quantity(
        contract, target_quantity)

```

How it works...

Our order methods take two required arguments and various keyword arguments based on the order type that's passed. The two required arguments are the contract to trade and the order type to use. The keyword arguments are additional arguments to be passed to the order.

Ordering for a fixed amount of money with `order_value`

The `order_value` method places an order for a specific monetary value rather than a specified quantity of contracts. It calculates the quantity of the asset to be ordered by calling the

`_calculate_order_value_quantity` method, which determines the quantity based on the current market price of the asset and its multiplier. This calculation divides the specified dollar value by the product of the asset's last market price and its multiplier (defaulting to 1 if none are specified).

The method then creates an order of the given `order_type` object with the calculated quantity, along with any additional parameters passed via `**kwargs`. Finally, it places the order by invoking `send_order` with the contract and the order.

Ordering to adjust a position to the target number of shares with `order_target_quantity`

The `order_target_quantity` method adjusts a position to a specified target quantity of contracts. It calculates the quantity by determining the difference between the target quantity and the current position size using the `_calculate_order_target_quantity` method. If the contract already has an existing position, this method adjusts the position to the target by calculating the difference. If no position exists, it treats the target as the total quantity for a new order.

The method then creates an order of the specified `order_type` object, setting the action to `BUY` or `SELL` based on whether the calculated quantity is positive or negative. It uses the absolute value of this quantity. Finally, it places the order by calling `send_order` with the contract and the order.

Ordering to a given percent of current portfolio value with `order_percent`

The `order_percent` method uses the `_calculate_order_percent_quantity` helper method, which multiplies the portfolio's net liquidation value by the specified percentage, then converts this value into the appropriate quantity of the asset using `_calculate_order_value_quantity`.

The method then creates an order of the specified `order_type` object with this calculated quantity (and any additional parameters passed via `**kwargs`). Finally, it places the order by calling `send_order` with the contract and the newly created order.

Ordering to adjust a position to a target value with `order_target_value`

The `order_target_value` method adjusts a position to a specified target monetary value for a contract. It first calculates the target quantity of contracts needed to reach the target value using the `_calculate_order_value_quantity` method, which determines the quantity based on the current market price and the contract's multiplier. Then, it calculates the actual quantity to order, taking into account the current position, by using the `_calculate_order_target_quantity` method. This method

computes the difference between the target quantity and the current position, treating it as a new order if no position exists or as an adjustment if a position already exists.

The method then creates an order of the specified `order_type` object, setting the action to `BUY` or `SELL` based on whether the calculated quantity is positive or negative, and uses the absolute value of this quantity. Finally, the method places the order by calling `send_order` with the contract and the order.

Ordering to adjust a position to a target percent of the current portfolio value with `order_target_percent`

The `order_target_percent` method adjusts a position so that it represents a specific target percentage of the current portfolio value. This method first calculates the target quantity of the asset needed to achieve the desired portfolio percentage using the `_calculate_order_target_percent_quantity` method. This calculation takes the current value of the portfolio and the target percentage into account, adjusting the position size accordingly. If the position in the specified contract already exists, the method calculates the difference between the current and target percentages and adjusts the position size to match the target.

The method then creates an order of the specified `order_type` object, determining whether to buy or sell based on whether the calculated quantity is positive or negative, and sets the quantity to the absolute value of the calculated amount. Finally, the method places the order by calling `send_order` with the contract and the order.

There's more...

The following examples show how to use the new order methods.

1. In `app.py`, add the following imports at the top of the file:

```
from order import market, limit, BUY, SELL
```

2. In the main section of the file under the instantiation of the trading app, define an AAPL contract:

```
aapl = stock("AAPL", "SMART", "USD")
```

3. Submit a `market` order to buy \$1,000 worth of AAPL:

```
app.order_value(aapl, market, 1000, action=BUY)
```

4. Submit a `market` order to adjust the portfolio to be short five shares of AAPL:

```
app.order_target_quantity(aapl, market, -5)
```

5. Submit a `limit` order for AAPL for an equivalent of 10% of the net liquidation value of the portfolio:

```
app.order_percent(aapl, limit, 0.1, action=BUY,
```

```
    limit_price=185.0)
```

6. Submit a **stop loss** order to adjust the portfolio to \$3,000 worth of AAPL:

```
app.order_target_value(aapl, stop, 3000,  
                      stop_price=180.0)
```

7. Submit a **market** order to adjust the portfolio so that 50% of the net liquidation value is allocated to AAPL:

```
app.order_target_percent(aapl, market, 0.5)
```

See also

The logic to compute the target values and percentages is based on the `zipline-reloaded` implementation for backtesting. You can review the API reference for details at <https://zipline.ml4trading.io/api-reference.html>.

Deploying a monthly factor portfolio strategy

We'll now integrate the momentum factor we built in [Chapter 5, Build Alpha Factors for Stock Portfolios](#), into our trading app. The app is designed to download and process premium U.S. equities data encompassing a comprehensive universe of approximately 20,000 stocks. The advantage of using the premium data is that it lets us build factor portfolios that include the entire universe of U.S.-traded equities.

The trading app is designed to be run on a periodic rebalancing schedule after market hours, typically monthly. Each time it runs, it acquires the latest price data for the entire stock universe. It then computes the momentum factor for these stocks. Based on this computation, the app identifies the top stocks exhibiting the strongest momentum and the bottom stocks showing the weakest momentum. The trading strategy involves going long on the top stocks and short on the bottom stocks.

Our trading app can execute orders that align with a specific target percentage allocation for each stock in the portfolio. This feature simplifies the process of maintaining the desired portfolio balance, requiring only a single line of code to buy or sell shares as needed. This approach lets us rebalance the portfolio so it consistently reflects the target portfolio adjusted for the computed momentum factor.

Top of Form

Bottom of Form

Getting ready

In [Chapter 1](#), *Acquire Free Financial Market Data with Cutting-edge Python Libraries*, we explained how to set up an account with Nasdaq Data Link. For this recipe, we'll rely on a paid subscription to **QuoteMedia's End of Day US Stock Prices**. If you decide against the paid subscription, you can use the free data described in [Chapter 7](#), *Event-Based Backtesting Factor Portfolios with Zipline Reloaded*. This product offers end-of-day prices, dividends, adjustments, and splits for U.S. publicly traded stocks with price history dating back to 1996. The product covers all stocks with a primary listing on NASDAQ, AMEX, NYSE, and ARCA. You can find the page to subscribe at <https://data.nasdaq.com/databases/EOD/data>. Once you are subscribed, you'll be able to use the data through the same API key we set up in [Chapter 1](#), *Acquire Free Financial Market Data with Cutting-edge Python Libraries*.

To ingest the data to build our factor portfolio, we'll need to set up some custom files. Please visit the following URL to get the instructions:

<https://pyquantnews.com/ingest-premium-market-data-with-zipline-reloaded/>

After you have set up the custom extensions that we have linked, create a file named `.env` in the same directory as your `app.py` file. In this file, add the following line:

`DATALINK_API_KEY=<YOUR-API-KEY>`

Replace `<YOUR-API-KEY>` with your Nasdaq Data Link API key, which you acquired in [Chapter 1](#), *Acquire Free Financial Market Data with Cutting-edge Python Libraries*.

How to do it...

We'll integrate the code we built in [Chapter 5](#), *Build Alpha Factors for Stock Portfolios* into our trading app.

1. In `app.py`, add the following imports to the top of the file and reorder them to adhere to PEP8 specifications:

```
import os
import sqlite3
import threading
import time
import empyrical as ep
import exchange_calendars as xcals
import numpy as np
import pandas as pd
from dotenv import load_dotenv
from zipline.data import bundles
from zipline.data.bundles.core import load
from zipline.pipeline import Pipeline
from zipline.pipeline.data import USEquityPricing
from zipline.pipeline.engine import SimplePipelineEngine
from zipline.pipeline.factors import CustomFactor, Returns
from zipline.pipeline.loaders import USEquityPricingLoader
from zipline.utils.run_algo import load_extensions
```

```

from client import IBClient
from contract import stock
from order import market
from wrapper import IBWrapper
load_dotenv()

```

2. Below our **IBApp** class, add the custom momentum factor that **zipline-reloaded** uses to compute the factor value for each asset in our universe:

```

class MomentumFactor(CustomFactor):
    inputs = [USEquityPricing.close, Returns(window_length=126)]
    window_length = 252
    def compute(self, today, assets, out, prices,
                returns):
        out[:] = (
            (prices[-21] - prices[-252]) / prices[-252]
            - (prices[-1] - prices[-21]) / prices[-21]
        ) / np.nanstd(returns, axis=0)

```

3. Now add a function that creates and returns a **zipline-reloaded** pipeline. The pipeline is responsible for creating a DataFrame with the data that we'll use to select which assets to trade:

```

def make_pipeline():
    momentum = MomentumFactor()
    return Pipeline(
        columns={
            "factor": momentum,
            "longs": momentum.top(top_n),
            "shorts": momentum.bottom(top_n),
            "rank": momentum.rank(ascending=False),
        },
    )

```

4. Now, we'll add the code to generate the weights required to rebalance our portfolio. You'll need to add all of the following code in the main section of the app under the **app = IBApp("127.0.0.1", 7497, client_id=11, account="DU*****")** line. Start with creating a session calendar based on the **New York Stock Exchange** to set the lookback time for computing the momentum. The **top_n** variable determines how many long and short positions we want in our portfolio:

```

top_n = 10
xnyse = xcals.get_calendar("XNYS")
today = pd.Timestamp.today().strftime("%Y-%m-%d")
start_date = xnyse.session_offset(today,
                                   count=-252).strftime("%Y-%m-%d")

```

5. Add the code to load the extension file that we created in the *Getting ready* section of this recipe. After the extension file has been loaded, we can invoke the code to ingest our data bundle using the premium data subscription. Finally, we will load the data bundle into memory so it's ready to use:

```

load_extensions(True, [], False, os.environ)
bundles.ingest("quotemedia")
bundle_data = load("quotemedia", os.environ, None)

```

The ingestion can take around 30 minutes once run, so be patient. You should see output resembling the following once the bundling process begins:

```
[2023-12-19T17:03:59-0700-INFO][zipline.data.bundles.core]
Ingesting quotemedia
[########################################] 100%
Merging daily equity files: [#-----] 1293
```

Figure 12.2: The data ingestion process output to the console

6. Add the code that creates the U.S. equity pricing loader, which aligns the price data to U.S. exchange calendars:

```
pipeline_loader = USEquityPricingLoader(
    bundle_data.equity_daily_bar_reader,
    bundle_data.adjustment_reader,
    fx_reader=None
)
```

7. Create the engine that brings the pipeline loader and the bundled data together:

```
engine = SimplePipelineEngine(
    get_loader=lambda col: pipeline_loader,
    asset_finder=bundle_data.asset_finder
)
```

8. Add the code that runs the pipeline and generates the output:

```
results = engine.run_pipeline(
    make_pipeline(),
    start_date,
    today
)
```

9. Clean up the resulting DataFrame by removing records with no factor data, adding names to the MultiIndexes, and sorting the values first by date, then by factor value:

```
results.dropna(subset="factor", inplace=True)
results.index.names = ["date", "symbol"]
results.sort_values(by=["date", "factor"],
    inplace=True)
```

The result is a MultiIndex DataFrame including the raw factor value, boolean values indicating a short or long position, and details on how the stock's factor value is ranked among the universe. The last date in the DataFrame should reflect the last trading day which will allow us to use the results to rebalance our portfolio:

			factor	longs	shorts	rank
	date	symbol				
2022-12-14	Equity(20942 [XBIOW])	-109.975559	False	True	10339.0	
	Equity(16489 [ROCGW])	-65.102756	False	True	10338.0	
	Equity(3014 [BWACW])	-55.947030	False	True	10337.0	
	Equity(18846 [TGVCW])	-40.394871	False	True	10336.0	
	Equity(20099 [VIIAW])	-38.196519	False	True	10335.0	
...
2023-12-15	Equity(1113 [APCA])	15.747811	True	False	5.0	
	Equity(11471 [LBBB])	16.065618	True	False	4.0	
	Equity(15639 [PUCK])	20.079347	True	False	3.0	
	Equity(10485 [IVCB])	20.102028	True	False	2.0	
	Equity(14991 [PHYT])	22.416517	True	False	1.0	

Figure 12.3: A MultiIndex DataFrame with factor information and trading indicators

10. Use the `query` method on the `results` MultiIndex DataFrame to select the long and short positions that will make up our portfolio:

```
longs = results.xs(today, level=0).query(
    "longs == True")
shorts = results.xs(today, level=0).query(
    "shorts == True")
```

11. Create a simple equal-weight to determine how to allocate the assets in our portfolio:

```
weight = 1 / top_n / 2
```

12. Add the code to loop through each of the `longs` and `shorts` DataFrames and order a target percent of each asset based on the weight:

```
for row in pd.concat([longs, shorts]).itertuples():
    side = 1 if row.longs else -1
    symbol = row.Index.symbol
    contract = stock(symbol, "SMART", "USD")
    app.order_target_percent(contract, market,
        side * weight)
```

How it works...

The mechanics of the `zipline-reloaded` pipeline API are covered extensively in [Chapter 5, Build Alpha Factors for Stock Portfolios](#). The contribution of this chapter is to isolate the stocks that we

want to go long and short, generate a simple weighting scheme, and send the orders for the target percentage.

The `longs` and `shorts` variables are derived from a `results` DataFrame. For a specific date, the `xs` method extracts a subset of the DataFrame at a `date` level, and the `query` method filters this subset to include only rows where `longs` or `shorts` are `True`, respectively.

The `weight` variable is calculated as the inverse of twice the number of top stocks (`top_n`) to be traded, creating an equally distributed allocation among all the stocks.

The `for` loop iterates over each row of the concatenated `longs` and `shorts` DataFrames. For each row, it determines the side of the trade: `1` for long positions and `-1` for short positions. The symbol of the stock is extracted from the row's index.

For each stock symbol, a `contract` object is created using the `stock` function, which specifies the stock symbol, the exchange, and the currency. Finally, the `order_target_percent` method is called for each stock, setting the target percentage of the portfolio to be allocated to this stock. The percentage is calculated by multiplying the integer `side` by the `weight` float, effectively scaling the allocation based on whether the position is long or short.

There's more...

You may want to include a screen while building the universe for which to compute momentum and rank. One way to do that is to filter stocks based on their daily average dollar volume, which helps us focus on more liquid and potentially less volatile assets. We can use the `AverageDollarVolume` class, which includes only those stocks that exceed a specified threshold in daily average dollar volume.

To do it, import the class:

```
from zipline.pipeline.factors import AverageDollarVolume
```

Then update the `make_pipeline` function to resemble the following:

```
def make_pipeline():
    momentum = MomentumFactor()
    dollar_volume = AverageDollarVolume(window_length=21)
    return Pipeline(
        columns={
            "factor": momentum,
            "longs": momentum.top(top_n),
            "shorts": momentum.bottom(top_n),
            "rank": momentum.rank(ascending=False),
        },
        screen=dollar_volume.top(100)
    )
```

The screen will apply the filter after it computes the momentum value and ranks the stocks. That means that even though we asked for the `top_n` number of assets to include in the pipeline, we may end up with fewer than `top_n` if they are screened out after ranking.

See also

Learn how to use the Pipeline API in a `zipline-reloaded` backtest at

<https://www.pyquantnews.com/the-pyquant-newsletter/backtest-a-custom-momentum-strategy-with-zipline>.

Deploying an options combo strategy

Options can provide algorithmic traders with the ability to express market views that go beyond the linear profit and loss potential of stock trading. Unlike stocks, whose profit or loss is directly tied to the movement of the price on a dollar-for-dollar basis, options are non-linear instruments whose value is affected by multiple dimensions such as the price of the underlying asset, time to expiration, volatility, and interest rates. This multidimensionality allows for strategies that can profit from a range of outcomes.

A straddle strategy includes buying a call and a put option with the same strike prices and expiration dates. This lets traders speculate on the asset's volatility instead of predicting the market's directional movement. This approach makes money for the trader when the underlying asset experiences substantial movement in any direction.

Premium collection strategies such as a short iron condor involve selling both a call and a put spread with the goal of earning the premium from the options sold. This is predicated on the belief that the underlying asset's price will remain within a specific range, leading to the options expiring worthless and the trader keeping the premium collected.

In this recipe, you'll learn how to model options trades in **Trader Workstation (TWS)** and submit the orders using Python through the IB API.

Getting ready

We need to add two functions to our `contract.py` file. Open it and add the following code at the end of the file:

```
def combo_leg(contract_details, ratio, action):
    leg = ComboLeg()
    leg.conId = contract_details.contract.conId
```

```

leg.ratio = ratio
leg.action = action
leg.exchange = contract_details.contract.exchange
return leg
def spread(legs):
    contract = Contract()
    contract.symbol = "USD"
    contract.secType = "BAG"
    contract.currency = "USD"
    contract.exchange = "SMART"
    contract.comboLegs = legs
    return contract

```

These two functions allow us to create the legs of a combo order and use those legs to construct a single tradeable instrument.

Note that in the `combo_leg` method, we need to pass in a contract's `conId` string. The `conId` string is a unique identifier maintained by IB. To get the `conId` string, we have to send a request to IB to get the details of that contract maintained by IB. To do that, we use an IB method called `reqContractDetails`. In the `client.py` file, add the following method directly under the `__init__` method in the `IBClient` class:

```

def resolve_contract(self, contract,
                     request_id=DEFAULT_CONTRACT_ID):
    self.reqContractDetails(reqId=request_id,
                           contract=contract)
    time.sleep(2)
    self.contractDetailsEnd(reqId=request_id)
    return self.resolved_contract

```

This method takes a contract object, requests the details from IB, and returns its details.

Let's see how it works.

How to do it...

We'll construct an options position called a **strangle**, which will let us bet on an increase in the volatility of the underlying. The following code should be added under the creation of the app under the `if __name__ == "__main__":` line in `app.py`.

1. Add the code to create a long call option contract on TSLA expiring in March 2024 with a strike price of \$260:

```

long_call_contract = option("TSLA", "SMART", "202403",
                           260, "CALL")
long_call = app.resolve_contract(long_call_contract)

```

2. Add the code to create a long put option contract on TSLA expiring on the same date with the same strike price:

```

long_put_contract = option("TSLA", "SMART", "202404",
                           260, "PUT")
long_put = app.resolve_contract(long_put_contract)

```

3. Convert each resolved contract into a leg of our spread contract:

```

leg_1 = combo_leg(long_call, 1, BUY)
leg_2 = combo_leg(long_put, 1, BUY)

```

4. Pass a list of the legs to the `spread` function to put the legs together into a single tradeable instrument:

```
long_strangle = spread([leg_1, leg_2])
```

5. Create a market order object and send the order to the IB API:

```

order = market(BUY, 1)
app.send_order(long_strangle, order)

```

How it works...

The `combo_leg` function takes three parameters: `contract_details`, `ratio`, and `action`. It creates an instance of `ComboLeg` and sets `conId`, `ratio`, `action`, and `exchange`. The fully defined `ComboLeg` object is then returned. The `spread` function takes a list of `ComboLeg` objects created by the `combo_leg` function. The function creates an instance of `Contract`, sets its `symbol` to `USD`, `type` to `BAG` (which stands for a **bag** or combination of contracts), `currency` to `USD`, and `exchange` to `SMART`. It then assigns the `legs` parameter to the `comboLegs` attribute of the contract. The complete `contract` object is then returned.

In this example, we constructed a long straddle. A long options straddle is a market-neutral strategy whereby we purchase both a call option and a put option on the same underlying asset with the same strike price and expiration date. Straddles allow us to bet on volatility rather than the direction of the price movement. The position profits if the underlying asset moves up or down, allowing the trader to exercise one of the options for a profit that exceeds the total cost of both premiums.

We can model options trades in TWS. To get started, click the plus sign to the right of the tabs at the bottom of TWS. Click **Browse** on the right side of the popup to browse predefined layouts. Click **Options** on the left side, then select **Add Layout** on the **Options Trading** card in the top left. Once this has been added, load your favorite underlying. In the **Options Chains** window, toggle **Strategy Builder**.

For example, using the Strategy Builder, we can construct our straddle.

ACTN	RT	LST TRD DAY	STRIKE	TYPE	DELTA	THETA	BID/ASK	SIZE
x Leg 1	Buy	▼ 1 MAR 15 '24	▼ 260	▼ Call	0.548	-0.149	23.10x23.35	957x16
x Leg 2	Buy	▼ 1 MAR 15 '24	▼ 260	▼ Put	-0.460	-0.114	22.55x22.85	1,401x17
Mar'24 260 Straddle							45.65x46.20	1,401x17

Figure 12.4: A long straddle modeled using the Strategy Builder in TWS

By clicking the **Profile** button just under the **Strategy Builder** window, we can get more details about the strategy including the payoff charts as shown in the following screenshot:

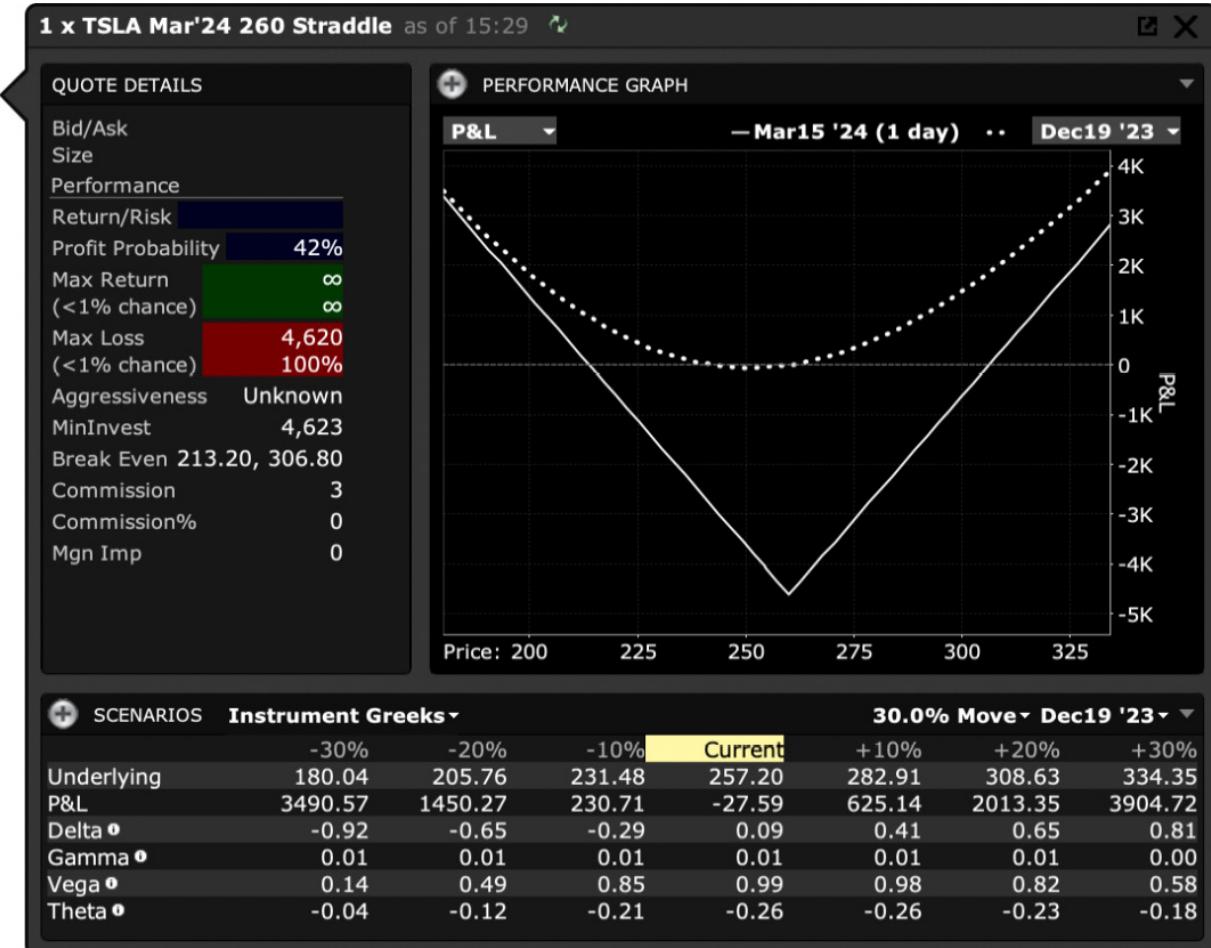


Figure 12.5: A long straddle profile in TWS

After running our app, the legs of the trade will be grouped together in the orders pane as follows:

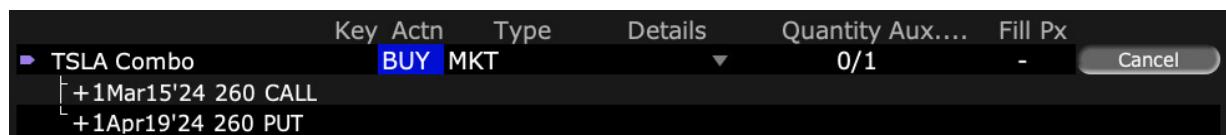


Figure 12.6: Long straddle legs grouped together in the orders pane in TWS

There's more...

Let's model another, more complex strategy called an iron condor. We'll describe a short iron condor. A short iron condor is executed by selling a put option at a lower out-of-the-money strike, purchasing another put with an even lower strike, selling a call option at a higher out-of-the-money strike, and then buying another call with an even higher strike. Typically, all options have the same expiration date. The position is constructed in such a way that we collect the premium from the options sold. The bet is that the underlying asset's price will remain between the inner strikes of the options sold.

and the options expire worthless. This strategy allows traders to bet on low volatility in the underlying asset's price, aiming to profit from a range-bound market within the span of the options' expiration period.

This strategy is beneficial in a low-volatility environment, where large movements are not expected. It's a limited-risk strategy, as the long options of the spreads act as insurance against substantial adverse moves in the asset's price. The profitability is capped to the premiums received, while losses are limited to the difference between the strikes less the net premium collected.

We can model options trades in TWS. For example, using the Strategy Builder, we can construct our straddle as shown here:

	ACTN	RT	LST TRD DAY	STRIKE	TYPE	DELTA	THETA	BID/ASK	SIZE	
✗ Leg 1	Sell ▾		1 MAR 15 '24	▼ 295	▼ Call ▾	0.326	-0.130	10.80x10.95	198x218	Closed
✗ Leg 2	Buy ▾		1 MAR 15 '24	▼ 305	▼ Call ▾	0.274	-0.119	8.60x8.75	203x206	Closed
✗ Leg 3	Sell ▾		1 MAR 15 '24	▼ 240	▼ Put ▾	-0.322	-0.108	13.75x13.85	2x89	Closed
✗ Leg 4	Buy ▾		1 MAR 15 '24	▼ 233....	▼ Put ▾	-0.280	-0.104	11.40x11.55	63x171	Closed
Total								-4.80x-4.25	203x206	

Figure 12.7: A short iron condor modeled using the Strategy Builder in TWS

By clicking the **Profile** button just under the **Strategy Builder** window, we can get more details about the strategy, including the payoff charts.

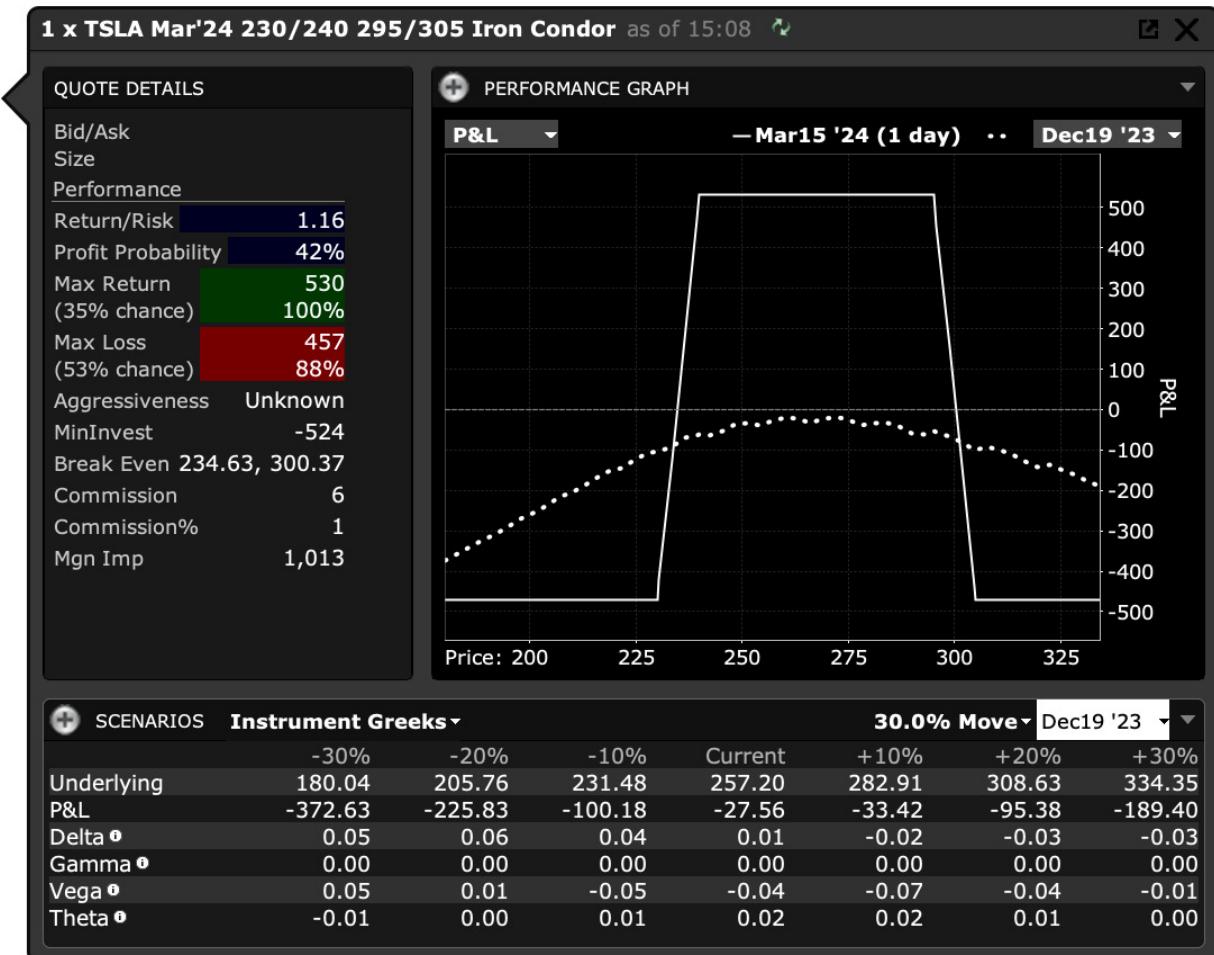


Figure 12.8: A short iron condor profile in TWS

After running our app, the legs of the trade will be grouped together in the **Orders** pane.

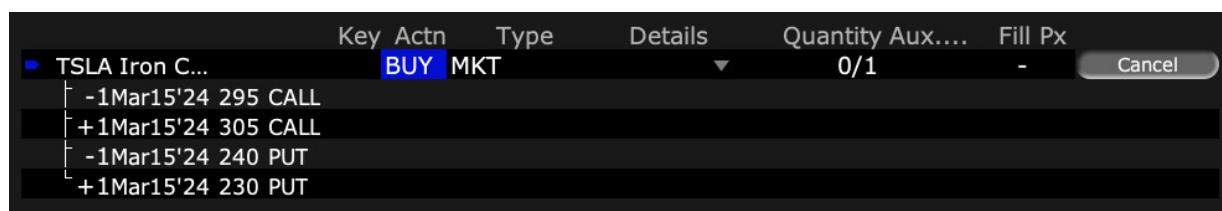


Figure 12.9: Short iron condor legs grouped together in the orders pane in TWS

See also

To learn more about options trading, you can check out the Options Industry Council's website at <https://www.optionseducation.org/>. A great desk reference is one of the most comprehensive guides to options trading on the market: *Options as a Strategic Investment*, which can be found on Amazon: <https://amzn.to/3GQRspX>

For more information on options spread orders using the IB API, see the documentation:
https://interactivebrokers.github.io/tws-api/spread_contracts.html#bag_opt

Deploying an intraday multi-asset mean reversion strategy

In this recipe, we introduce a relative value strategy that uses the crack spread to identify opportunities to buy relatively cheap refiner stocks. The crack spread is a trading strategy in the energy sector that captures the price differential between crude oil and its refined products, most commonly gasoline and heating oil. Our strategy will use a 1:2:3 combination of one contract of **Heating Oil (HO)** futures, two contracts of **NYMEX RBOB Gasoline Index (RB)** futures, and short three **Light Sweet Crude Oil (CL)** futures.

When trading the crack spread against a refiner stock, we are comparing the profitability of refining operations to the market's valuation of a specific refining company. Refiner stocks should theoretically benefit from a widening crack spread, as their margins improve when the price difference between crude and its products increases. The economic rationale behind our strategy is straightforward: refiners purchase crude oil and sell its refined products. When the spread between the two widens, the refiner's profit margin expands. Consequently, refiner stocks should move in tandem with crack spreads. Other factors such as operational efficiency, regulatory changes, and broader market sentiment can cause the relationship to break down, which we can take advantage of.

In this recipe, we'll build a strategy that identifies times when the crack spread widens substantially but a refiner's stock hasn't reacted proportionally. We'll use the opportunity to go long **Philips 66 (PSX)**, expecting the stock to catch up to the increased profitability suggested by the crack spread.

Getting ready

We assume that you've created the `app.py` files in the `trading-app` directory. If you have not, do it now.

How to do it...

We'll add the code required to connect to TWS, define the contracts for our strategy, check for trading signals every 60 seconds, and execute trades accordingly. Start by opening `app.py` and replacing any code under the `if __name__ == "__main__":` line with the following.

1. Create the connection to our trading app, replacing `DU*****` with your account number:

```
app = IBApp("127.0.0.1", 7497, client_id=11,
            account="DU*****")
```

2. Define the refiner stock that we'll trade (in this case, PSX) and the futures contracts that make up the crack spread:

```
psx = stock("PSX", "SMART", "USD")
ho = future("HO", "NYMEX", "202403")
rb = future("RB", "NYMEX", "202403")
cl = future("CL", "NYMEX", "202403")
```

3. Set up a window of time to compute a rolling z-score that we'll use as our trading signal along with the number of standard deviations that trigger a trade:

```
window = 60
thresh = 2
```

4. We'll now set up an infinite loop that will continuously run throughout the trading day. The rest of the code should be under a **while** statement:

```
while True:
```

5. Download the historical data for the four contracts that we just defined. In our example, we'll get one-minute bars over the last week. This will give us flexibility in case we want to extend our lookback window:

```
data = app.get_historical_data_for_many(
    request_id=99,
    contracts=[psx, ho, rb, cl],
    duration="1 W",
    bar_size="1 min",
).dropna()
```

6. Compute the 1:2:3 crack spread and add the data to a new column in our DataFrame that contains the price data:

```
data["crack_spread"] = data.HO + 2 * data.RB - 3 * data.CL
```

7. To normalize the value of the crack spread and the refiner, compute the rolling 60-period rank of the last value as a percentage of the preceding values:

```
data["crack_spread_rank"] = data.crack_spread.rolling(
    window).rank(pct=True)
data["refiner_rank"] = data.PSX.rolling(window).rank(pct=True)
```

8. Compute the difference between the crack spread's 60-period rank and the refiner's 60-period rank:

```
data["rank_spread"] = data.refiner_rank - data.crack_spread_rank
```

9. Compute the rolling 60-period z-score of the difference between the crack spread's rank and the refiner's rank and take the last value as the trading signal:

```
roll = data.rank_spread.rolling(window)
zscore = ((data.rank_spread - roll.mean()) / roll.std())
signal = zscore[-1]
```

10. Enter a market order to buy 10 shares of PSX if the current signal is less than the threshold and we do not currently hold a position in PSX. Set the target percent of PSX to 0% if our signal exceeds 0 and we do not currently hold a position in PSX:

```
holding = psx.symbol in app.positions.keys()
```

```

if signal <= -thresh and not holding:
    order = market(BUY, 10)
    app.send_order(psx, order)
elif signal >= 0 and holding:
    app.order_target_percent(psx, market, 0)

```

11. Enter a market order to sell 10 shares of PSX if the current signal is greater than the threshold and we do not currently hold a position in PSX. Set the target percent of PSX to 0% if our signal goes below 0 and we do not currently hold a position in PSX:

```

if signal >= thresh and not holding:
    order = market(SELL, 10)
    app.send_order(psx, order)
elif signal <= 0 and holding:
    app.order_target_percent(psx, market, 0)

```

12. Finally, wait for 60 seconds before running the process again:

```
time.sleep(60)
```

The final result should resemble the following in `app.py`:

```

if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=11,
                account="DU7129120")
    psx = stock("PSX", "SMART", "USD")
    ho = future("HO", "NYMEX", "202403")
    rb = future("RB", "NYMEX", "202403")
    cl = future("CL", "NYMEX", "202403")
    window = 60
    thresh = 2
    while True:
        data = app.get_historical_data_for_many(
            request_id=99,
            contracts=[psx, ho, rb, cl],
            duration="1 W",
            bar_size="1 min", ).dropna()
        data["crack_spread"] = data.HO + 2 * data.RB - 3 * data.
        CL
        data["crack_spread_rank"] = data.crack_spread.rolling(
            window).rank(pct=True)
        data["refiner_rank"] = data.PSX.rolling(
            window).rank(pct=True)
        data["rank_spread"] = data.refiner_rank - data.crack_
        spread_rank
        roll = data.rank_spread.rolling(window)
        zscore = ((data.rank_spread - roll.mean()) / roll.std())
        signal = zscore[-1]
        holding = psx.symbol in app.positions.keys()
        if signal <= -thresh and not holding:
            order = market(BUY, 10)
            app.send_order(psx, order)
        elif signal >= 0 and holding:
            app.order_target_percent(psx, market, 0)
        if signal >= thresh and not holding:
            order = market(SELL, 10)
            app.send_order(psx, order)
        elif signal <= 0 and holding:
            app.order_target_percent(psx, market, 0)
        time.sleep(60)
    app.disconnect()

```

How it works...

We covered connecting to TWS, defining contract objects, downloading historical data, manipulating data in pandas DataFrames, and submitting orders in previous chapters. Here, we'll focus on how the trading strategy works conceptually.

To run this strategy, we would need to start it at the beginning of the trading day. The algorithm operates in a continuous loop, fetching historical data for the contracts with a one-week duration and a one-minute bar size. For each loop, we calculate the crack spread, which is then ranked over a rolling window along with the rank of PSX stock's performance. The difference between these ranks is used to generate a trading signal. This signal is a z-score, calculated over a rolling window, indicating how many standard deviations the current rank spread is from its rolling mean.

We use the `get_historical_data_for_many` method that we created in [Chapter 10, Set up the Interactive Brokers Python API](#). This method returns a DataFrame with the open, high, low, and closing prices for the PSX equity contract and HO, RB, and CL futures contracts.

IMPORTANT NOTE

As of the time of writing this, the March 2024 contract was the most active contract. You may need to adjust the expiration of the contracts used during your own implementation of this strategy.

The result of acquiring the data is the following pandas DataFrame.

symbol		CL	HO	PSX	RB
time					
2023-12-13	09:30:00-05:00	69.36	2.4674	124.78	2.0143
2023-12-13	09:31:00-05:00	69.32	2.4649	124.75	2.0133
2023-12-13	09:32:00-05:00	69.39	2.4677	124.77	2.0155
2023-12-13	09:33:00-05:00	69.47	2.4703	124.94	2.0170
2023-12-13	09:34:00-05:00	69.47	2.4700	124.79	2.0166
...	
2023-12-13	15:55:00-05:00	70.29	2.4908	127.03	2.0638
2023-12-13	15:56:00-05:00	70.29	2.4910	127.09	2.0642
2023-12-13	15:57:00-05:00	70.31	2.4920	127.09	2.0641
2023-12-13	15:58:00-05:00	70.33	2.4919	127.10	2.0646
2023-12-13	15:59:00-05:00	70.36	2.4930	127.16	2.0654

Figure 12.10: A pandas DataFrame with historical market data

Now that we have the price data, we're able to compute the crack spread using the same data manipulation techniques that we learned in [Chapter 2, Analyze and Transform Financial Market](#)

Data with pandas. The result of computing the crack spread is a time series that resembles the following chart.

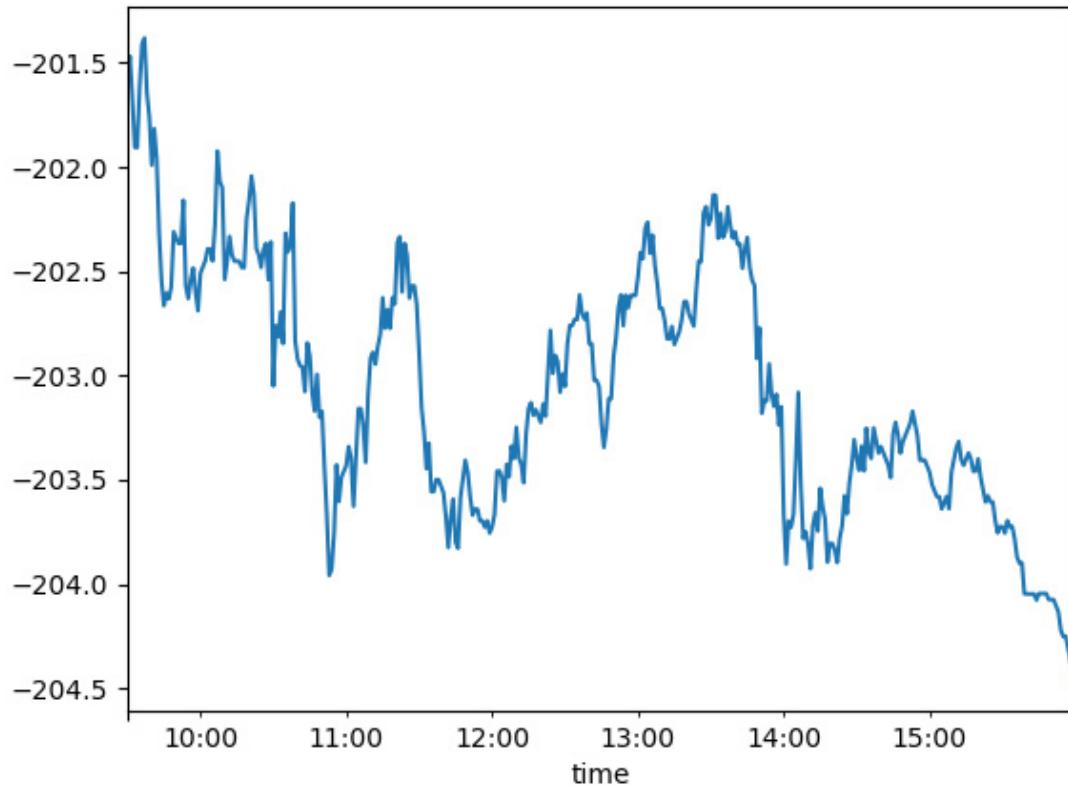


Figure 12.11: A 1:2:3 crack spread plotted over a single trading day

The prices of equities do not go negative, so we need a way to compare the relative value of the crack spread to the refiner that we're trading against. One way to do this is to determine the percent rank of the current price relative to the preceding prices. We use that technique for both the crack spread and the refiner stock. The result of these rolling percentile ranks is two time series that resemble the following chart.

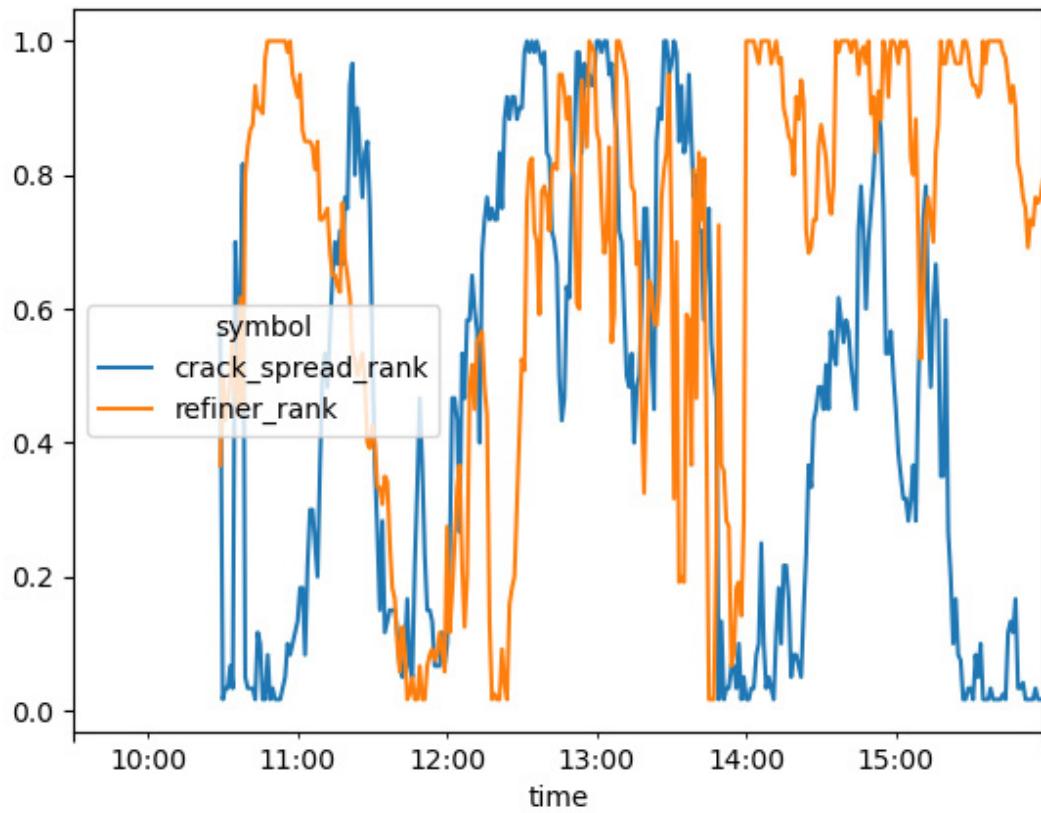


Figure 12.12: The evolution of the price rank of PSX and the crack spread relative to their 60-period windows

Now that the refiner and crack spread prices have been normalized, we can compare them by taking the difference between the two. The result of this difference is a time series that resembles the following chart.

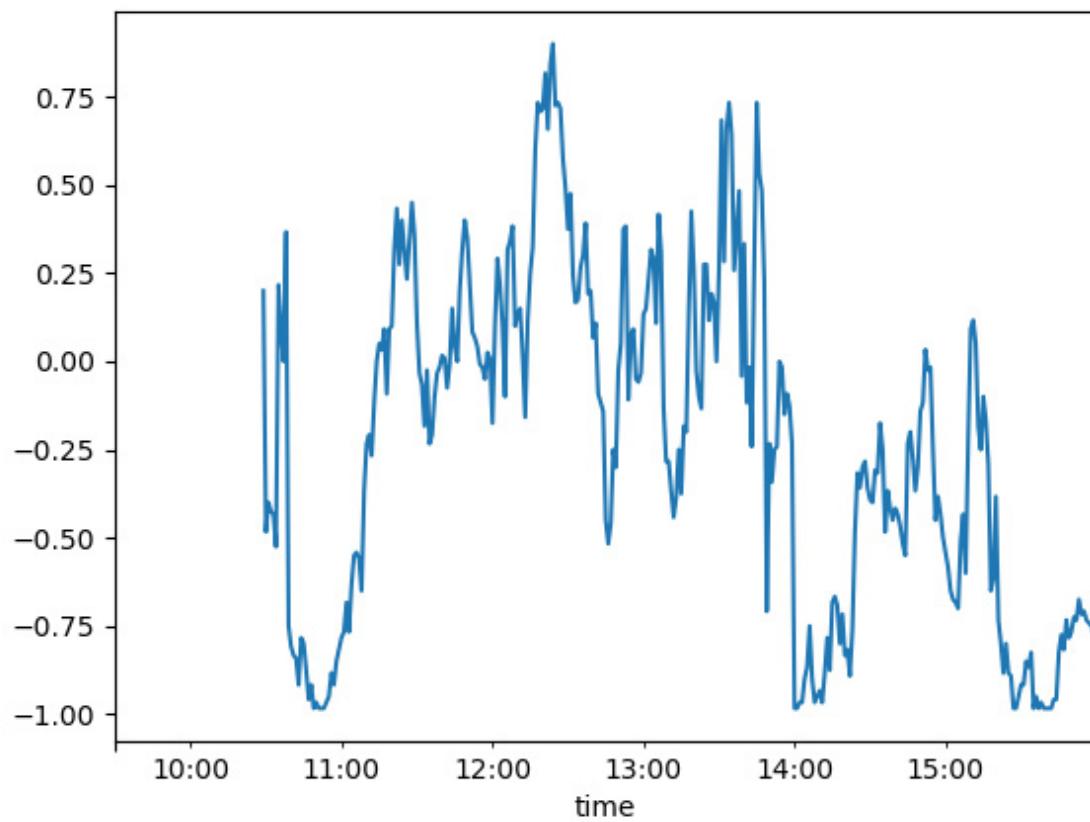


Figure 12.13: The difference between the crack spread's rank and the refiner's rank

We can see some mean reverting behavior, which we can more concisely capture through a z-score. Computing the z-score creates a time series that resembles the following chart.

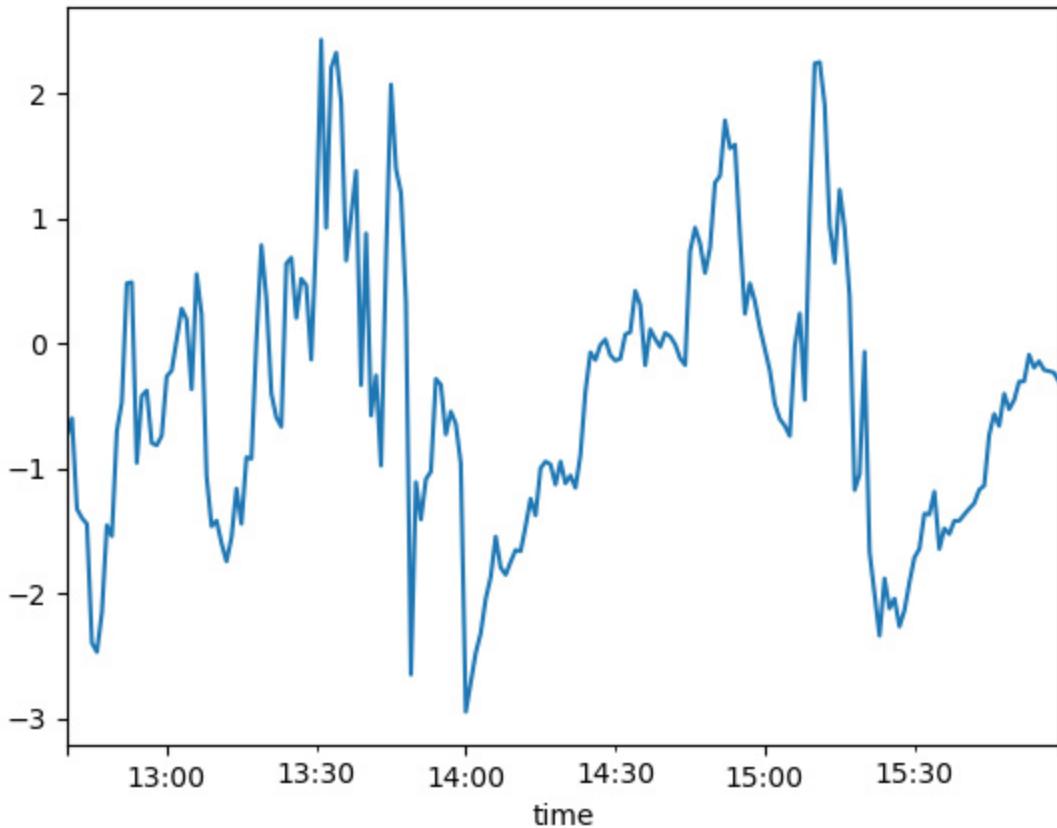


Figure 12.14: A rolling z-score of the differences between the crack spread's rank and the refiner's rank

Trading decisions are made based on this z-score. If the signal is below a negative threshold and PSX is not already held, we enter a buy order. If the signal is non-negative and PSX is held, the position is liquidated to 0. Conversely, if the signal exceeds a positive threshold and PSX is not held, we enter a `sell` order. If the signal is negative and PSX is held, the position is again liquidated. The script pauses for 60 seconds at the end of each loop iteration. This process repeats until we stop execution.

There's more...

In this strategy, we use the crack spread as a relative value indicator. We assume that the relative value of the refiner and the crack spread will oscillate around a mean, which we can exploit. We can also create a spread order and trade the crack spread itself. The crack spread is a type of intercommodity spread by IB.

Intercommodity spreads for futures contracts are constructed using combo legs and combo orders, which allows us to simultaneously trade multiple futures contracts as a single entity. This approach is particularly relevant for strategies such as crack spreads, whereby the profit is derived from the price

differences between related commodities such as crude oil and its refined products. When building these spreads, each leg of the combo order represents a different futures contract, and it's crucial to specify the **local symbol** accurately for each contract. The local symbol is a unique identifier for futures contracts that includes information about the underlying asset, expiration date, and other contract specifics.

IB currently does not support intercommodity spreads for crack spread strategies, which prevents us from directly implementing this particular strategy on their platform. IB does support intercommodity spreads between CL and RB, as well as between RB and HO. We could technically construct a crack spread using a combination of these two intercommodity spreads but the complexity involved makes it easier to just trade the refiner stock.

See also

To learn more about the intuition behind our trade, you can read about the crack spread at
<https://www.investopedia.com/terms/c/crackspread.asp>.

You can read more about building combo legs and futures spread contracts using the IB API at
https://interactivebrokers.github.io/tws-api/spread_contracts.html#bag_fut.

Advanced Recipes for Market Data and Strategy Management

This final chapter covers advanced recipes to stream and store options data, generate risk alerts, and store key strategy information to automate end-of-day reporting. We will start with a deep dive into real-time data handling using **Theta Data**. ThetaData is a data service that specializes in providing real-time options data. It offers a comprehensive stream of unfiltered options market data, including quotes, trades, volumes, and Greeks. With ThetaData, we can combine contracts to price complex options positions in real time. This service is an option for algorithmic traders who want to research and develop complex trading strategies using options contracts. After streaming the data, we will introduce advanced data management storage using ArcticDB. ArcticDB is an open source project built by the systematic strategy manager **Man Group** and is designed to store petabytes of data in DataFrame format.

In [Chapter 12, Deploy Strategies to a Live Environment](#), we built a series of risk and performance metrics. In this chapter, we'll design an alerting system that will send an email if a defined risk level is breached. This approach is popular in professional trading businesses where traders must adhere to defined risk limits. We'll then introduce recipes to store key strategy information in the SQL database we created in [Chapter 10, Set Up the Interactive Brokers API](#), automating end-of-day strategy management processes.

By the end of the chapter, you will have the tools to handle and analyze real-time data, manage risk more carefully, and maintain detailed trade records. All techniques that are instrumental for algorithmic traders.

In this chapter, we will present the following recipes:

- Streaming real-time options data with ThetaData
- Using the ArcticDB DataFrame database for tick storage
- Triggering real-time risk limit alerts
- Storing trade execution details in a SQL database

Streaming real-time options data with ThetaData

The **Options Price Reporting Authority (OPRA)** functions as a securities information processor, aggregating options quotes and transaction details from predominant U.S. exchanges. Approximately

1.4 million active options contracts are traded, generating in excess of 3 terabytes of data on a daily basis. OPRA is responsible for the real-time consolidation and dissemination of this data. ThetaData, through its connection to OPRA, facilitates the distribution of this data in an unfiltered format to non-professional users. Furthermore, ThetaData's Python API is capable of streaming quotes and trades with a latency measured in milliseconds. This efficiency is achieved by compressing the data to approximately 1/30th of its original volume.

The Theta Terminal is an intermediate layer that bridges our data-providing server with the Python API. The terminal runs as a background process. It hosts a local server on your machine, to which the Python API connects. Primarily, it simplifies data access and processing by handling complex tasks such as forwarding requests to the appropriate server and interpreting responses. Additionally, separating data processing activities like decompression from API-specific features enhances efficiency and usability for traders using the service. This recipe will demonstrate how to use ThetaData for streaming options data.

Getting ready

To use ThetaData, you'll need Java installed on your computer. Java usually comes pre-installed but in case it's not, you can find the installation process for your computer at <https://www.java.com/en/>.

To install the ThetaData Python library, use `pip`:

```
pip install thetadata
```

How to do it...

We'll demonstrate how to stream real-time trade data for all options contracts, as well as for a single options contract:

1. Import the libraries we need to set up the streaming data:

```
import datetime as dt
import thetadata.client
from thetadata import (
    Quote,
    StreamMsg,
    ThetaClient,
    OptionRight,
    StreamMsgType,
    StreamResponseType
)
```

2. Implement the callback that responds to each trade message. In this example, we will simply print information about the contract, trade, and quote:

```

def callback(msg):
    if msg.type == StreamMsgType.TRADE:
        print(
            "-----"
        )
        print(f"Contract: {msg.contract.to_string()}")
        print(f"Trade: {msg.trade.to_string()}")
        print(f"Last quote at time of trade: {
            msg.quote.to_string()}")

```

3. Implement the function that connects to the ThetaData client, registers the callback, and starts the options data stream (replace **YOURPASSWORD** with your ThetaData login credentials):

```

def stream_all_trades():
    client = ThetaClient(
        username="strimp101@gmail.com",
        passwd="YOURPASSWORD"
    )
    client.connect_stream(
        callback
    )
    req_id = client.req_full_trade_stream_opt()
    response = client.verify(req_id)
    if (
        client.verify(req_id) != StreamResponseType.SUBSCRIBED
    ):
        raise Exception("Unable to stream.")

```

4. Start the streaming data:

```
stream_all_trades()
```

After ThetaData connects, you'll start to see options trades print to the screen:

```

-----
Contract: root: KWEB isOption: True exp: 2024-06-21 strike: 30.0 isCall: True
Trade: ms_of_day: 46545319 sequence: 3754634011 size: 5 condition: SINGLE_LEG_AUCTION_NON_ISO price: 1.47 exchange:
XBOS date: 2023-12-22
Last quote at time of trade: ms_of_day: 46368226 bid_size: 2 bid_exchange: ARCX bid_price: 1.45 bid_condition: NATI
ONAL_BBO ask_size: 10 ask_exchange: PERL ask_price: 1.48 ask_condition: NATIONAL_BBO date: 2023-12-22

Contract: root: KWEB isOption: True exp: 2024-06-21 strike: 30.0 isCall: True
Trade: ms_of_day: 46545319 sequence: 3754634012 size: 5 condition: SINGLE_LEG_AUCTION_NON_ISO price: 1.47 exchange:
XBOS date: 2023-12-22
Last quote at time of trade: ms_of_day: 46368226 bid_size: 2 bid_exchange: ARCX bid_price: 1.45 bid_condition: NATI
ONAL_BBO ask_size: 10 ask_exchange: PERL ask_price: 1.48 ask_condition: NATIONAL_BBO date: 2023-12-22

Contract: root: CROX isOption: True exp: 2023-12-22 strike: 99.0 isCall: True
Trade: ms_of_day: 45375636 sequence: 4173523168 size: 1 condition: AUTO_EXECUTION price: 0.8 exchange: XBOS date: 2
023-12-22
Last quote at time of trade: ms_of_day: 45371916 bid_size: 2 bid_exchange: PERL bid_price: 0.75 bid_condition: NATI
ONAL_BBO ask_size: 1 ask_exchange: XBOS ask_price: 0.8 ask_condition: NATIONAL_BBO date: 2023-12-22
-----
```

Figure 13.1: The firehose of options trades

5. Define an active options contract. In this case, we are focused on **SPY** options with a \$474 strike expiring on December 12, 2023:

```

ticker = "SPY"
expiration_date = dt.date(2023, 12, 22)
strike = 474

```

6. Implement the function that connects to the ThetaData client, registers the callback, and starts the options data stream for the specific contract we define (replace **youremail@example.com** and **YOURPASSWORD** with your ThetaData login credentials):

```

def stream_contract():
    client = ThetaClient(
        username="youremail@example.com",
        passwd=" YOURPASSWORD "
    )
    client.connect_stream(callback)
    req_id = client.req_trade_stream_opt(
        ticker, expiration_date, strike, OptionRight.CALL)
    response = client.verify(req_id)
    if (
        client.verify(req_id) != StreamResponseType.SUBSCRIBED
    ):
        raise Exception("Unable to stream.")

```

7. Start the streaming data:

```
stream_contract()
```

After ThetaData connects, you'll start to see options trades print to the screen for the contract you specified:

```

Contract: root: SPY isOption: True exp: 2023-12-22 strike: 474.0 isCall: True
Trade: ms_of_day: 46841281 sequence: 2793706501 size: 4 condition: AUTO_EXECUTION price: 0.95 exchange: EDGX date:
2023-12-22
Last quote at time of trade: ms_of_day: 46841109 bid_size: 168 bid_exchange: ARCX bid_price: 0.95 bid_condition: NA
TIONAL_BBO ask_size: 57 ask_exchange: PERL ask_price: 0.96 ask_condition: NATIONAL_BBO date: 2023-12-22

Contract: root: SPY isOption: True exp: 2023-12-22 strike: 474.0 isCall: True
Trade: ms_of_day: 46841046 sequence: 2793716130 size: 1 condition: AUTO_EXECUTION price: 0.96 exchange: BATS date:
2023-12-22
Last quote at time of trade: ms_of_day: 46841046 bid_size: 276 bid_exchange: EDGX bid_price: 0.95 bid_condition: NA
TIONAL_BBO ask_size: 205 ask_exchange: ARCX ask_price: 0.97 ask_condition: NATIONAL_BBO date: 2023-12-22

Contract: root: SPY isOption: True exp: 2023-12-22 strike: 474.0 isCall: True
Trade: ms_of_day: 46843223 sequence: 2793741289 size: 4 condition: AUTO_EXECUTION price: 0.97 exchange: ARCX date:
2023-12-22
Last quote at time of trade: ms_of_day: 46843220 bid_size: 126 bid_exchange: ARCX bid_price: 0.96 bid_condition: NA
TIONAL_BBO ask_size: 5 ask_exchange: ARCX ask_price: 0.97 ask_condition: NATIONAL_BBO date: 2023-12-22

```

Figure 13.2: Options trade data for a specific contract

How it works...

The Theta Terminal serves as an intermediary layer, facilitating the connection between our data server and the Python API. Operating as a background process, it establishes a local server on the user's machine, which is then accessed by the Python API. When a request is initiated via the Python API, the process is as follows: the API first relays the request to the Theta Terminal. The Terminal, in turn, forwards the request to the nearest ThetaData **Market Data Distribution Server (MDDS)** and awaits a response. Upon receiving the response, the Terminal processes and sends the data to the user's Python application. The Python API subsequently parses the response into a format that is more accessible to the end user. This approach of using an intermediate application offers numerous advantages, the most notable being the segregation of data processing tasks (such as decompression) from the functionalities specific to the language-based API.

We must first define a callback function designed to handle the streaming trade messages that come from the Theta Terminal. The function checks whether the incoming message is a `TRADE` constant. When a trade message is received, the function prints details of the contract and trade, as well as the last quote at the time of the trade. These details are obtained by calling the `to_string` method on the respective attributes of the `msg` object.

The `stream_all_trades` and `stream_contract` functions initiate streaming connections to receive real-time trade data. They begin by creating a `Thetaclient` instance with specified user credentials. The client then connects to a streaming service using the `connect_stream` method, which uses a callback function for handling incoming data. A request for a full trade stream is made using `req_full_trade_stream_opt` or a single contract with `req_trade_stream_opt`. The response is verified. If the verification fails, it prints a message to the screen.

There's more...

We demonstrated how to stream real-time data for single contracts. Now we'll demonstrate how to combine quote data from multiple contracts to generate real-time quotes straddles and iron condors:

1. Create empty `Quote` objects and set a price:

```
last_call_quote = Quote()
last_put_quote = Quote()
price = 0
```

2. Implement a callback that creates the straddle price out of the bid and ask prices of the contracts that make up the straddle:

```
def callback_straddle(msg):
    if (msg.type != StreamMsgType.QUOTE):
        return
    global price
    if msg.contract.isCall:
        last_call_quote.copy_from(msg.quote)
    else:
        last_put_quote.copy_from(msg.quote)
    straddle_bid = round(last_call_quote.bid_price + last_put_quote.bid_price, 2)
    straddle_ask = round(last_call_quote.ask_price + last_put_quote.ask_price, 2)
    straddle_mid = round((straddle_bid + straddle_ask) / 2, 2)
    time_stamp = thetadata.client.ms_to_time(
        msg.quote.ms_of_day
    )
    if price != straddle_mid:
        print(
            f"time: {time_stamp} bid: {straddle_bid} mid: {straddle_mid} ask: {straddle_ask}"
        )
    price = straddle_mid
```

3. Implement the function that starts the streaming data for each of our defined contracts:

```

def stream_straddle():
    client = ThetaClient(
        username="strimp101@gmail.com",
        passwd="kdk_fzu6pyb0UZA-yuz"
    )
    client.connect_stream(
        callback_straddle
    )
    req_id_call = client.req_quote_stream_opt(
        "SPY", dt.date(2024, 3, 28), 475, OptionRight.CALL
    )
    req_id_put = client.req_quote_stream_opt(
        "SPY", dt.date(2024, 3, 28), 475, OptionRight.PUT
    )
    if (
        client.verify(req_id_call) != StreamResponseType.
        SUBSCRIBED
        or client.verify(req_id_put) != StreamResponseType.
        SUBSCRIBED
    ):
        raise Exception("Unable to stream.")

```

4. Start the streaming data:

```
stream_straddle()
```

After ThetaData connects, you'll start to see the computed bid, mid, and ask prices for the straddle print to the screen for the contracts that you specified:

```

time: 13:34:07.278000 bid: 10.62 mid: 10.66 ask: 10.71
time: 13:34:07.297000 bid: 26.44 mid: 26.55 ask: 26.66
time: 13:34:07.394000 bid: 26.44 mid: 26.62 ask: 26.8
time: 13:34:07.405000 bid: 26.44 mid: 26.55 ask: 26.67
time: 13:34:07.406000 bid: 26.44 mid: 26.62 ask: 26.8
time: 13:34:07.417000 bid: 26.44 mid: 26.55 ask: 26.67
time: 13:34:23.693000 bid: 26.42 mid: 26.54 ask: 26.66

```

Figure 13.3: Streaming straddle prices

Now let's implement streaming data for a short iron condor. In *Chapter 11, Deploy Strategies to a Live Environment*, we learned that a short iron condor is an options strategy that involves selling a lower strike out-of-the-money put, buying an even lower strike out-of-the-money put, selling a higher strike out-of-the-money call, and buying an even higher strike out-of-the-money call with the same expiration date.

5. Define the ticker, expiration date, and strike prices that make up the iron condor:

```

ticker = "SPY"
expiration_date = dt.date(2024, 3, 28)
long_put_strike = 460
short_put_strike = 465
short_call_strike = 480
long_call_strike = 485

```

6. Create empty **Quote** objects and set a price:

```

long_put = Quote()
short_put = Quote()
short_call = Quote()
long_call = Quote()
price = 0

```

7. Implement the callback that captures quote data from each of the four options contracts, computes the bid, mid, and ask prices, and prints the result:

```

def callback_iron_condor(msg):
    if (msg.type != StreamMsgType.QUOTE):
        return
    global price
    if not msg.contract.isCall and msg.contract.strike == long_put_strike:
        long_put.copy_from(msg.quote)
    elif not msg.contract.isCall and msg.contract.strike == short_put_strike:
        short_put.copy_from(msg.quote)
    elif msg.contract.isCall and msg.contract.strike == short_call_strike:
        short_call.copy_from(msg.quote)
    elif msg.contract.isCall and msg.contract.strike == long_call_strike:
        long_call.copy_from(msg.quote)
    condor_bid = round(
        long_put.bid_price
        - short_put.bid_price
        + long_call.bid_price
        - short_call.bid_price,
        2,
    )
    condor_ask = round(
        long_put.ask_price
        - short_put.ask_price
        + long_call.ask_price
        - short_call.ask_price,
        2,
    )
    condor_mid = round((condor_ask + condor_bid) / 2, 2)
    time_stamp = thetadata.client.ms_to_time(
        msg.quote.ms_of_day
    )
    if price != condor_mid:
        print(
            f"time: {time_stamp} bid: {condor_bid} mid: {condor_mid} ask: {condor_ask}"
        )
    price = condor_mid

```

8. Implement the function that starts the streaming data for each of our defined contracts:

```

def streaming_iron_condor():
    client = ThetaClient(
        username="strimp101@gmail.com",
        passwd="YOURPASSWORD"
    )
    client.connect_stream(callback_iron_condor)
    lp_id = client.req_quote_stream_opt(
        ticker, expiration_date, long_put_strike,
        OptionRight.PUT
    )

```

```

        sp_id = client.req_quote_stream_opt(
            ticker, expiration_date, short_put_strike,
            OptionRight.PUT
        )
        client.req_quote_stream_opt(
            ticker, expiration_date, short_call_strike,
            OptionRight.CALL
        )
        client.req_quote_stream_opt(
            ticker, expiration_date, long_call_strike, O
            ptionRight.CALL
        )
    if (
        client.verify(lp_id) != StreamResponseType.SUBSCRIBED
        or client.verify(sp_id) != StreamResponseType.SUBSCRIBED
    ):
        raise Exception("Unable to stream.")

```

9. Start the streaming data:

```
stream_iron_condor()
```

After ThetaData connects, you'll start to see the computed bid, mid, and ask prices for the iron condor print to the screen for the contracts that you specified:

```

time: 13:44:10.242000 bid: -3.84 mid: -3.84 ask: -3.85
time: 13:44:12.132000 bid: -3.84 mid: -3.83 ask: -3.83
time: 13:44:13.729000 bid: -3.84 mid: -3.84 ask: -3.85
time: 13:44:14.988000 bid: -3.85 mid: -3.85 ask: -3.85
time: 13:44:14.988000 bid: -3.84 mid: -3.84 ask: -3.85
time: 13:44:16.798000 bid: -3.84 mid: -3.85 ask: -3.86

```

Figure 13.4: Streaming bid, mid, and ask prices for a short iron condor

IMPORTANT NOTE

The prices for the short iron condor are negative. That's because when we go short an iron condor, we are collecting the premium offered by the combined contracts. For example, if we see a price of -3.85 bid, -3.83 ask, that means if we sell the condor, we will sell at the ask and collect \$3.83.

See also

To learn more about OPRA and options price dissemination, see this URL:

<https://www.opraplan.com/>

Otherwise, check out the ThetaData documentation for more details on how you can use the service at <https://thetadata-api.github.io/thetadata-python/reference/>.

Using the ArcticDB DataFrame database for tick storage

ArcticDB is an embedded, serverless database engine, tailored for integration with pandas and the Python data science ecosystem. It's used for the storage, retrieval, and processing of petabyte-scale data in DataFrame format. It uses common object storage solutions such as S3-compatible storage systems and Azure Blob Storage or local storage. It can efficiently store a 20-year historical record of over 400,000 distinct securities under a single symbol with sub-second retrieval. In ArcticDB, each symbol is treated as an independent entity without data overlap. The engine operates independently of any additional infrastructure, requiring only a functional Python environment and object storage access.

ArcticDB was built by Man Group and has demonstrated its capacity for enterprise-level deployment in some of the world's foremost organizations. The library is slated for integration into Bloomberg's **BQuant** platform, which will empower algorithmic traders to rapidly test, deploy, and share models for alpha generation, risk management, and trading.

Some of the reasons why ArcticDB is a great fit for algorithmic trading include the following:

- ArcticDB offers remarkable processing speed, capable of handling billions of on-disk rows per second. Additionally, its installation process is straightforward and quick with no complex dependencies.
- ArcticDB accommodates both schema-based and schema-less data and is fully equipped to handle streaming data ingestion. It is a bitemporal platform, providing access to all historical versions of the stored data.
- Designed with simplicity in mind, ArcticDB is intuitively accessible to those with experience in Python and pandas, positioning itself as one of the simplest databases in the world.

In this recipe, we'll demonstrate how to use ArcticDB to store streaming trade data from ThetaData.

Getting ready

As of the time of authoring this book, Linux and Windows users can install ArcticDB with `pip`:

```
pip install arcticdb
```

For macOS, use `conda`:

```
conda install -c conda-forge arcticdb
```

For this recipe, we will assume that you have built the streaming options data solution using ThetaData in the *Streaming real-time options data with ThetaData* recipe in this chapter. If you have not, do it now.

How to do it...

We'll set up the ThetaData callback to create a DataFrame for each trade and store it in ArcticDB:

1. Import the libraries that we'll need for the streaming data use case:

```
import time
import pytz
import datetime as dt
import pandas as pd
import arcticdb as adb
import thetadata.client
from thetadata import (
    ThetaClient,
    OptionRight,
    StreamMsg,
    StreamMsgType,
    StreamResponseType,
)
```

2. Create a locally hosted **Lightning Memory-Mapped Database (LMDB)** instance in the current directory and set up an ArcticDB library to store the streaming options data:

```
arctic = adb.Arctic("lmdb://arcticdb_options")
lib = arctic.get_library("trades",
    create_if_missing=True)
```

3. Create a helper function that returns a **datetime** object for the timestamp of the trade:

```
def get_trade_datetime(today, ms_of_day):
    return today + dt.timedelta(
        milliseconds=ms_of_day)
```

4. Create a helper function that computes the number of days to expiration:

```
def get_days_to_expiration(today, expiration):
    return (expiration - today).days
```

5. Implement the callback function that parses the trade message from ThetaData and stores it in ArcticDB:

```
def callback(msg):
    today = dt.datetime.now(
        pytz.timezone("US/Eastern"))
    .replace(
        hour=0,
        minute=0,
        second=0,
        microsecond=0
    )
    if msg.type == StreamMsgType.TRADE:
        trade_datetime = get_trade_datetime(today,
            msg.trade.ms_of_day)
        expiration = pd.to_datetime(
            msg.contract.exp).tz_localize("US/Eastern")
        days_to_expiration = get_days_to_expiration(
            today, expiration)
        symbol = msg.contract.root
        trade = {
            "root": symbol,
            "expiration": expiration,
            "days_to_expiration": days_to_expiration,
            "is_call": msg.contract.isCall,
            "strike": msg.contract.strike,
            "size": msg.trade.size,
            "trade_price": msg.trade.price,
            "exchange": str(
```

```

        msg.trade.exchange.value[1]),
        "bid_size": msg.quote.bid_size,
        "bid_price": msg.quote.bid_price,
        "ask_size": msg.quote.ask_size,
        "ask_price": msg.quote.ask_price,
    }
trade_df = pd.DataFrame(
    trade, index=[trade_datetime])
if symbol in lib.list_symbols():
    lib.update(symbol, trade_df, upsert=True)
else:
    lib.write(symbol, trade_df)

```

6. Implement the function that connects to ThetaData and starts the stream. We'll let it run for 120 seconds before canceling the stream:

```

def stream_all_trades():
    client = ThetaClient(
        username="strimp101@gmail.com",
        passwd="kdk_fzu6pyb0UZA-yuz"
    )
    client.connect_stream(callback)
    req_id = client.req_full_trade_stream_opt()
    response = client.verify(req_id)
    if (client.verify(req_id) != StreamResponseType.SUBSCRIBED):
        raise Exception("Unable to stream.")
    time.sleep(120)
    print("Cancelling stream...")
    client.remove_full_trade_stream_opt()

```

7. Start the stream:

```
stream_all_trades()
```

8. After the stream has been canceled, we can interact with the data stored in ArcticDB. First, defragment the data stored in LMDB if required:

```

for symbol in lib.list_symbols():
    if lib.is_symbol_fragmented(symbol):
        lib.defragment_symbol_data(symbol)

```

9. List the number of symbols that we have acquired trade data for:

```
len(lib.list_symbols())
```

10. Get a DataFrame containing the trade data for a single symbol:

```
qqq = lib.read("QQQ").data
```

The result is a pandas DataFrame with one row for every trade:

root	expiration	days_to_expiration	is_call	strike	size	trade_price	exchange	bid_size	bid_price	ask_size	ask_price	
2023-12-27 11:10:37.488000-05:00	QQQ	2023-12-27 05:00:00	0	False	408.78	1	0.20	XPHL	25	0.19	1219	0.20
2023-12-27 11:10:37.493000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
2023-12-27 11:10:37.523000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
2023-12-27 11:10:37.544000-05:00	QQQ	2023-12-28 05:00:00	1	True	410.78	1	1.23	XASE	533	1.21	58	1.23
2023-12-27 11:10:37.558000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
...
2023-12-27 11:12:36.129000-05:00	QQQ	2023-12-29 05:00:00	2	True	412.78	1	1.00	XCBO	133	1.00	344	1.01
2023-12-27 11:12:36.366000-05:00	QQQ	2023-12-27 05:00:00	0	True	409.78	5	1.12	XASE	12	1.12	292	1.14
2023-12-27 11:12:36.781000-05:00	QQQ	2023-12-28 05:00:00	1	False	410.78	3	1.25	EDGX	33	1.24	406	1.26
2023-12-27 11:12:37.126000-05:00	QQQ	2023-12-28 05:00:00	1	True	413.78	2	0.24	XPHL	1009	0.24	548	0.25
2023-12-27 11:12:37.394000-05:00	QQQ	2023-12-29 05:00:00	2	False	389.78	1	0.04	EDGX	6	0.04	8189	0.05

Figure 13.5: A pandas DataFrame with trade data for ETF QQQ

11. We can use the powerful QueryBuilder tool to process the data before acquiring it, which speeds up retrieval. Find all trades where the spread is less than \$0.05:

```
q = adb.QueryBuilder()
filter = (q.ask_price - q.bid_price) < 0.05
q = q[filter]
data = lib.read("QQQ", query_builder=q).data
```

The result is a pandas DataFrame with all trades that were executed when the spread was less than \$0.05.

root	expiration	days_to_expiration	is_call	strike	size	trade_price	exchange	bid_size	bid_price	ask_size	ask_price	
2023-12-27 11:10:37.488000-05:00	QQQ	2023-12-27 05:00:00	0	False	408.78	1	0.20	XPHL	25	0.19	1219	0.20
2023-12-27 11:10:37.493000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
2023-12-27 11:10:37.523000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
2023-12-27 11:10:37.544000-05:00	QQQ	2023-12-28 05:00:00	1	True	410.78	1	1.23	XASE	533	1.21	58	1.23
2023-12-27 11:10:37.558000-05:00	QQQ	2023-12-27 05:00:00	0	True	412.78	43	0.05	XISX	18	0.05	1570	0.06
...
2023-12-27 11:12:36.129000-05:00	QQQ	2023-12-29 05:00:00	2	True	412.78	1	1.00	XCBO	133	1.00	344	1.01
2023-12-27 11:12:36.366000-05:00	QQQ	2023-12-27 05:00:00	0	True	409.78	5	1.12	XASE	12	1.12	292	1.14
2023-12-27 11:12:36.781000-05:00	QQQ	2023-12-28 05:00:00	1	False	410.78	3	1.25	EDGX	33	1.24	406	1.26
2023-12-27 11:12:37.126000-05:00	QQQ	2023-12-28 05:00:00	1	True	413.78	2	0.24	XPHL	1009	0.24	548	0.25
2023-12-27 11:12:37.394000-05:00	QQQ	2023-12-29 05:00:00	2	False	389.78	1	0.04	EDGX	6	0.04	8189	0.05

Figure 13.6: A pandas DataFrame with filtered trade data for ETF QQQ

12. Use QueryBuilder and native pandas methods to efficiently retrieve and post-process data:

```
q = adb.QueryBuilder()
filter = (q.days_to_expiration > 1)
q = (
    q[filter]
    .groupby("expiration")
    .agg({"bid_size": "sum", "ask_size": "sum"})
)
data = lib.read("QQQ", query_builder=q).data.sort_index()
```

The result is a pandas DataFrame with a snapshot of the aggregate bid and ask sizes at each expiration at the time of trade for all contracts with greater than 1 day to expiration.

	bid_size	ask_size
expiration		
2023-12-29 00:00:00-05:00	30300	63329
2024-01-02 00:00:00-05:00	380	733
2024-01-03 00:00:00-05:00	2283	8145
2024-01-04 00:00:00-05:00	21	686
2024-01-05 00:00:00-05:00	1566	29040

Figure 13.7: A pandas DataFrame with filtered, grouped, and aggregated bid and ask volumes across a range of strike prices

How it works...

We started by creating an instance of the `Arctic` class, which is initialized with the `lmdb://arcticdb_options` connection string indicating that the ArcticDB database will use LMDB as its storage engine. Then we call the `get_library` method on the `Arctic` instance. This method attempts to retrieve a library named `trades` from the underlying LMDB database. If the library does not exist, the `create_if_missing=True` argument ensures that it is created. The `get_library` method connects to the underlying storage engine to manage the data associated with the `trades` library.

NOTE

LMDB is a high-performance embedded key-value store designed to provide efficient, transactional data storage mechanisms for applications. Built on a memory-mapped file, LMDB allows for data to be quickly read and written with minimal overhead, supporting a substantial number of simultaneous read operations along with a single write transaction. Its architecture is optimized for speed, concurrency, and reliability, making it a suitable choice for applications requiring fast access to large datasets with a small footprint. LMDB operates with a no-overhead approach and can handle databases much larger than RAM, leveraging the operating system's virtual memory to manage data transactions effectively.

The `get_trade_datetime` function calculates and returns the datetime of a trade by adding a specific number of milliseconds to a given date. ThetaData uses the number of milliseconds since the beginning of the trade day as its timestamp, so this function gives us a more human-readable version. The `get_days_to_expiration` function first calculates a `Timedelta` object between the expiration date and today's date and returns the number of days between the two.

Storing data

The `callback` function processes incoming trade messages by first setting a `today` datetime object to the current date with the time set to the start of the day in the U.S. Eastern time zone. If the message

type is a trade, it computes the trade datetime by adding the number of milliseconds since midnight. The function then calculates the days remaining until the contract's expiration. From there, we construct a dictionary with trade details. These details include information about the contract and the trade itself such as the symbol, expiration, days to expiration, option type, strike price, size, and the current bid and ask sizes and prices. This trade information is then formatted into a pandas DataFrame with the trade datetime as its index. Depending on whether the symbol already exists in the DataFrame, the trade data is either appended to the existing symbol data or written as new data under the symbol. The `stream_all_trades` function is similar to the one we used in the previous recipe except that we cancel the stream after 120 seconds.

Retrieving data

We demonstrate how to use the `QueryBuilder` object to retrieve data. The first line, `q = adb.QueryBuilder()`, creates an instance of the `QueryBuilder` class, which is used for constructing database queries that are passed to the ArcticDB processing engine. The second line, `filter = (q.ask_price - q.bid_price) < 0.05`, defines a filter condition whereby only records with a difference between `ask_price` and `bid_price` of less than `0.05` are returned. This filter is applied in the third line, `q = q[filter]`, which modifies the `QueryBuilder` instance to include this filtering criterion. Finally, `data = lib.read("QQQ", query_builder=q).data` uses the `read` method of the `lib` object to execute the query against the `QQQ` dataset, retrieving records that match the filter. The `data` attribute accesses the actual data from the query result.

In the second example, we construct a `QueryBuilder` object to select records where `days_to_expiration` is greater than `1`. The query is then refined to group the results by the `expiration` field and aggregate the `bid_size` and `ask_size` fields by summing them up. Finally, the `read` method is used to execute this query against the `QQQ` dataset in the LMDB datastore. The resulting data is sorted by index.

There's more...

We demonstrated how to store data locally using LMDB. We can achieve a similar performance using remote object storage backends. At the time of writing, ArcticDB has tested the following S3 backends:

- AWS S3
- Ceph
- MinIO on Linux
- Pure Storage S3

- Scality S3
- VAST Data S3

To use ArcticDB with S3, you simply need an S3 bucket with PUT permissions. Then you can instantiate the Arctic class with its path. Use this example if your AWS `credentials` file is stored locally:

```
Arctic("s3://MY_ENDPOINT:MY_BUCKET?aws_auth=true")
```

Otherwise, you can specify the credentials in the connection string:

```
Arctic("s3://MY_ENDPOINT:MY_BUCKET?region=YOUR_REGION&access=ABCD&secret=DCBA")
```

See also

For more information on ArcticDB, see the following resources:

- The ArcticDB homepage with documentation, community information, and blog: <https://arcticdb.io/>
- Detailed documentation with walkthroughs: <https://docs.arcticdb.io/latest/>
- The ArcticDB GitHub page: <https://github.com/man-group/ArcticDB>
- LMDB documentation and homepage: <http://www.lmdb.tech/doc/>

Triggering real-time risk limit alerts

Once a strategy is in place, algorithmic trading revolves around managing performance and risk metrics, emphasizing monitoring exceptions or deviations that exceed predefined thresholds. Risk metrics are critical indicators that flag potential issues or opportunities in the trading strategy.

Professional algorithmic traders rely heavily on real-time alert systems. These systems detect and notify traders when specific risk metrics reach or surpass set thresholds. This notification enables traders to respond to market changes, adjust strategies, or mitigate risks. These alerts can be based on various risk metrics, such as **Conditional Values at Risk (CVaRs)**, drawdowns, or unusual trading volumes. Effectively managing these alerts and understanding the underlying causes is a cornerstone of successful algorithmic trading.

In this recipe, you'll set up real-time alerts to monitor our portfolio's intraday CVaR. In [Chapter 12, Deploy Strategies to a Live Environment](#), we learned that CVaR measures the expected losses that occur beyond the **Value at Risk (VaR)** in the tail end of a distribution of possible returns. Our focus will be on setting up the infrastructure for alerting. The decision of how to send alerts via email, SMS, Slack, Telegram bot, or other channels is left up to the reader.

Getting ready

For this recipe, we assume that you have built the real-time risk and performance metrics detailed in [Chapter 12, Deploy Strategies to a Live Environment](#). We also assume that you've followed the recipes in [Chapter 12](#). If you have not, do it now.

How to do it...

We'll edit the `app.py` file to include a new thread to check whether the portfolio CVaR exceeds a defined threshold. Since we've already set up the app's scaffolding, adding a new method to a background thread requires only a few lines of code:

1. Add a `watch_cvar` method at the end of the `IBApp` class in `app.py`:

```
def watch_cvar(self, threshold, interval):  
    print("Watching CVaR in 60 seconds...")  
    time.sleep(60)  
    while True:  
        cvar = self.cvar[1]  
        if cvar < threshold:  
            print(f"Portfolio CVaR ({  
                cvar}) crossed threshold ({  
                threshold})")  
            pass  
        time.sleep(interval)
```

2. Modify the `__init__` method to resemble the following:

```
def __init__(self, ip, port, client_id, account,  
            interval=5, **kwargs):  
    IBWrapper.__init__(self)  
    IBClient.__init__(self, wrapper=self)  
    self.account = account  
    self.create_table()  
    self.connect(ip, port, client_id)  
    threading.Thread(target=self.run,  
                      daemon=True).start()  
    time.sleep(5)  
    threading.Thread(  
        target=self.get_streaming_returns,  
        args=(99, interval, "unrealized_pnl"),  
        daemon=True  
    ).start()  
    time.sleep(5)  
    threading.Thread(  
        target=self.watch_cvar,  
        args=(kwargs["cvar_threshold"], interval),  
        daemon=True  
    ).start()
```

3. To run the app, modify the line under `if __name__ == "__main__":` to resemble the following:

```
app = IBApp(  
    "127.0.0.1",  
    7497,
```

```
    client_id=12,
    account="DU7129120",
    interval=10,
    cvar_threshold=-500
)
```

How it works...

The method monitors the CVaR of our portfolio against a specified threshold at regular intervals. Initially, it prints a message and pauses for 60 seconds. Then, in an infinite loop, it checks whether the dollar CVaR falls below the given threshold. If it does, `watch_cvar` prints a message indicating that the CVaR has crossed this threshold. After each check, the method pauses for a duration specified by the `interval` parameter before repeating the process. In this method, we can implement our actual alerting code.

To start the process of monitoring the intraday CVaR, we follow the same pattern that we introduced in previous chapters. We create and start a new thread that executes the `watch_cvar` method. The thread runs independently, passing the specified `cvar_threshold` and `interval` values from `kwargs` as arguments to the `watch_cvar` method. Running the method in a thread allows us to execute other code with the monitoring in the background.

There's more...

Depending on your use case, you may choose to send alerts via email or SMS. Short snippets are included to get you started with each. Depending on your email provider, you may have to take steps to enable sending emails through code. For example, Gmail stopped supporting logins through “less secure” means in 2022 and now requires the OAuth2 authorization framework. You can learn more at <https://developers.google.com/gmail/api/quickstart/python>.

Sending emails using Python

Python has two built-in libraries that make it easy to send simple emails. We'll demonstrate a simple example of how to send emails with Python that you can use for your alerts:

1. Include the following imports at the top of `app.py`:

```
<snip>
import smtplib
from email.message import EmailMessage
s = smtplib.SMTP("localhost")
```

2. Update the `watch_cvar` method to include the following code to construct and send an email:

```
def watch_cvar(self, threshold, interval):
```

```

print("Watching CVaR in 60 seconds...")
time.sleep(60)
while True:
    cvar = self.cvar[1]
    if cvar < threshold:
        print(f"Portfolio CVaR ({cvar}) crossed threshold ({threshold})")
        msg = EmailMessage()
        msg.set_content(
            """
            Risk alert:
            f"Portfolio CVaR ({cvar}) crossed threshold ({threshold})"
            """
        )
        msg["Subject"] = "CVaR threshold crossed"
        msg["From"] = "sender@email.com"
        msg["To"] = "you@email.com"
        s.send_message(msg)
        s.quit()
    time.sleep(interval)

```

First, we create an `EmailMessage` object and connect it to an email server. In a production application, you would replace `localhost` with the server details of your email provider. You would also need to include the provider's authentication code. Within the `watch_cvar` method, we must check whether CVaR exceeds our threshold and construct our email. The email's subject, sender, and recipient are then specified. Next, we send the prepared email message and close the connection.

IMPORTANT NOTE

There are several paid services that offer bulk email sending for very little cost. Mailgun is one that offers thousands of emails per month for a reasonable cost. If you don't expect to send hundreds or thousands of emails per month, consider using your existing email provider. Depending on your provider, instructions for authentication and sending emails will differ.

Sending alerts via SMS

There are many third-party services that we can use to send SMS via Python. We'll look at the most popular one, **Twilio**. Twilio's APIs provide seamless interaction with users through voice, SMS, video, or messaging. Thousands of businesses use Twilio for sending SMS notifications such as password changes or informational alerts. Twilio operates on a pay-as-you-use pricing model.

To start, install the Twilio library with `pip`:

```
pip install twilio
```

We'll demonstrate a simple example of how to send SMS with Python using the Twilio API that you can use for your alerts.

1. Include the following imports at the top of `app.py`:

```
<snip>
from twilio.rest import Client
account_sid = TWILIO_ACCOUNT_SID
auth_token = TWILIO_AUTH_TOKEN
client = Client(account_sid, auth_token)
```

2. Update the `watch_cvar` method to include the following code to construct and send an SMS:

```
def watch_cvar(self, threshold, interval):
    print("Watching CVaR in 60 seconds...")
    time.sleep(60)
    while True:
        cvar = self.cvar[1]
        if cvar < threshold:
            print(f"Portfolio CVaR ({cvar}) crossed threshold ({threshold})")
            body = """
Risk alert:
f"Portfolio CVaR ({cvar}) crossed threshold ({threshold})"
"""
            message = client.messages.create(
                body, from_=FROM_NUMBER, to=TO_NUMBER
            )
        time.sleep(interval)
```

Replace `TWILIO_ACCOUNT_SID` and `TWILIO_AUTH_TOKEN` with the account ID and auth token you can retrieve from the Twilio console. Then replace `FROM_NUMBER` with the number you purchased and `TO_NUMBER` with your mobile number.

See also

To learn more about sending emails and SMS alerts, see the following resources:

- Mailgun homepage : <https://www.mailgun.com/>
- Twilio homepage: <https://www.twilio.com/en-us>
- Amazon Simple Notification Service (SNS): <https://aws.amazon.com/sns/>

Storing trade execution details in a SQL database

Capturing trade data is essential in algorithmic trading. Recording order execution data provides the ability to monitor the time interval between the placement and fulfillment of orders. This aspect is particularly important in volatile markets where prices are moving quickly, since it allows us to evaluate and enhance our execution strategies for better efficiency. Another benefit is measuring slippage – the difference between the expected and actual execution prices. Analyzing slippage is important for understanding the impact of execution on trading costs and profitability. Since

transaction costs, both broker fees and slippage, erode returns, minimizing costs can have a real impact on returns. Tracking open orders can help us assess current market exposure but also ensures adherence to risk management processes and helps in balancing our portfolio.

As we've learned, the IB API facilitates this process through its EWrapper callback methods. These methods are called in response to events such as order status changes, order executions, and open orders. The callback methods offer a streamlined and event-driven approach for capturing order-related data. In this recipe, we'll use the three order callback methods we established in [Chapter 10, Set up the Interactive Brokers Python API](#), to capture order details.

Getting ready

For this recipe, we assume that you have set up the code to establish the SQLite database in [Chapter 10, Set up the Interactive Brokers Python API](#), and the methods to acquire portfolio data in [Chapter 11, Manage Orders, Positions, and Portfolios with the IB API](#). If you have not, do it now.

We'll do some cleanup to make our code more organized. In the `IBApp` class, find the `create_table` method. In the method, there's a long line of SQL we use to create the table to store bid and ask data. We'll move that to the `utils.py` file.

1. Cut the SQL from the `create_table` method and paste it into `utils.py` similar to the following:

```
CREATE_BID_ASK_DATA = """
CREATE TABLE IF NOT EXISTS bid_ask_data
(
    timestamp DATETIME,
    symbol STRING,
    bid_price REAL,
    ask_price REAL,
    bid_size INTEGER,
    ask_size INTEGER
) """
```

2. Now in `app.py`, add the following import under the existing imports:

```
from utils import CREATE_BID_ASK_DATA
```

3. Modify `create_table` to resemble the following:

```
def create_table(self):
    cursor = self.connection.cursor()
    cursor.execute(CREATE_BID_ASK_DATA)
```

4. Update the name of the SQLite file being created to the name of the strategy since now we're storing more than just bid and ask data:

```
@property
def connection(self):
    return sqlite3.connect("strategy_1.sqlite",
        isolation_level=None)
```

How to do it...

We'll add three new SQL statements to create tables to store order status, open order details, and executed trades.

1. In **utils.py**, add the following statement under the previous declaration to create an SQLite table to store open order details:

```
CREATE_OPEN_ORDERS = """
CREATE TABLE IF NOT EXISTS open_orders
(
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    order_id INTEGER,
    symbol STRING,
    sec_type STRING,
    exchange STRING,
    action STRING,
    order_type STRING,
    quantity INTEGER,
    status STRING
)"""
```

2. In **utils.py**, add the following statement under the previous declaration to create an SQLite table to store order execution details:

```
CREATE_TRADES = """
CREATE TABLE IF NOT EXISTS trades
(
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    request_id INTEGER,
    order_id INTEGER,
    execution_id INTEGER,
    symbol STRING,
    sec_type STRING,
    currency STRING,
    quantity INTEGER,
    last_liquidity REAL
)"""
```

3. In **app.py**, update the import for the bid ask data to include all imports:

```
from utils import (
    CREATE_BID_ASK_DATA,
    CREATE_OPEN_ORDERS,
    CREATE_TRADES
)
```

4. Modify **create_table** in the **IBApp** class in **app.py** to resemble the following:

```
def create_table(self):
    cursor = self.connection.cursor()
    cursor.execute(CREATE_BID_ASK_DATA)
    cursor.execute(CREATE_OPEN_ORDERS)
    cursor.execute(CREATE_TRADES)
```

5. In **wrapper.py**, update the **openOrder** method to generate and execute a SQL insert statement when the callback is triggered.

Replace the code that simply prints a message with the following:

```
def openOrder(self, order_id, contract, order, order_state):
    cursor = self.connection.cursor()
```

```

query = "INSERT INTO open_orders (
    order_id, symbol, sec_type, exchange, action,
    order_type, quantity, status)
VALUES (?, ?, ?, ?, ?, ?, ?, ?)"
values = (
    order_id,
    contract.symbol,
    contract.secType,
    contract.exchange,
    order.action,
    order.orderType,
    order.totalQuantity,
    order_state.status,
)
cursor.execute(query, values)

```

When submitting an order, we will end up with an entry in the SQLite `open_orders` table.

1	2023-12-27 15:30:45	3	PSX	STK	SMART	BUY	MKT	10 Submitted
2	2023-12-27 15:30:46	3	PSX	STK	SMART	BUY	MKT	10 Filled
3	2023-12-27 15:30:46	3	PSX	STK	SMART	BUY	MKT	10 Filled

Figure 13.8: Entry in the `open_orders` table executed from the `openOrder` callback

6. In `wrapper.py`, update the `execDetails` method to generate and execute a SQL insert statement when the callback is triggered. Replace the code that simply prints a message with the following:

```

def execDetails(self, request_id, contract,
                execution):
    cursor = self.connection.cursor()
    query = "INSERT INTO trades (request_id,
        symbol, sec_type, currency, execution_id,
        order_id, quantity, last_liquidity)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?)"
    values = (
        request_id,
        contract.symbol,
        contract.secType,
        contract.currency,
        execution.execId,
        execution.orderId,
        execution.shares,
        execution.lastliquidity,
    )
    cursor.execute(query, values)

```

When an order is executed, we will end up with an entry in the SQLite `trades` table.

	timestamp	:	request_id	:	order_id	:	execution_id	:	symbol	:	sec_type	:	currency	:	quantity	:	last_liquidity	:
1	2023-12-27 15:30:46		-1		3	00025b49.658bd468.01.01	PSX		STK		USD		10		2			

Figure 13.9: Entry in the `trades` table executed from the `execDetails` callback

How it works...

We extend the `create_table` method to automatically create additional tables to store order or trade status and execution data. These tables will be created when the `IBApp` class is instantiated. For the

`openOrder` and `execDetail` methods, SQL insert statements are created when they are triggered. The methods specify the columns to be populated and the corresponding values. These values are then executed against the query to insert the data into the database. The question mark in the SQL statement serves as a placeholder for parameters that will be substituted with actual values when the query is executed.

There's more...

We demonstrated a small sample of the fields that can be captured in `execDetails`. Both callback methods receive objects that contain many properties.

Contract

The `contract` object has the following attributes that can be stored in your tables:

- `conId`: A unique identifier
- `symbol`: An asset symbol
- `secType`: Type (stock, option, etc.)
- `lastTradeDateOrContractMonth`: The last trading day or month
- `strike`: An option's strike price
- `right`: Used with a call or put option
- `multiplier`: An options or futures multiplier
- `exchange`: A contract's exchange
- `currency`: An asset's currency
- `localSymbol`: A local exchange symbol
- `primaryExch`: The primary exchange
- `tradingClass`: The contract trading class
- `includeExpired`: Includes expired futures
- `secIdType`: The identifier type (ISIN, CUSIP)
- `secId`: A security identifier
- `description`: The contract description
- `issuerId`: An issuer identifier
- `comboLegsDescription`: Combo legs details
- `comboLegs`: Combined contract legs
- `deltaNeutralContract`: Delta neutral details

Order

The `order` object has 66 properties available. Here we capture some of the more valuable ones that can be stored in your tables:

- `orderId`: A client's order ID
- `clientId`: Placing a client ID
- `permId`: The host order identifier
- `action`: The side (`BUY`, `SELL`, etc.)
- `totalQuantity`: The positions number
- `orderType`: The order's type
- `lmtPrice`: `LIMIT` order price
- `auxPrice`: Stop the price for `STP LMT`
- `tif`: Time in force (`DAY`, `GTC`, etc.)

Order state

The `order_state` object has the following attributes that can be stored in your tables:

- `status`: An order's current status
- `initMarginBefore`: The initial margin before
- `maintMarginBefore`: The maintenance margin before
- `equityWithLoanBefore`: Account equity with loan before
- `initMarginChange`: The initial margin change
- `maintMarginChange`: The maintenance margin change
- `equityWithLoanChange`: The equity with loan change
- `initMarginAfter`: The initial margin after an order
- `maintMarginAfter`: The maintenance margin after an order
- `equityWithLoanAfter`: The equity with loan after an order
- `commission`: The generated commission
- `minCommission`: The minimum commission
- `maxCommission`: The maximum commission
- `commissionCurrency`: The commission currency
- `warningText`: An order warning message
- `completedTime`: The order completion time
- `completedStatus`: The order completion status

Executions

The execution object has the following attributes that can be stored in your tables:

- **orderId**: The client's order ID
- **clientId**: The placing client ID
- **execId**: The execution identifier
- **time**: The execution server time
- **acctNumber**: The allocated account number
- **exchange**: The execution exchange
- **side**: The transaction side (**BOT**, **SLD**)
- **shares**: Number of shares filled
- **price**: The execution price
- **permId**: The TWS order identifier
- **liquidation**: The IB liquidation indicator
- **cumQty**: The cumulative quantity
- **avgPrice**: The average price
- **orderRef**: A user-customizable string
- **evRule**: The **Economic Value (EV)** rule
- **evMultiplier**: The EV price change unit
- **modelCode**: The model code
- **lastLiquidity**: The execution liquidity type

See also

For more details on the various IB API callbacks and objects used in this recipe, see the following resources:

- Documentation on the **openOrder** EWrapper method: https://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html#aa05258f1d005accd3efc0d60bc151407
- Documentation on the **execDetails** EWrapper method: https://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html#a09f82de3d0666d13b00b5168e8b9313d
- Documentation on the **Contract** object: https://interactivebrokers.github.io/tws-api/classIBApi_1_1Contract.html
- Documentation on the **Order** object: https://interactivebrokers.github.io/tws-api/classIBApi_1_1Order.html
- Documentation on the **OrderState** object: https://interactivebrokers.github.io/tws-api/classIBApi_1_1OrderState.html
- Documentation on the **Execution** object: https://interactivebrokers.github.io/tws-api/classIBApi_1_1Execution.html

Index

As this ebook edition doesn't have fixed pagination, the page numbers below are hyperlinked for reference only, based on the printed edition of this book.

A

aggregates

 used, for creating columns [38-40](#)

alerts

 sending, via SMS [367](#), [368](#)

algorithmic trading app

 building [254-257](#)

 class [259](#)

 inheritance and overriding [258](#)

 request-callback pattern [258](#)

 testing [259](#), [260](#)

alpha factors [92](#)

AlphaLens

 backtest results, preparing [200-205](#)

 factor return performance, examining [210-216](#)

 factor turnover, evaluating [216-221](#)

 IC, evaluating [205-210](#)

 apply function [75](#)

ArcticDB

 reference link [363](#)

ArcticDB DataFrame database

 using, for storage [356-361](#)

asfreq method

reference link [69](#)

asset returns

calculating, with pandas [53-57](#)

autocommit mode [289](#)

Average True Range (ATR) [169](#)

B

backtest results

preparing [200-205](#)

bag [335](#)

basic lower band [169](#)

basic upper band [169](#)

basis trade [15](#)

before_trading_start function [185](#)

context parameter [185](#)

data parameter [186](#)

Booleans

used, for creating columns [38-40](#)

BQuant platform [356](#)

C

callback function [362](#)

data, retrieving [362, 363](#)

data, storing [362](#)

Calmar ratio [234](#)

Chicago Board Options Exchange (CBOE) [15](#)

Chicago Mercantile Exchange (CME) [5](#)

combo_leg function [335](#)
combo_leg method [334](#)
Comma-Separated Values (CSV) [106](#)
compound annual growth rate (CAGR) [234](#)
compound returns [53](#)
Conditional Value at Risk (CVaR) [320](#)
continuous futures data
exploring, with Nasdaq Data Link [3, 4-7](#)
contract object [373](#)
 attributes [373](#)
 creating, with IB API [261-264](#)
 reference link [376](#)
create_table method [372](#)
CSV format
 data, storing on disk in [106-108](#)
cumulative returns [317](#)
cumulative return series
 cumulative sum of compound returns [62, 63](#)
 cumulative sum of simple returns [61](#)
 generating [60, 61](#)
 working [63, 64](#)
curve method, OpenBB SDK
 reference link [19](#)
custom functions
 applying, for analysis of time series data [73-76](#)
Cyclically Adjusted PE ratio (CAPE ratio) [9](#)

D

data

 storing, in PostgreSQL database server [114-118](#)

 storing, in ultra-fast HDF5 format [120-123](#)

 storing, on disk with SQLite [109](#)

dataclass [282](#)

DataFrames

 at, using for fast access to scalar [52](#)

 building [32](#), [33](#)

 columns, creating with aggregates [38-40](#)

 columns, creating with Booleans [38-40](#)

 columns, creating with strings [38-40](#)

 concatenating [40](#)

 custom functions, applying [46](#)

 data, examining [48](#), [49](#), [52](#)

 data, grouping [47](#)

 data, grouping on key or index [41](#)

 data, selecting from [48](#), [49](#), [52](#)

 data, selection by Boolean indexing [51](#)

 data, selection by label with loc [50](#)

 data, selection by position with loc [50](#)

 data, transforming [47](#)

 grouping, by multiple columns [44](#), [45](#)

 manipulating [37](#)

 methods, applying to columns [45](#), [46](#)

 nsmallest and nlargest, using [52](#)

 options data, joining together [42-44](#)

 partial string indexing [52](#)

pivoting, as Excel [41](#)
query method, using [53](#)
transforming [38](#)
data granularity [70](#)
data, on disk
 storing, in CSV format [106-108](#)
 storing, with SQLite [109-112](#)
data resampling
 for different time frames [64-68](#)
data visualization
 pandas, using [78-83](#)
DatetimeIndex [29](#)
 reference link [31](#)
delta [21](#)
drawdown analysis
 building [236-241](#)

E

Economic Value (EV) [375](#)
emails
 sending, with Python [366, 367](#)
empyrical-reloaded [313](#)
exchange-traded fund (ETF) [133](#)
execDetails EWrapper method
 reference link [375](#)
exec_trades function [186](#)
execution object [375](#)
 attributes [375](#)

reference link [376](#)

Expected Shortfall (ES) [320](#)

exposure

analyzing [242](#)

F

factor rank autocorrelation, in factor portfolio

impacting, reasons [219](#)

factor ranking model

for Apple Silicon users [150](#)

for Windows, Unix/Linux, and Mac Intel users [150](#)

preparing, with Zipline Pipelines [150-153](#)

factor return performance

examining [210-216](#)

factors [199](#)

factor turnover

evaluating [216-221](#)

Fama-French factors

portfolio sensitivities, analyzing to [137-143](#)

fillna method

reference link [73](#)

Finviz stock screener [13](#)

reference link [14](#)

G

get_trade_datetime function [362](#)

Global Interpreter Lock (GIL) [258](#)

gross leverage [234](#)

H

Heating Oil (HO) [338](#)

Hierarchical Data Format (HDF) [120](#)

hierarchical index [30](#)

High Minus Low (HML) [22](#), [141](#)

historical market data

fetching [267-271](#)

historic data, receiving [273-275](#)

historic data, requesting [271-273](#)

historic futures data

fetching, with OpenBB SDK [15-19](#)

Homebrew

reference link [170](#)

I

IB API [293](#)

contract object, creating with [261-263](#)

Order object, creating with [264-266](#)

orders, executing with [294-298](#)

portfolio details, obtaining [302-306](#)

portfolio profit and loss, computing [309](#), [311](#), [312](#)

position-level details, inspecting [306-308](#)

iloc method [51](#)

information coefficients (ICs)

evaluating [205-210](#)

initialize function [186](#)

Interactive Brokers (IB) [253](#)

Interactive Development Environment (IDE) [275](#)

interactive PCA analytics dashboard

creating, with Plotly Dash [97-104](#)

interpolate method

reference link [73](#)

intraday multi-asset mean reversion strategy

deploying [338-345](#)

J

Just-In-Time (JIT) compiler [156](#)

K

Kurtosis [234](#)

L

latent return drivers

identifying, with PCA [126-131](#)

leverage

analyzing [241-247](#)

Lightning Memory-Mapped Database (LMDB) [358](#)

Light Sweet Crude Oil (CL) [338](#)

linear regression

used, for finding and hedging portfolio beta [132](#)

Linux

executable shell file, creating [113, 114](#)

live market data

streaming [278-282](#)

streaming data, receiving [283, 285](#)

streaming data, requesting [282](#)

live tick data

storing, in local SQL database [285-291](#)

local SQL database

 storing, in local SQL database [285-291](#)

loc method [51](#)

M

Mac

 executable shell file, creating [113, 114](#)

Mailgun [367](#)

 reference link [368](#)

make_pipeline function [195](#)

Man Group [347](#)

Marginal Contribution to Active Risk (MCAR) [141](#)

market closures [69](#)

Market Data Distribution Server (MDDS) [351](#)

market data snapshot

 obtaining [276, 277](#)

market inefficiency, based on volatility

 assessing [143-149](#)

Matplotlib

 used, for animating evolution of yield curve [84-87](#)

 used, for plotting options implied volatility surfaces [88-91](#)

max drawdown [318](#)

mean reversion strategy

 exploring, with Zipline Reloaded [189-197](#)

meme stock hysteria [10](#)

missing data issues

 addressing [69-73](#)

momentum factor strategy

backtesting, with Zipline Reloaded [180-188](#)

monthly factor portfolio strategy

deploying [326-331](#)

MultiIndex [30](#)

reference link [31](#)

MultiIndex DataFrame [34](#)

building, from scratch [34](#)

MultiIndex object

existing DataFrame, reindexing with [35](#), [36](#)

N

nan value [31](#)

Nasdaq Data Link

continuous futures data, exploring with [3-7](#)

reference link [7](#)

S&P 500 ratios data, exploring with [8](#), [9](#)

NYMEX RBOB Gasoline Index (RB) [338](#)

O

Object Relational Mapping (ORM) [118](#)

ohlc aggregate [42](#)

omega_ratio method [234](#), [319](#)

OpenBB SDK

historical options data [21](#)

Options Greeks [21](#)

options market data, navigating with [19](#), [20](#)

reference link [12](#)

stock screeners, building [13](#), [14](#)

used, for fetching historic futures data [14-19](#)

used, for working with stock market data [10](#), [11](#)

openOrder EWrapper method

reference link [375](#)

options chain [19](#)

options combo strategy

deploying [332-337](#)

options implied volatility surfaces

plotting, with Matplotlib [88-91](#)

options market data

navigating, with OpenBB SDK [19](#), [20](#)

Options Price Reporting Authority (OPRA) [348](#)

reference link [356](#)

order object [373](#)

attributes [373](#)

creating, with IB API [264-266](#)

reference link [376](#)

orders

executing, with IB API [294-298](#)

managing [299-302](#)

orders, based on portfolio targets

odering, for order_value method [324](#)

order_percent method [324](#)

order_target_percent method [325](#)

order_target_quantity method [324](#)

order_target_value method [325](#)

sending [321-324](#)

order_state object [374](#)

 attributes [374](#)

 reference link [376](#)

P

pandas

 asset returns, calculating with [53-57](#)

 used, for data visualization [78-83](#)

pandas cumprod method

 reference link [64](#)

pandas cumsum method

 reference link [64](#)

pandas DataFrames

 reference link [37, 48](#)

pandas_datareader

 factor data, harnessing with [23-25](#)

pandas DatetimeIndexes

 reference link [31](#)

pandas indexes

 reference link [31](#)

pandas index types

 DatetimeIndex [29](#)

 exploring [28](#)

 MultiIndex [30](#)

 PeriodIndex [30](#)

 reference link [31](#)

 working [29](#)

pandas MultiIndexes

reference link [31](#)

pandas resample method

reference link [69](#)

pandas Series

building [31, 33](#)

reference link [37](#)

pandas to_csv method [108](#)

pandas unique method [42](#)

PeriodIndex [30](#)

pgAdmin

user interface [119](#)

pipeline_output function [186](#)

Plotly Dash

used, for creating interactive PCA analytics dashboard [97-104](#)

portfolio

conditional value at risk [320](#)

cumulative returns [317](#)

max drawdown [318](#)

Omega ratio [319](#)

sharpe_ratio method [319](#)

volatility [318](#)

portfolio beta

finding, with linear regression [132-137](#)

hedging, with linear regression [132-137](#)

portfolio, IB API

details, obtaining [302-306](#)

portfolio profit and loss, IB API

computing [309-312](#)

portfolio sensitivities

analyzing, to Fama-French factors [137-143](#)

position-level details, IB API

inspecting [306-308](#)

Postgres [114](#)

PostgreSQL database server

data, storing in [114-118](#)

principal component analysis (PCA) [97](#), [126](#)

latent return drivers, identifying with [126-131](#)

principal components [126](#)

profit and loss (PnL) [293](#)

PyCharm [275](#)

Pyfolio

Zipline backtest results, preparing for [224-229](#)

Pyfolio Reloaded (Pyfolio) [223](#)

Python

reference link [366](#)

used, for sending emails [366](#), [367](#)

Python generator [283](#)

Python list comprehension [201](#)

Q

Quandl [4](#)

quantile_turnover() function [218](#)

Quantitative Research (AQR) [216](#)

R

real-time key performance and risk indicators

calculating [314-317](#)

real-time options data

streaming, with ThetaData [348-356](#)

real-time risk limit alerts

triggering [364, 365](#)

rebalance function [186](#)

return series

volatility, measuring [57-60](#)

Rm-Rf [22](#)

rolling risk analysis

building [236-241](#)

key risk metrics [240](#)

rolling_split method [167](#)

lett_to_right parameter [167](#)

n parameter [167](#)

set_lens parameter [167](#)

window_len parameter [167](#)

S

SciPy stats module

reference link [168](#)

Seaborn

used, for visualizing statistical relationships [92-97](#)

sector allocations

analyzing [242-247](#)

set_commission function [195](#)

set_slippage function [195](#)

sharpe_ratio method [319](#)

Simple Notification Service (SNS)

reference link [368](#)

simple returns [53](#)

skew [234](#)

Small Minus Big (SMB) [22](#), [141](#)

SMS

used, for sending alerts [367](#), [368](#)

S&P 500 ratios data

exploring, with Nasdaq Data Link [8](#), [9](#)

SQL database

used, for storing trade execution details [368-372](#)

SQLite [105](#)

used, for storing data on disk [109-112](#)

statistical relationships

visualizing, with Seaborn [92-97](#)

stock market data

working with, OpenBB SDK used [10](#), [11](#)

strangle [334](#)

strategy holdings

analyzing [241](#)

strategy performance

breaking down, to trade level [248-251](#)

strategy performance and return analytics

generating [230-236](#)

strings

used, for creating columns [38](#), [39](#), [40](#)

SuperTrend indicator

reference link [178](#)

SuperTrend strategy

optimizing, with VectorBT [169-178](#)

T

technical strategies

building, with VectorBT [155-162](#)

ThetaData [347](#)

reference link [356](#)

used, for streaming real-time options data [348-356](#)

time series data

analyzing, with custom functions [73-76](#)

Trade Blotter [229](#)

trade execution details

contract object [373](#)

execution object [375](#)

order object [373](#)

order_state object [374](#)

storing, in SQL database [368-372](#)

Trader Workstation (TWS) [253](#), [333](#)

long straddle modeled, using Strategy Builder [336](#)

long straddle profile [336](#)

short iron condor profile [338](#)

short iron condor modeled, using Strategy Builder [337](#)

ttest_ind function [168](#)

Twilio [367](#)

reference link [368](#)

U

ultra-fast HDF5 format

 data, storing in [120-122](#)

underwater plot [238](#)

Unix

 executable shell file, creating [113](#), [114](#)

V

Value at Risk (VaR) [234](#), [364](#)

VectorBT

 installing, for Apple Silicon users [169](#)

 installing, for Mac Intel users [169](#)

 installing, for Unix/Linux users [169](#)

 installing, for Windows users [169](#)

 reference link [163](#)

 used, for building technical strategies [155-162](#)

 used, for implementing walk-forward optimization [163-167](#)

 used, for optimizing SuperTrend strategy [169-178](#)

VectorBT Pro

 reference link [170](#)

volatility method [318](#)

VSCode [275](#)

W

walk-forward optimization

 implementing, with VectorBT [163-167](#)

Windows

script, automating on [112](#), [113](#)

Y

yield curve

evolution, animating with Matplotlib [84](#)-[87](#)

Z

Zipline backtest results

preparing, for Pyfolio [224](#)-[229](#)

Zipline Pipelines

factor ranking model, preparing with [150](#)-[154](#)

Zipline Reloaded

installing, for Mac Intel users [179](#)

installing, for Mac M1/M2 users [180](#)

installing, for Unix/Linux users [179](#)

installing, for Windows users [179](#)

reference link [189](#)

used, for backtesting momentum factor strategy [180](#)-[188](#)

used, for exploring mean reversion strategy [189](#)-[197](#)

Zipline Reloaded, advanced features

reference link [198](#)

zipline-reloaded backtest

reference link [332](#)



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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 packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Developing High-Frequency Trading Systems

Learn how to implement high-frequency trading from scratch
with C++ or Java basics



Sebastien Donadio | Sourav Ghosh | Romain Rossier



Developing High-Frequency Trading Systems

Sebastien Donadio, Sourav Ghosh, Romain Rossier

ISBN: 978-1-80324-281-1

- Understand the architecture of high-frequency trading systems
- Boost system performance to achieve the lowest possible latency
- Leverage the power of Python programming, C++, and Java to build your trading systems
- Bypass your kernel and optimize your operating system
- Use static analysis to improve code development
- Use C++ templates and Java multithreading for ultra-low latency
- Apply your knowledge to cryptocurrency trading

Hands-On Financial Trading with Python

A practical guide to using Zipline and other Python libraries for backtesting trading strategies



Jiri Pik | Sourav Ghosh



Hands-On Financial Trading with Python

Jiri Pik, Sourav Ghosh

ISBN: 978-1-83898-288-1

- Discover how quantitative analysis works by covering financial statistics and ARIMA
- Use core Python libraries to perform quantitative research and strategy development using real datasets
- Understand how to access financial and economic data in Python
- Implement effective data visualization with Matplotlib
- Apply scientific computing and data visualization with popular Python libraries
- Build and deploy backtesting algorithmic trading strategies

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Algorithmic Trading with Python Cookbook*, we'd love to hear your thoughts! [If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.](#)

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781835084700>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

The Packt logo, featuring the word "packt" in a lowercase, sans-serif font with a small orange arrow pointing to the right integrated into the letter "p".

1ST EDITION

Python for Algorithmic Trading Cookbook

Recipes for designing, building, and deploying
algorithmic trading strategies with Python

**JASON STRIMPEL**