O'REILLY®

Python for Algorithmic Trading

From Idea to Cloud Deployment

Early Release
RAW & UNEDITED

Yves Hilpisch

Python for Algorithmic Trading

From Idea to Cloud Deployment

Yves Hilpisch

Python for Algorithmic Trading

by Yves Hilpisch

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Michelle Smith

Development Editor: Michele Cronin

Production Editor: Daniel Elfanbaum

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: O'Reilly Media

January 2021: First Edition

Revision History for the Early Release

- 2020-07-09: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492053354 for release details.

978-1-492-05328-6

[LSI]

Chapter 1. Python and Algorithmic Trading

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at hilpisch@gmail.com.

> *At Goldman [Sachs] the number of people engaged in trading shares has fallen from a peak of 600 in 2000 to just two today.*[1]
>
> —The Economist

This chapter provides background information for, and an overview of, the topics covered in this book. Although Python for algorithmic trading is a niche at the intersection of Python programming and finance, it is a fast-growing one that touches on such diverse topics as Python deployment, interactive financial analytics, machine and deep learning, object oriented programming, socket communication, visualization of streaming data, and trading platforms.

For a quick refresher on important Python topics, read [Link to Come] first.

Python for Finance

The Python programming language originated in 1991 with the first release by Guido van Rossum of a version labeled 0.9.0. In 1994, version 1.0 followed. However, it took almost two decades for Python to establish itself as a major programming language and technology platform in the financial industry. Of course, there were early adopters, mainly hedge funds, but widespread adoption probably started only around 2011.

One major obstacle to the adoption of Python in the financial industry has been the fact that the default Python version, called CPython, is an interpreted, high level language. Numerical algorithms in general and financial algorithms in particular are quite often implemented based on (nested) loop structures. While compiled, low level languages like C or C++ are really fast at executing such loops, Python — which relies on interpretation instead of compilation — is generally quite slow at doing so. As a consequence, pure Python proved too slow for many real-world financial applications, such as option pricing or risk management.

Python vs. Pseudo-Code

Although Python was never specifically targeted towards the scientific and financial communities, many people from these fields nevertheless liked the beauty and conciseness of its syntax. Not too long ago, it was generally considered good tradition to explain a (financial) algorithm and at the same time present some pseudo-code as an intermediate step towards its proper technological

implementation. Many felt that, with Python, the pseudo-code step would not be necessary anymore. And they were proven mostly correct.

Consider, for instance, the Euler discretization of the geometric Brownian motion as in Equation 1-1.

Equation 1-1. Euler discretization of geometric Brownian motion

$$S_T = S_0 \exp\left(\left(r - 0.5\sigma^2\right)T + \sigma z\sqrt{T}\right)$$

For decades, the Latex markup language and compiler have been the gold standard for authoring scientific documents containing mathematical formulae. In many ways, Latex syntax is similar to or already like pseudo-code when, for example, layouting equations as in Equation 1-1. In this particular case, the Latex version looks like this:

```
S_T = S_0 \exp((r - 0.5 \sigma^2) T + \sigma z
\sqrt{T})
```

In Python, this translates to executable code — given respective variable definitions — that is also really close to the financial formula as well as to the Latex representation:

```
S_T = S_0 * exp((r - 0.5 * sigma ** 2) * T + sigma * z *
sqrt(T))
```

However, the speed issue remains. Such a difference equation, as a numerical approximation of the respective stochastic differential equation, is generally used to price derivatives by Monte Carlo simulation or to do risk analysis and management based on simulation.[2] These tasks in turn can require millions of simulations that need to be finished in due time — often in almost real-time or at least near-time. Python, as an interpreted high-level programming language, was never designed to be fast enough to tackle such computationally demanding tasks.

NumPy and Vectorization

In 2006, version 1.0 of the `NumPy` Python package was released by Travis Oliphant. `NumPy` stands for *numerical Python*, suggesting that it targets scenarios that are numerically demanding. The base Python interpreter tries to be as general as possible in many areas, which often leads to quite a bit of overhead at run-time.[3] `NumPy`, on the other hand, uses specialization as its major approach to avoid overhead and to be as good and as fast as possible in *certain* application scenarios.

The major class of `NumPy` is the regular array object, called `ndarray` object for *n-dimensional array*. It is immutable, which means that it cannot be changed in size, and can only accommodate a single data type, called `dtype`. This specialization allows for the implementation of concise and fast code. One central approach in this context is *vectorization*. Basically, this approach avoids looping on the Python level and delegates the looping to specialized `NumPy` code, implemented in general in C and therefore rather fast.

Consider the simulation of 1,000,000 end of period values S_T according to Equation 1-1 with pure Python. The major part of the code below is a `for` loop with 1,000,000 iterations.

```
In [1]: %%time
        import random
        from math import exp, sqrt

        S0 = 100  ❶
        r = 0.05  ❷
        T = 1.0  ❸
        sigma = 0.2  ❹

        values = []  ❺

        for _ in range(1000000):  ❻
            ST = S0 * exp((r - 0.5 * sigma ** 2) * T +
                          sigma * random.gauss(0, 1) * sqrt(T))
❼

            values.append(ST)  ❽
        CPU times: user 923 ms, sys: 10.2 ms, total: 933 ms
        Wall time: 932 ms
```

❶ The initial index level.

❷ The constant short rate.

❸ The time horizon in year fractions.

❹ The constant volatility factor.

❺ An empty `list` object to collect simulated values.

❻ The main `for` loop.

❼ The simulation of a *single* end-of-period value.

❽ Appends the simulated value to the `list` object.

With `NumPy`, you can avoid looping on the Python level completely by the use of vectorization. The code is much more concise, more readable, and faster by a factor of about 25.

```
In [2]: %%time
        import numpy as np

        S0 = 100
        r = 0.05
        T = 1.0
        sigma = 0.2

        ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T +
                         sigma *
np.random.standard_normal(1000000) * np.sqrt(T))  ❶
        CPU times: user 97.3 ms, sys: 22 ms, total: 119 ms
        Wall time: 119 ms
```

❶ This single line of `NumPy` code simulates all the values and stores them in an `ndarray` object.

> **TIP**
>
> Vectorization is a powerful concept for writing concise, easy-to-read and easy-to-maintain code in finance and algorithmic trading. With `NumPy`, vectorized code does not only make code more concise, it also can speed up code execution considerably, like in the Monte Carlo simulation example by a factor of about 25.

It's safe to say that `NumPy` has significantly contributed to the success of Python in science and finance. Many other popular Python packages from the so-called *scientific Python stack* build on `NumPy` as an efficient, performing data structure to store and handle numerical data. In fact, `NumPy` is an outgrowth of the `SciPy` package project, which provides a wealth of functionality frequently needed in science. The `SciPy` project recognized the need for a more powerful numerical data structure and consolidated older projects like `Numeric` and `NumArray` in this area into a new, unifying one in the form of `NumPy`.

In algorithmic trading, Monte Carlo simulation might not be the most important use case for a programming language. However, if you enter the algorithmic trading space, the management of larger or even big financial time series data sets is, for example, a very important use case. Just think of the backtesting of (intraday) trading strategies or the processing of tick data streams during trading hours. This is where the `pandas` data analysis package comes into play (`pandas` home page).

pandas and the DataFrame Class

Development of `pandas` began in 2008 by Wes McKinney, who back then was working at AQR Capital Management, a big hedge fund operating out of Greenwich, Connecticut. Like for any other hedge fund, working with time series data is of paramount importance for AQR Capital Management, but back then Python did not provide any kind of appealing support for this type of data. Wes's idea was to create a package that mimics the capabilities of the R statistical language (*http://r-project.org*) in this area. This is reflected, for example, in naming the major class `DataFrame`, whose counterpart in R is called `data.frame`. Not being considered close enough to the core business of money management, AQR Capital Management open sourced the `pandas` project in 2009, which marks the beginning of a major success story in open source-based data and financial analytics.

Partly due to `pandas`, Python has become a major force in data and financial analytics. Many people who adopt Python, coming from diverse other languages, cite `pandas` as a major reason for their decision. In combination with open data sources like Quandl, `pandas` even allows students to do sophisticated financial analytics with the lowest barriers of entry ever: a regular notebook with an Internet connection suffices.

Assume an algorithmic trader is interested in trading Bitcoin, the cryptocurrency with the largest market capitalization. A first step might be to retrieve data about the historical exchange rate in USD. Using Quandl data and `pandas`, such a task is accomplished in less than a minute. Figure 1-1 shows the plot that results from the Python code below, which is (omitting some plotting style related

parameterizations) only four lines. Although `pandas` is not explicitly imported, the Quandl Python wrapper package by default returns a `DataFrame` object which is then used to add a simple moving average (SMA) of 100 days, as well as to visualize the raw data alongside the SMA.

```
In [3]: %matplotlib inline
        from pylab import mpl, plt  ❶
        plt.style.use('seaborn')  ❶
        mpl.rcParams['font.family'] = 'serif'  ❶

In [4]: import configparser  ❷
        c = configparser.ConfigParser()  ❷
        c.read('../pyalgo.cfg')  ❷
Out[4]: ['../pyalgo.cfg']

In [5]: import quandl as q  ❸
        q.ApiConfig.api_key = c['quandl']['api_key']  ❸
        d = q.get('BCHAIN/MKPRU')  ❹
        d['SMA'] = d['Value'].rolling(100).mean()  ❺
        d.loc['2013-1-1':].plot(title='BTC/USD exchange rate',
                                figsize=(10, 6));  ❻
```

❶ Imports and configures the plotting package.

❷ Imports the `configparser` module and reads credentials.

❸ Imports the Quandl Python wrapper package and provides the API key.

❹

Retrieves daily data for the Bitcoin exchange rate and returns a `pandas` `DataFrame` object with a single column.

❺ Calculates the SMA for 100 days in vectorized fashion.

❻ Selects data from 1st of January 2013 on and plots it.

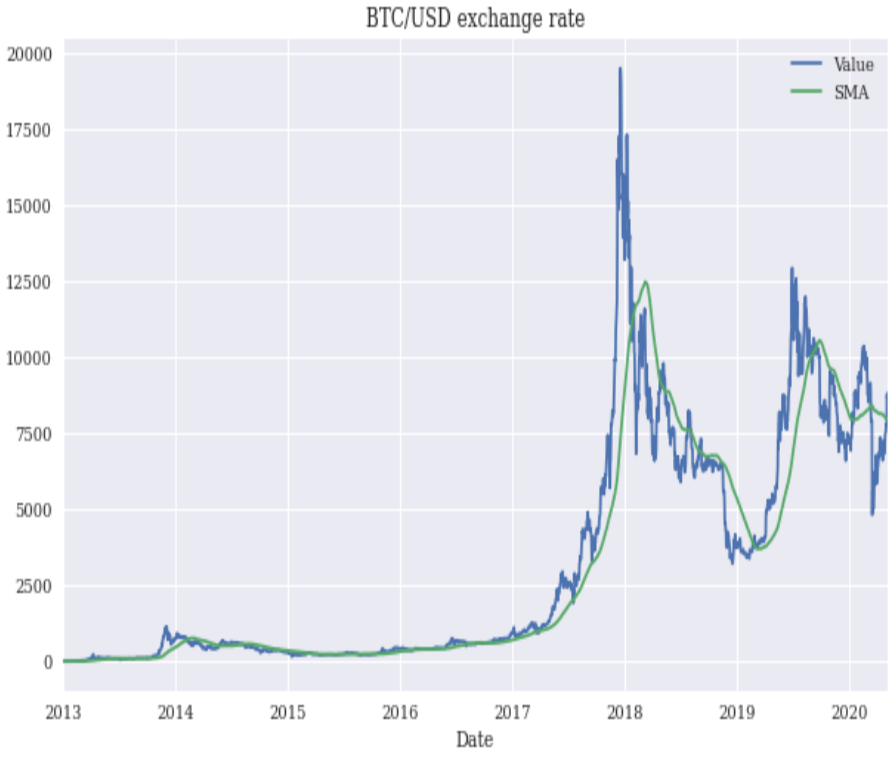

Figure 1-1. Historical Bitcoin exchange rate in USD from the beginning of 2013 until mid-2019

Obviously, `NumPy` and `pandas` measurably contribute to the success of Python in finance. However, the Python ecosystem has much more to offer in the form of additional Python packages that solve rather fundamental problems and sometimes also specialized ones. This book will make use of, among others, packages for data retrieval and

storage (e.g. `PyTables`, `TsTables`, `SQLite`) and for machine and deep learning (e.g. `scikit-learn`, `tensorflow`) — to name just two categories. Along the way, we will also implement classes and modules that will make any algorithmic trading project more efficient. But the main packages used throughout will be `NumPy` and `pandas`.

> **NOTE**
>
> While `NumPy` provides the basic data structure to store numerical data and work with it, `pandas` brings powerful time series management capabilities to the table. It also does a great job of wrapping functionality from other packages into an easy-to-use API. The Bitcoin example just described shows that a single method call on a `DataFrame` object is enough to generate a plot with two financial time series visualized. Like `NumPy`, `pandas` allows for rather concise, vectorized code that is also generally executed quite fast due to heavy use of compiled code under the hood.

Algorithmic Trading

The term *algorithmic trading* is neither uniquely nor universally defined. On a rather basic level, it refers to the trading of financial instruments based on some formal algorithm. An *algorithm* is a set of operations (mathematical, technical) to be conducted in a certain sequence to achieve a certain goal. For example, there are mathematical algorithms to solve a Rubik's cube.[4] Such an algorithm can solve the problem at hand via a step-by-step procedure, often perfectly. Another example is algorithms for finding the root(s) of an equation if it (they) exist(s) at all. In that sense, the objective of a

mathematical algorithm is often well specified and an optimal solution is often expected.

But what about the objective of financial trading algorithm? This question is not that easy to answer in general. It might help to step back for a moment and consider motives for trading in general. In Dorn et al. (2008), they write:

> *Trading in financial markets is an important economic activity. Trades are necessary to get into and out of the market, to put unneeded cash into the market, and to convert back into cash when the money is wanted. They are also needed to move money around within the market, to exchange one asset for another, to manage risk, and to exploit information about future price movements.*

The view expressed here is more technical than economic in nature, focusing mainly on the process itself and only partly on *why* people initiate trades in the first place. For our purposes, a non-exhaustive list of financial trading motives of people and also of financial institution managing money of their own or for others includes:

- **Beta trading**: Earning market risk premia by investing, for instance, in exchange traded funds (ETFs) that replicate the performance of the S&P 500.

- **Alpha generation**: Earning risk premia independent of the market by, for example, selling short stocks listed in the S&P 500 or ETFs on the S&P 500.

- **Static hedging**: Hedging against market risks by buying, for example, out-of-the-money put options on the S&P 500

- **Dynamic hedging**: Hedging against market risks affecting options on the S&P 500 by, for example, dynamically trading futures on the S&P 500 and appropriate cash, money market, or rate instruments

- **Asset-liability management**: Trading S&P 500 stocks and ETFs to be able to cover liabilities resulting from, for example, writing life insurance policies.

- **Market making**: Providing, for example, liquidity to options on the S&P 500 by buying and selling options at different bid and ask prices.

All these types of trades can be implemented by a discretionary approach, with the human trader making decisions mainly on his or her own. as well as based on algorithms supporting the human trader or even replacing him completely in the decision making process. In this context, computerization of financial trading of course plays an important role. While in the beginning of financial trading, floor trading with a large group of people shouting at each other ("open outcry") was the only way of executing trades, computerization and the advent of the Internet and web technologies have revolutionized trading in the financial industry. The quote at the beginning of this chapter illustrates this impressively in terms of the number of people actively engaged in financial at Goldman Sachs in 2000 and in 2016.

It is a trend that was foreseen 25 years ago, as Solomon and Corso (1991) point out:

> *Computers have revolutionized the trading of securities and the stock market is currently in the midst of a dynamic transformation. It is clear that the market of the future will not resemble the markets of the past.*
>
> *Technology has made it possible for information regarding stock prices to be sent all over the world in seconds. Presently, computers route orders and execute small trades directly from the brokerage firm's terminal to the exchange. Computers now link together various stock exchanges, a practice which is helping to create a single global market for the trading of securities. The continuing improvements in technology will make it possible to execute trades globally by electronic trading systems.*

Interestingly, one of the oldest and most widely used algorithms is found in dynamic hedging of options. Already with the publication of the seminal papers about the pricing of European options by Black and Scholes (1973) and Merton (1973), the algorithm, called *delta hedging*, was made available — long before computerized and electronic trading even started. Delta hedging as a trading algorithm shows how to hedge away all market risks in a simplified, perfect, continuous model world. In the real world, with transaction costs, discrete trading, imperfectly liquid markets, and other frictions ("imperfections"), the algorithm has proven — somewhat surprisingly maybe — its usefulness and robustness as well. It might not allow to perfectly hedge away market risks affecting options, but it is useful in

getting close to the ideal and is therefore still used on a large scale in the financial industry.[5]

This book focuses on algorithmic trading in the context of *alpha generating strategies*. Although there are more sophisticated definitions for alpha, for the purposes of this book alpha is seen as the difference between a trading strategy's return over some period of time and the return of the benchmark (single stock, index, cryptocurrency, etc.). For example, if the S&P 500 returns 10% in 2018 and an algorithmic strategy returns 12%, then alpha is +2% points. If the strategy returns 7%, then alpha is -3% points. In general, such numbers are not adjusted for risk, and other risk characteristics like maximal drawdown (period) are usually considered to be of second order importance, if at all.

> **NOTE**
>
> This book focuses on alpha-generating strategies, that is strategies that try to generate positive returns (above a benchmark) independent of the market's performance itself. Alpha is defined in this book in the simplest way as the excess return of a strategy over the benchmark financial instrument's performance.

There are other areas where trading-related algorithms play an important role. One is the *high frequency trading* (HFT) space, where speed is typically the discipline in which players compete.[6] The motives for HFT are diverse, but market making and alpha generation probably play a prominent role. Another one is *trade execution*, where algorithms are deployed to optimally execute certain non-

standard trades. Motives in this area might include the execution (at best possible prices) of large orders or the execution of an order with as little market and price impact as possible. A more subtle motive might be to disguise an order by executing it on a number of different exchanges.

An important question remains to be addressed: is there any advantage to using algorithms for trading instead of human research, experience, and discretion? This question can hardly be answered in any generality. For sure, there are human traders and portfolio managers who have earned, on average, more than their benchmark for investors over longer periods of time. The paramount example in this regard is Warren Buffett. On the other hand, statistical analyses show that the majority of active portfolio managers rarely beat relevant benchmarks consistently. Referring to the year 2015, Adam Shell writes:

> *Last year, for example, when the Standard & Poor's 500-stock index posted a paltry total return of 1.4% with dividends included, 66% of "actively managed" large-company stock funds posted smaller returns than the index … The longer-term outlook is just as gloomy, with 84% of large-cap funds generating lower returns than the S&P 500 in the latest five year period and 82% falling shy in the past 10 years, the study found.*[7]

In an empirical study published in December 2016, Harvey et al. (2016) write:

We analyze and contrast the performance of discretionary and systematic hedge funds. Systematic funds use strategies that are rules-based, with little or no daily intervention by humans … We find that, for the period 1996-2014, systematic equity managers underperform their discretionary counterparts in terms of unadjusted (raw) returns, but that after adjusting for exposures to well-known risk factors, the risk-adjusted performance is similar. In the case of macro, systematic funds outperform discretionary funds, both on an unadjusted and risk-adjusted basis.

Table 1-1 reproduces the major quantitative findings of the study by Harvey et al. (2016).[8] In the table, *factors* include traditional ones (equity, bonds, etc.), dynamic ones (value, momentum, etc.), and volatility (buying at-the-money puts and calls). The *adjusted return appraisal ratio* divides alpha by the adjusted return volatility. For more details and background, see the paper itself.

The study's results illustrate that systematic ("algorithmic") macro hedge funds perform best as a category, both in unadjusted and risk-adjusted terms. They generate an annualized alpha of 4.85% points over the period studied. These are hedge funds implementing strategies that are typically global, cross-asset, and often involve political and macroeconomic elements. Systematic equity hedge funds only beat their discretionary counterparts on the basis of the adjusted return appraisal ratio (0.35 vs. 0.25).

Table 1-1. Annualized performance of hedge fund categories

	systematic macro	discretionary macro	systematic equity	discretionary equity
return average	5.01%	2.86%	2.88%	4.09%
return attributed to factors	0.15%	1.28%	1.77%	2.86%
adj. return average (alpha)	4.85%	1.57%	1.11%	1.22%
adj. return volatility	10.93%	5.10%	3.18%	4.79%
adj. return appraisal ratio	0.44	0.31	0.35	0.25

Compared to the S&P 500, hedge fund performance over all was quite meager for the year 2017. While the S&P 500 index returned 21.8%, hedge funds only returned 8.5% to investors (see http://investopedia.com). This illustrates how hard it is — even with multi-million dollar budgets for research and technology — to generate alpha.

Python for Algorithmic Trading

Python is used in many corners of the financial industry, but has become particularly popular in the algorithmic trading space. There are a few good reasons for this:

- **Data analytics capabilities**: A major requirement for every algorithmic trading project is the ability to manage and process financial data efficiently. Python, in combination with packages like `NumPy` and `pandas`, makes life easier in this regard for every algorithmic trader than most other programming languages do.

- **Handling of modern APIs**: Modern online trading platforms like the ones from FXCM and Oanda offer RESTful application programming interfaces (APIs) and socket (streaming) APIs to access historical and live data. Python is in general well suited to efficiently interact with such APIs.

- **Dedicated packages**: In addition to the standard data analytics packages, there are multiple packages available that are dedicated to the algorithmic trading space, such as `PyAlgoTrade` and `Zipline` for the backtesting of trading strategies, and `Pyfolio` for performing portfolio and risk analysis.

- **Vendor sponsored packages**: More and more vendors in the space release open source Python packages to facilitate

access to their offerings. Among them are online trading platforms like Oanda as well as the leading data providers like Bloomberg and Refinitiv.

- **Dedicated platforms**: Quantopian, for example, offers a standardized backtesting environment as a Web-based platform where the language of choice is Python and where people can exchange ideas with like-minded others via different social network features. From its founding until 2020, Quantopian has attracted more than 300,000 users.

- **Buy- and sell-side adoption**: More and more institutional players have adopted Python to streamline development efforts in their trading departments. This, in turn, requires more and more staff proficient in Python, which makes learning Python a worthwhile investment.

- **Education, training, and books**: Prerequisites for the wide-spread adoption of a technology or programming language are academic and professional education and training programs in combination with specialized books and other resources. The Python ecosystem has seen a tremendous growth in such offerings recently, educating and training more and more people in the use of Python for finance. This can be expected to reinforce the trend of Python adoption in the algorithmic trading space.

In summary, it is rather safe to say that Python plays an important role in algorithmic trading already, and seems to have strong

momentum to become even more important in the future. It is therefore a good choice for anyone trying to enter the space, be it as an ambitious "retail" trader or as a professional employed by a leading financial institution engaged in systematic trading.

Focus and Prerequisites

The focus of this book is on Python as a programming language *for* algorithmic trading. The book assumes that the reader already has some experience with Python and popular Python packages used for data analytics. Good introductory books are, for example, Hilpisch (2018), McKinney (2017), and VanderPlas (2016), which all can be consulted to build a solid foundation in Python for data analysis and finance. The reader is also expected to have some experience with typical tools used for interactive analytics with Python, such as `IPython`, to which VanderPlas (2016) also provides an introduction.

This book presents and explains Python code that is applied to the topics at hand, like backtesting trading strategies or working with streaming data. It cannot provide a thorough introduction to all packages used in different places. It tries, however, to highlight those capabilities of the packages that are central to the exposition (such as vectorization with `NumPy`).

The book also cannot provide a thorough introduction and overview of all financial and operational aspects relevant for algorithmic trading. The approach instead focuses on the use of Python to build the necessary infrastructure for automated, algorithmic trading systems. Of course, the majority of examples used are taken from the

algorithmic trading space. However, when dealing with, say, momentum or mean-reversion strategies, they are more or less simply used without providing (statistical) verification or an in-depth discussion of their intricacies. Whenever it seems appropriate, references are given that point the reader to sources that address issues left open during the exposition.

All in all, this book is written for readers who have some experience with both Python and (algorithmic) trading. For such a reader, the book is a practical guide to the creation of automated trading systems using Python and additional packages.

> **CAUTION**
>
> This book uses a number of Python programming approaches (for example, object oriented programming) and packages (for example, `scikit-learn`) that cannot be explained in detail. The focus is on *applying* these approaches and packages to different steps in an algorithmic trading process. It is therefore recommended that those who do not yet have enough Python (for finance) experience additionally consult more introductory Python texts.

Trading Strategies

Throughout this book, four different algorithmic trading strategies are used as examples. They are introduced briefly below and in some more detail in [Link to Come]. All these trading strategies can be classified as mainly *alpha seeking strategies* since their main objective is to generate positive, above-market returns independent of the market direction. Canonical examples throughout the book when

it comes to financial instruments traded are a *stock index, a single stock*, or *a cryptocurrency* (denominated in a fiat currency). The book does not cover strategies involving multiple financial instruments at the same time (pair trading strategies, strategies based on baskets, etc.). It also covers only strategies whose trading signals are derived from structured, financial time series data and not, for instance, from unstructured data sources like news or social media feeds. This keeps the discussions and the Python implementations concise and easier to understand, in line with the approach (discussed earlier) of focusing on Python *for* algorithmic trading.[9]

The remainder of this section gives a quick overview of the four trading strategies used in this book.

Simple Moving Averages

The first type of trading strategy relies on simple moving averages (SMAs) to generate trading signals and market positionings. These trading strategies have been popularized by so-called technical analysts or chartists. The basic idea is that a shorter-term SMA being higher in value than a longer term SMA signals a long market position and the opposite scenario signals a neutral or short market position.

Momentum

The basic idea behind momentum strategies is that a financial instrument is assumed to perform in accordance with its recent performance for some additional time. For example, when a stock

index has seen a negative return on average over the last five days, it is assumed that its performance will be negative tomorrow as well.

Mean-Reversion

In mean-reversion strategies, a financial instrument is assumed to revert to some mean or trend level if it is currently far enough away from such a level. For example, assume that a stock trades 10 USD under its 200 days SMA level of 100. It is then expected that the stock price will return to its SMA level sometime soon.

Machine and Deep Learning

With machine and deep learning algorithms, one generally takes a more black box-like approach to predicting market movements. For simplicity and reproducibility, the examples in this book mainly rely on historical return observations as features to train machine and deep learning algorithms to predict stock market movements.

> **CAUTION**
>
> This book does not introduce algorithmic trading in a systematic fashion. Since the focus lies on applying Python in this fascinating field, readers not familiar with algorithmic trading should consult other, dedicated resources on the topic, some of which are cited in this chapter and the others that follow. But be aware of the fact that the algorithmic trading world in general is secretive and that almost everybody who is successful there is naturally reluctant to share his or her secrets in order to protect their sources of success, i.e. alpha.

Conclusions

Python is already a force in finance in general, and is on its way to becoming a major force in algorithmic trading. There are a number of good reasons to use Python for algorithmic trading, among them the powerful ecosystem of packages that allow for efficient data analysis or the handling of modern APIs. There are also a number of good reasons to learn Python for algorithmic trading, chief among them the fact that some of the biggest buy- and sell-side institutions make heavy use of Python in their trading operations and constantly look for seasoned Python professionals.

This book focuses on applying Python to the different disciplines in algorithmic trading, like backtesting trading strategies or interacting with online trading platforms. It cannot replace a thorough introduction to Python itself nor to trading in general. However, it systematically combines these two fascinating worlds to provide a valuable source for the generation of alpha in today's competitive financial and cryptocurrency markets.

Further Resources

Research papers cited in this chapter:

- Black, Fischer and Myron Scholes (1973): "The Pricing of Options and Corporate Liabilities." *Journal of Political Economy*, Vol. 81, No. 3, 638-659.

- Harvey, Campbell, Sandy Rattray, Andrew Sinclair and Otto Van Hemert (2016): "Man vs. Machine: Comparing

Discretionary and Systematic Hedge Fund Performance." White Paper, Man Group.

- Dorn, Anne, Daniel Dorn, and Paul Sengmueller (2008): "Why do People Trade?" *Journal of Applied Finance*, Fall/Winter, 37-50.

- Merton, Robert (1973): "Theory of Rational Option Pricing." *Bell Journal of Economics and Management Science*, Vol. 4, 141-183.

- Solomon, Lewis and Louise Corso (1991): "The Impact of Technology on the Trading of Securities: The Emerging Global Market and the Implications for Regulation." *The John Marshall Law Review*, Vol. 24, No. 2, 299-338.

Books cited in this chapter:

- Chan, Ernest (2013): *Algorithmic Trading*. John Wiley & Sons, Hoboken et al.

- Kissel, Robert (2013): *Algorithmic Trading and Portfolio Management*. Elsevier/Academic Press, Amsterdam et al.

- Lewis, Michael (2015): *Flash Boys*. W.W. Norton & Company, New York & London.

- Harari, Yuval Noah (2015): *Homo Deus — A Brief History of Tomorrow.* Harvill Secker, London.

- Hilpisch, Yves (2020): *Artificial Intelligence in Finance*. O'Reilly, Beijing et al. Resources under *http://aiif.tpq.io*.
- Hilpisch, Yves (2018): *Python for Finance*. 2nd ed., O'Reilly, Beijing et al. Resources under *http://pff.tpq.io*.
- Hilpisch, Yves (2015): *Derivatives Analytics with Python*. Wiley Finance. Resources under *http://dawp.tpq.io*.
- McKinney, Wes (2017): *Python for Data Analysis*. 2nd ed., O'Reilly, Beijing et al.
- Narang, Rishi (2013): *Inside the Black Box*. John Wiley & Sons, Hoboken et al.
- VanderPlas, Jake (2016): *Python Data Science Handbook*. O'Reilly, Beijing et al.

---

1 "Too Squid to Fail." The Economist, 29. October 2016.

2 For details, see Hilpisch (2018, ch. 12).

3 For example, `list` objects are not only mutable, which means that they can be changed in size, they can also contain almost any other kind of Python object, like `int`, `float`, `tuple` objects or `list` objects themselves.

4 See The Mathematics of the Rubik's Cube or Algorithms for Solving Rubik's Cube.

5 See Hilpisch (2015) for a detailed analysis of delta hedging strategies for European and American options using Python.

6 See the book by Lewis (2015) for a non-technical introduction to HFT.

7 Source: "66% of Fund Managers Can't Match S&P Results." USA Today, March 14, 2016.

8 Annualized performance (above the short term interest rate) and risk measures for hedge fund categories comprising a total of 9,000 hedge funds over the period from June 1996 to December 2014.

9 See the book by Kissel (2013) for an overview of topics related to algorithmic trading, the book by Chan (2013) for an in-depth discussion of momentum and mean-reversion strategies, or the book by Narang (2013) for a coverage of quantitative and HFT trading in general.

Chapter 2. Python Infrastructure

In building a house, there is the problem of the selection of wood.

It is essential that the carpenter's aim be to carry equipment that will cut well and, when he has time, to sharpen that equipment.

—Miyamoto Musashi (The Book of Five Rings)

For someone new to Python, Python deployment might seem all but straightforward. The same holds true for the wealth of libraries and packages that can be installed optionally. First of all, there is not only *one* Python. Python comes in many different flavors, like CPython, Jython, IronPython or PyPy. Then there is still the divide between Python 2.7 and the 3.x world. In what follows, the chapter focuses on *CPython*, the most popular version of the Python programming language, and here on *version 3.7*.

Even when focusing on CPython 3.7 (henceforth just "Python"), deployment is made difficult due to a number of reasons:

- The interpreter (a standard CPython installation) only comes with the so-called *standard library* (e.g. covering typical mathematical functions).

- Optional Python packages need to be installed separately — and there are hundreds of them.
- Compiling ("building") such non-standard packages on your own can be tricky due to dependencies and operating system-specific requirements.
- Taking care of such dependencies and of version consistency over time (maintenance) is often tedious and time consuming.
- Updates and upgrades for certain packages might cause the need for re-compiling a multitude of other packages.
- Changing or replacing one package might cause trouble in (many) other places.

Fortunately, there are tools and strategies available that help with the Python deployment issue. This chapter covers the following types of technologies that help with Python deployment:

- **Package manager**: Package managers like `pip` or `conda` help with the installing, updating and removing of Python packages. They also help with version consistency of different packages.
- **Virtual environment manager**: A virtual environment manager like `virtualenv` or `conda` allows to manage multiple Python installations in parallel (for example, to have both a Python 2.7 and 3.7 installation on a single machine or to test the most recent development version of a fancy Python package without risk).[1]
- **Container**: Docker containers represent complete file systems containing all pieces of a system needed to run a certain software, like code, runtime or system tools. For example, you can run a Ubuntu

19.10 operating system with a Python 3.7 install and the respective Python codes in a Docker container hosted on a machine running Mac OS or Windows 10, for example.

- **Cloud instance**: Deploying Python code for financial applications generally requires high availability, security, and also performance. These requirements can typically only be met by the use of professional compute and storage infrastructure that is nowadays available at attractive conditions in the form of fairly small to really large and powerful cloud instances. One benefit of a cloud instance (virtual server) compared to a dedicated server rented longer term is that users generally get charged only for the hours of actual usage. Another advantage is that such cloud instances are available literally in a minute or two if needed which helps agile development and also with scalability.

The structure of this chapter is as follows. "Conda as a Package Manager" introduces `conda` as a package manager for Python. "Conda as a Virtual Environment Manager" focuses on `conda` capabilities for virtual environment management. "Using Docker Containers" gives a brief overview of Docker as a containerization technology and focuses on the building of a Ubuntu-based container with Python 3.7 installation. "Using Cloud Instances" shows how to deploy Python and `Jupyter Lab` — as a powerful, browser-based tool suite — for Python development and deployment in the cloud.

The goal of this chapter is to have a proper Python installation with the most important tools as well as numerical, data analysis, and visualization packages available on a professional infrastructure. This combination then serves as the backbone for implementing and deploying the Python codes in later chapters, be it interactive financial analytics code or code in the form of scripts and modules.

Conda as a Package Manager

Although `conda` can be installed stand alone, an efficient way of doing it is via *Miniconda* — a minimal Python distribution including `conda` as a package and virtual environment manager.

Installing Miniconda

You can download the different versions of Miniconda on the Miniconda page. In what follows, the Python 3.7 64-bit version is assumed which is available for Linux, Windows and Mac OS. The main example in this sub-section is a session in a Ubuntu-based Docker container which downloads the Linux 64-bit installer via `wget` and then installs Miniconda. The code as shown should work — with maybe minor modifications — on any other Linux-based or Mac OS-based machine as well.[2]

```
$ docker run -ti -h pyalgo -p 11111:11111 ubuntu:latest
/bin/bash

root@pyalgo:/# apt-get update; apt-get upgrade -y
...
root@pyalgo:/# apt-get install -y gcc wget
...
root@pyalgo:/# cd root
root@pyalgo:~# wget \
> https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-
x86_64.sh \
> -O miniconda.sh
...
HTTP request sent, awaiting response... 200 OK
Length: 85055499 (81M) [application/x-sh]
Saving to: 'Miniconda.sh'

Miniconda.sh        100%[===================>]  81.12M   660KB/s
in 26s

2020-05-05 08:26:16 (3.08 MB/s) - 'Miniconda.sh' saved
[85055499/85055499]
```

```
root@pyalgo:~# bash miniconda.sh

Welcome to Miniconda3 4.8.2

In order to continue the installation process, please review the
license
agreement.
Please, press ENTER to continue
>>>
```

Simply pressing the `ENTER` key starts the installation process. After reviewing the license agreement, approve the terms by answering `yes`.

```
...Last updated February 25, 2020

Do you accept the license terms? [yes|no]
[no] >>> yes

Miniconda3 will now be installed into this location:
/root/miniconda3

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below

[/root/miniconda3] >>>
PREFIX=/root/miniconda3
Unpacking payload ...
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /root/miniconda3
...
  python             pkgs/main/linux-64::python-3.7.6-h0371630_2
...
Preparing transaction: done
```

```
Executing transaction: done
installation finished.
```

After you have agreed to the licensing terms and have confirmed the install location, you should allow Miniconda to prepend the new Miniconda install location to the `PATH` environment variable by answering `yes` once again.

```
Do you wish the installer to initialize Miniconda3
by running conda init? [yes|no]
[no] >>> yes
...
no change     /root/miniconda3/etc/profile.d/conda.csh
modified      /root/.bashrc

==> For changes to take effect, close and re-open your current
shell. <==

If you'd prefer that conda's base environment not be activated
on startup,
   set the auto_activate_base parameter to false:

conda config --set auto_activate_base false

Thank you for installing Miniconda3!
root@pyalgo:~#
```

After that, you might want to update `conda` since the Miniconda installer is in general not as regularly updated as `conda` itself.

```
root@pyalgo:~# export PATH="/root/miniconda3/bin/:$PATH"
root@pyalgo:~# conda update -y conda
...
root@pyalgo:~# echo ". /root/miniconda3/etc/profile.d/conda.sh"
>> ~/.bashrc
root@pyalgo:~# bash
(base) root@pyalgo:~#
```

After this rather simple installation procedure, there are now both a basic Python installation as well as `conda` available. The basic Python installation comes already with some nice batteries included like the `SQLite3` database engine. You might try out whether you can start Python in a *new shell instance* or after *appending the relevant path* to the respective environment variable (as done above) .

```
(base) root@pyalgo:~# python
Python 3.7.6 (default, Jan  8 2020, 19:59:22)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> print('Hello Python for Algorithmic Trading World.')
Hello Python for Algorithmic Trading World.
>>> exit()
(base) root@pyalgo:~#
```

Basic Operations with Conda

`conda` can be used to efficiently handle, among others, the installing, updating and removing of Python packages. The following list provides an overview of the major functions.

installing Python x.x

```
conda install python=x.x
```

updating Python

```
conda update python
```

installing a package

```
conda install $PACKAGE_NAME
```

updating a package

```
conda update $PACKAGE_NAME
```

removing a package

```
conda remove $PACKAGE_NAME
```

updating conda itself

```
conda update conda
```

searching for packages

```
conda search $SEARCH_TERM
```

listing installed packages

```
conda list
```

Given these capabilities, installing, for example, `NumPy` — as one of the most important packages of the so-called *scientific stack* — is a single command only. When the installation takes place on a machine with an Intel processor, the procedure automatically installs the Intel Math Kernel Library `mkl` which speeds up numerical operations not only for `NumPy` on Intel machines but also for a few other scientific Python packages.[3]

```
(base) root@pyalgo:~# conda install numpy
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /root/miniconda3

  added / updated specs:
    - numpy


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    blas-1.0                   |              mkl           6 KB
```

```
    intel-openmp-2020.0        |              166         756 KB
    libgfortran-ng-7.3.0       |       hdf63c60_0        1006 KB
    mkl-2020.0                 |              166       128.9 MB
    mkl-service-2.3.0          |   py37he904b0f_0         218 KB
    mkl_fft-1.0.15             |   py37ha843d7b_0         154 KB
    mkl_random-1.1.0           |   py37hd6b4f25_0         321 KB
    numpy-1.18.1               |   py37h4f9e942_0           5 KB
    numpy-base-1.18.1          |   py37hde5b4d6_1         4.2 MB
    ------------------------------------------------------------
                                           Total:       135.5 MB

The following NEW packages will be INSTALLED:

  blas               pkgs/main/linux-64::blas-1.0-mkl
  intel-openmp       pkgs/main/linux-64::intel-openmp-2020.0-166
  libgfortran-ng     pkgs/main/linux-64::libgfortran-ng-7.3.0-
hdf63c60_0
  mkl                pkgs/main/linux-64::mkl-2020.0-166
  mkl-service        pkgs/main/linux-64::mkl-service-2.3.0-
py37he904b0f_0
  mkl_fft            pkgs/main/linux-64::mkl_fft-1.0.15-
py37ha843d7b_0
  mkl_random         pkgs/main/linux-64::mkl_random-1.1.0-
py37hd6b4f25_0
  numpy              pkgs/main/linux-64::numpy-1.18.1-
py37h4f9e942_0
  numpy-base         pkgs/main/linux-64::numpy-base-1.18.1-
py37hde5b4d6_1


Proceed ([y]/n)? y


Downloading and Extracting Packages
numpy-base-1.18.1    | 4.2 MB    |
####################################### | 100%
mkl_fft-1.0.15       | 154 KB    |
####################################### | 100%
libgfortran-ng-7.3.0 | 1006 KB   |
####################################### | 100%
mkl_random-1.1.0     | 321 KB    |
####################################### | 100%
blas-1.0             | 6 KB      |
```

```
##################################### | 100%
mkl-service-2.3.0    | 218 KB    |
##################################### | 100%
mkl-2020.0           | 128.9 MB  |
##################################### | 100%
intel-openmp-2020.0  | 756 KB    |
##################################### | 100%
numpy-1.18.1         | 5 KB      |
##################################### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
(base) root@pyalgo:~#
```

Multiple packages can also be installed at once. The `-y` flag indicates that all (potential) questions shall be answered with `yes`.

```
(base) root@pyalgo:~# conda install -y ipython matplotlib pandas
\
> pytables scikit-learn scipy
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /root/miniconda3

  added / updated specs:
    - ipython
    - matplotlib
    - pandas
    - pytables
    - scikit-learn
    - scipy


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
```

```
    backcall-0.1.0             |           py37_0          20 KB
    ...
    zstd-1.3.7                 |       h0b5b093_0         401 KB
    ------------------------------------------------------------
                                           Total:       136.6 MB

The following NEW packages will be INSTALLED:

  backcall           pkgs/main/linux-64::backcall-0.1.0-py37_0
  ...
  zstd               pkgs/main/linux-64::zstd-1.3.7-h0b5b093_0



Downloading and Extracting Packages
icu-58.2             | 10.5 MB   |
######################################## | 100%
...
pygments-2.6.1       | 654 KB    |
######################################## | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
(base) root@pyalgo:~#
```

After the resulting installation procedure, some of the most important libraries for financial analytics are available in addition to the standard ones.

IPython

An improved interactive Python shell.

matplotlib

The standard plotting library for Python.

NumPy

Efficient handling of numerical arrays.

pandas

Management of tabular data, like financial time series data.

PyTables

A Python wrapper for the HDF5 library.

scikit-learn

A package for machine learning and related tasks.

SciPy

A collection of scientific classes and functions.

This provides a basic tool set for data analysis in general and financial analytics in particular. The next example uses `IPython` and draws a set of pseudo-random numbers with `NumPy`.

```
(base) root@pyalgo:~# ipython
Python 3.7.6 (default, Jan  8 2020, 19:59:22)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.13.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import numpy as np

In [2]: np.random.seed(100)

In [3]: np.random.standard_normal((5, 4))
Out[3]:
array([[-1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],
       [ 0.98132079,  0.51421884,  0.22117967, -1.07004333],
       [-0.18949583,  0.25500144, -0.45802699,  0.43516349],
       [-0.58359505,  0.81684707,  0.67272081, -0.10441114],
       [-0.53128038,  1.02973269, -0.43813562, -1.11831825]])

In [4]: exit
(base) root@pyalgo:~#
```

Executing `conda list` shows which packages are installed.

```
(base) root@pyalgo:~# conda list
# packages in environment at /root/miniconda3:
#
# Name                    Version                   Build
Channel
_libgcc_mutex             0.1                         main
asn1crypto                1.3.0                    py37_0
backcall                  0.1.0                    py37_0
...
yaml                      0.1.7                had09818_2
zlib                      1.2.11               h7b6447c_3
zstd                      1.3.7                h0b5b093_0
(base) root@pyalgo:~#
```

In case a package is not needed anymore, it is efficiently removed with `conda remove`.

```
(base) root@pyalgo:~# conda remove matplotlib
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /root/miniconda3

  removed specs:
    - matplotlib


The following packages will be REMOVED:

  cycler-0.10.0-py37_0
  ...
  tornado-6.0.4-py37h7b6447c_1


Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
```

```
   Executing transaction: done
   (base) root@pyalgo:~#
```

`conda` as a package manager is already quite useful. However, its full power only becomes evident when adding virtual environment management to the mix.

> **TIP**
>
> `conda` as a package manager makes installing, updating and removing of Python packages a pleasant experience. There is no need to take care of building and compiling packages on your own — which can be tricky sometimes given the list of dependencies a package specifies and given the specifics to be considered on different operating systems.

Conda as a Virtual Environment Manager

Having installed Miniconda with `conda` included provides a default Python installation depending on what version of Miniconda has been chosen. The virtual environment management capabilities of `conda` allow, for example, to add to a Python 3.7 default installation a completely separated installation of Python 2.7.x. To this end, `conda` offers the following functionality.

creating a virtual environment

```
conda create --name $ENVIRONMENT_NAME
```

activating an environment

```
conda activate $ENVIRONMENT_NAME
```

deactivating an environment

```
conda deactivate $ENVIRONMENT_NAME
```

removing an environment

```
conda env remove --name $ENVIRONMENT_NAME
```

export to an environment file

```
conda env export > $FILE_NAME
```

creating an environment from file

```
conda env create -f $FILE_NAME
```

listing all environments

```
conda info --envs
```

As a simple illustration, the example code that follows creates an environment called `py27`, installs `IPython` and executes a line of Python 2.7.x code. Although the support for Python 2.7 has ended, the example illustrates how legacy Python 2.7 code can easily be executed and tested.

```
(base) root@pyalgo:~# conda create --name py27 python=2.7
Collecting package metadata (current_repodata.json): done
Solving environment: failed with repodata from
current_repodata.json,
will retry with next repodata source.
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /root/miniconda3/envs/py27

  added / updated specs:
    - python=2.7


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    certifi-2019.11.28         |           py27_0         153 KB
```

```
    libffi-3.3                |       he6710b0_1          50 KB
    pip-19.3.1                |           py27_0         1.7 MB
    python-2.7.18             |       h15b4118_1         9.9 MB
    readline-8.0              |       h7b6447c_0         356 KB
    setuptools-44.0.0         |           py27_0         512 KB
    wheel-0.33.6              |           py27_0          42 KB
    ------------------------------------------------------------
                                          Total:        12.6 MB

The following NEW packages will be INSTALLED:

  _libgcc_mutex      pkgs/main/linux-64::_libgcc_mutex-0.1-main
  ca-certificates    pkgs/main/linux-64::ca-certificates-
2020.1.1-0
  ...
  zlib               pkgs/main/linux-64::zlib-1.2.11-h7b6447c_3


Proceed ([y]/n)? y


Downloading and Extracting Packages
python-2.7.18        | 9.9 MB    |
##################################### | 100%
...
wheel-0.33.6         | 42 KB     |
##################################### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate py27
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) root@pyalgo:~#
```

Notice how the prompt changes to include `(py27)` after the activation of the environment.

```
(base) root@pyalgo:~# conda activate py27
(py27) root@pyalgo:~# conda install -y ipython
...
Executing transaction: done
(py27) root@pyalgo:~#
```

Finally, this allows to use `IPython` with Python 2.7 syntax.

```
(py27) root@pyalgo:~# ipython
Python 2.7.18 |Anaconda, Inc.| (default, Apr 23 2020, 22:42:48)
Type "copyright", "credits" or "license" for more information.

IPython 5.8.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra
details.

In [1]: print "Hello Python for Algorithmic Trading World."
Hello Python for Algorithmic Trading World.

In [2]: exit
(py27) root@pyalgo:~#
```

As this example demonstrates, `conda` as a virtual environment manager allows to install different Python versions alongside each other. It also allows to install different versions of certain packages. The default Python installation is not influenced by such a procedure, nor are other environments which might exist on the same machine. All available environments can be shown via `conda info --envs`.

```
(py27) root@pyalgo:~# conda env list
# conda environments:
#
base                     /root/miniconda3
py27                  *  /root/miniconda3/envs/py27

(py27) root@pyalgo:~#
```

Sometimes, it is necessary to share environment information with others or to use environment information on multiple machines, for instance. To this end, one can export the installed packages list to a file with `conda env export`. However, this only works properly by default for the same operating system since the build versions are specified in the resulting `yaml` file. However, they can be deleted to only specify the package version via the `--no-builds` flag.

```
(py27) root@pyalgo:~# conda deactivate
(base) root@pyalgo:~# conda env export --no-builds > base.yml
(base) root@pyalgo:~# cat base.yml
name: base
channels:
  - defaults
dependencies:
  - _libgcc_mutex=0.1
  - asn1crypto=1.3.0
  - backcall=0.1.0
  - blas=1.0
  ...
  - yaml=0.1.7
  - zlib=1.2.11
  - zstd=1.3.7
prefix: /root/miniconda3

(base) root@pyalgo:~#
```

Often, virtual environments, which are technically not that much more than a certain (sub-)folder structure, are created to do some quick tests.[4] In such a

case, an environment is easily removed (after deactivation) via `conda env remove`.

```
(base) root@pyalgo:~# conda env remove -n py27

Remove all packages in environment /root/miniconda3/envs/py27:

(base) root@pyalgo:~#
```

This concludes the overview of `conda` as a virtual environment manager.

> **TIP**
>
> `conda` does not only help with managing packages, it is also a virtual environment manager for Python. It simplifies the creation of different Python environments, allowing to have multiple versions of Python and optional packages available on the same machine without influencing each other in any way. `conda` also allows to export environment information to easily replicate it on multiple machines or to share it with others.

Using Docker Containers

Docker containers have taken over the IT world by storm (see Docker). Although the technology is still relatively young, it has established itself as one of the benchmarks for the efficient development and deployment of almost any kind of software application.

For our purposes, it suffices to think of a Docker container as a separated ("containerized") file system that includes an operating system (for example, Ubuntu 20.04 LTS for server), a (Python) runtime, additional system and development tools as well as further (Python) libraries and packages as needed. Such a Docker container might run on a local machine with Windows 10 Professional 64 Bit or on a cloud instance with a Linux operating system, for instance.

This section does not allow to go into the exciting details of Docker containers. It is rather a concise illustration of what the Docker technology can do in the context of Python deployment.[5]

Docker Images and Containers

However, before moving on to the illustration, two fundamental terms need to be distinguished when talking about Docker. The first is a *Docker image* which can be compared to a Python class. The second is a *Docker container* which can be compared to an instance of the respective Python class.

On a more technical level, you find the following definition for a *Docker image* in the Docker glossary:

> *Docker images are the basis of containers. An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems stacked on top of each other. An image does not have state and it never changes.*

Similarly, you find the following definition for a *Docker container* in the Docker glossary which makes the analogy to Python classes and instances of such classes transparent:

> *A container is a runtime instance of a docker image.*
>
> *A Docker container consists of*
>
> - *A Docker image*
> - *An execution environment*
> - *A standard set of instructions*
>
> *The concept is borrowed from Shipping Containers, which define a standard to ship goods globally. Docker defines a standard to ship software.*

Depending on the operating system, the installation of Docker is somewhat different. That is why this section does not go into the respective details. More information and further links are found on the Get Docker page.

Building a Ubuntu & Python Docker Image

This sub-section illustrates the building of a Docker image based on the latest version of Ubuntu that includes Miniconda as well as a few important Python packages. In addition, it also does some Linux housekeeping by updating the Linux packages index, upgrading packages if required and installing certain, additional system tools. To this end, two scripts are needed. One is a `Bash` script doing all the work on the Linux level.[6] The other is a so-called `Dockerfile` which controls the building procedure for the image itself.

The `Bash` script in Example 2-1 which does the installing consists of three major parts. The first part handles the Linux housekeeping. The second part installs Miniconda while the third part installs optional Python packages. There are also more detailed comments inline.

Example 2-1. Script installing Python and optional packages

```
#!/bin/bash
#
# Script to Install
# Linux System Tools and
# Basic Python Components
#
# Python for Algorithmic Trading
# (c) Dr. Yves J. Hilpisch
# The Python Quants GmbH
#
# GENERAL LINUX
apt-get update  # updates the package index cache
apt-get upgrade -y  # updates packages
# installs system tools
apt-get install -y bzip2 gcc git  # system tools
apt-get install -y htop screen vim wget  # system tools
```

```
apt-get upgrade -y bash  # upgrades bash if necessary
apt-get clean  # cleans up the package index cache

# INSTALL MINICONDA
# downloads Miniconda
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-
Linux-x86_64.sh -O Miniconda.sh
bash Miniconda.sh -b  # installs it
rm -rf Miniconda.sh  # removes the installer
export PATH="/root/miniconda3/bin:$PATH"  # prepends the new path

# INSTALL PYTHON LIBRARIES
conda install -y pandas  # installs pandas
conda install -y ipython  # installs IPython shell

# CUSTOMIZATION
cd /root/
wget http://hilpisch.com/.vimrc  # Vim configuration
```

The `Dockerfile` in Example 2-2 uses the `Bash` script in Example 2-1 to build a new Docker image. It also has its major parts commented inline.

Example 2-2. Dockerfile to build the image

```
#
# Building a Docker Image with
# the Latest Ubuntu Version and
# Basic Python Install
#
# Python for Algorithmic Trading
# (c) Dr. Yves J. Hilpisch
# The Python Quants GmbH
#

# latest Ubuntu version
FROM ubuntu:latest

# information about maintainer
MAINTAINER yves

# add the bash script
```

```
ADD install.sh /
# change rights for the script
RUN chmod u+x /install.sh
# run the bash script
RUN /install.sh
# prepend the new path
ENV PATH /root/miniconda3/bin:$PATH

# execute IPython when container is run
CMD ["ipython"]
```

If these two files are in a single folder and Docker is installed, then the building of the new Docker image is straightforward. Here, the tag `pyalgo:basic` is used for the image. This tag is needed to reference the image, for example, when running a container based on it.

```
(base) mac:Docker$ docker build -t pyalgo:basic .
...
Removing intermediate container a9526fc21535
 ---> 49eeccc5273d
Step 6/7 : ENV PATH /root/miniconda3/bin:$PATH
 ---> Running in e17f6597c034
Removing intermediate container e17f6597c034
 ---> 88c98621f808
Step 7/7 : CMD ["ipython"]
 ---> Running in 96866d53d4af
Removing intermediate container 96866d53d4af
 ---> 67c68efe672a
Successfully built 67c68efe672a
Successfully tagged pyalgo:basic
(work) mac:Docker yves$
```

Existing Docker images can be listed via `docker images`. The new image should be on top of the list.

```
(work) mac:Docker yves$ docker images
REPOSITORY     TAG            IMAGE ID       CREATED
SIZE
```

```
pyalgo         basic          67c68efe672a   7 minutes ago
1.82GB
ubuntu         latest         1d622ef86b13   11 days ago
73.9MB
(work) mac:Docker yves$
```

Having built the `pyalgo:basic` image successfully allows to run a respective Docker container with `docker run`. The parameter combination `-ti` is needed for interactive processes running within a Docker container, like a shell process (see the Docker Run Reference page).

```
(work) mac:Docker yves$ docker run -ti pyalgo:basic
Python 3.7.6 (default, Jan  8 2020, 19:59:22)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.13.0 -- An enhanced Interactive Python. Type '?' for
help.

In [1]: import numpy as np

In [2]: np.random.seed(100)

In [3]: a = np.random.standard_normal((5, 3))

In [4]: import pandas as pd

In [5]: df = pd.DataFrame(a, columns=['a', 'b', 'c'])

In [6]: df
Out[6]:
          a         b         c
0 -1.749765  0.342680  1.153036
1 -0.252436  0.981321  0.514219
2  0.221180 -1.070043 -0.189496
3  0.255001 -0.458027  0.435163
4 -0.583595  0.816847  0.672721

In [7]:
```

Exiting `IPython` will exit the container as well since it is *the only* application running within the container. However, you can detach from a container via

```
Ctrl+p --> Ctrl+q
```

After having detached from the container, the `docker ps` command shows the running container (and maybe other currently running containers):

```
(work) mac:Docker yves$ docker ps
CONTAINER ID  IMAGE         COMMAND     CREATED       ...
NAMES
e4cc8a2a6432  pyalgo:basic  "ipython"   About a minute ago
busy_jang
9ab01ab5102f  ubuntu:latest "/bin/bash" 54 minutes ago
pedantic_mendel
(work) mac:Docker yves$
```

Attaching to the Docker container is accomplished by `docker attach $CONTAINER_ID` (notice that a few letters of the `CONTAINER ID` are enough):

```
(base) mac:Docker$ docker attach e4cc8a
In [7]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
a    5 non-null float64
b    5 non-null float64
c    5 non-null float64
dtypes: float64(3)
memory usage: 200.0 bytes

In [8]:
```

The `exit` command terminates `IPython` and therewith stops the Docker container as well. It can be removed by `docker rm`.

```
In [8]: exit
(base) mac:Docker yves$ docker rm e4cc8a
08a811
(base) mac:Docker yves$
```

Similarly, the Docker image `pyalgo:basic` can be removed via `docker rmi` if not needed any longer. While containers are relatively light weight, single images might consume quite a bit of storage. In the case of the `pyalgo:basic` image, the size is close to 2 GB. That is why you might want to regularly clean up the list of Docker images.

```
(work) mac:Docker yves$ docker rmi 67c68
Untagged: pyalgo:basic
Deleted:
sha256:67c68efe672a5d57482f1527732889ff8c5041828151b78423b76bb850d6d259

...
Deleted:
sha256:e2171e7f6ccf594bded356cecd6496177947bb456ec98f2093d0d855b5a31b95

(work) mac:Docker yves$
```

Of course, there is much more to say about Docker containers and their benefits in certain application scenarios. For the purposes of this book, they provide a modern approach to deploy Python, to do Python development in a completely separated (containerized) environment and to ship codes for algorithmic trading.

> **TIP**
>
> If you are not yet using Docker containers, you should consider starting to use them. They provide a number of benefits when it comes to Python deployment and development efforts, not only when working locally but in particular when working with remote cloud instances and servers deploying code for algorithmic trading.

Using Cloud Instances

This section shows how to set up a full-fledged Python infrastructure on a DigitalOcean cloud instance. There are many other cloud providers out there, among them Amazon Web Services (AWS) as the leading provider. However, DigitalOcean is well known for its simplicity and also its relatively low rates for smaller cloud instances, which they call *Droplet*. The smallest Droplet, which is generally sufficient for exploration and development purposes, only costs 5 USD per month or 0.007 USD per hour. Usage is charged by the hour so that one can easily spin up a Droplet for 2 hours, say, destroy it afterwards and get charged just 0.014 USD.[7]

The goal of this section is to set up a Droplet on DigitalOcean that has a Python 3.7 installation plus typically needed packages (such as `NumPy` and `pandas`) in combination with a password-protected and Secure Sockets Layer (SSL)-encrypted Jupyter Lab server installation.[8] As a web-based tool suite, `Jupyter Lab` provides several tools that can be used via a regular browser:

- **Jupyter Notebook**: This is one of the most popular — if not *the* most popular — browser-based, interactive development environment that features a selection of different language kernels like Python, R and Julia.

- **Python console**: This is an `IPython`-based console that has a graphical user interface different from the look and feel of the standard, terminal-based implementation.

- **Terminal**: A system shell implementation accessible via the browser which allows for all typical system administration tasks but also for usage of such helpful tools like `Vim` for code editing or `git` for version control.

- **Editor**: Another major tool is a browser-based text file editor with syntax highlighting for many different programming languages and file types as well as typical text/code editing capabilities.

- **File manager**: `Jupyter Lab` also provides a full-fledged file manager that allows for typical file operations, such as uploading, downloading, renaming, and so on.

Having `Jupyter Lab` installed on a Droplet allows to do Python development and deployment via the browser, circumventing the need to log in to the cloud instance via Secure Shell (SSH) access.

To accomplish the goal of this section, a number of files is needed.

- **Server set-up script**: This script orchestrates all steps necessary, like, for instance, copying other files to the Droplet and running them on the Droplet.

- **Python and `Jupyter` installation script**: This script installs Python, additional packages, `Jupyter Lab` and starts the `Jupyter Lab` server

- **Jupyter Notebook configuration file**: this file is for the configuration of the `Jupyter Lab` server, for example, with regard to password protection.

- **RSA public and private key files**: These two files are needed for the SSL encryption of the communication with the `Jupyter Lab` server.

In what follows, the section works backwards through this list of files — since the set-up script is executed first but the other files need to have been created beforehand.

RSA Public and Private Keys

In order to accomplish a secure connection to the `Jupyter Lab` server via an arbitrary browser, a SSL certificate consisting of RSA public and private keys (see RSA Wikipedia page) is needed. In general, one would expect that such a certificate comes from a so-called Certificate Authority (CA). For the purposes of this book, however, a self-generated certificate is "good enough".[9] A popular tool to generate RSA key pairs is `OpenSSL`. The brief interactive session to follow generates a certificate appropriate for use with a `Jupyter Lab` server (see Running a notebook server).

```
(work) mac:cloud yves$ openssl req -x509 -nodes -days 365 -
newkey rsa:2048 \
> -keyout mykey.key -out mycert.pem
Generating a RSA private key
.........++++
\begin{equation}+
..........\end{equation}
+++++
writing new private key to 'mykey.key'
-----
You are about to be asked to enter information that will be
incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:DE
State or Province Name (full name) [Some-State]:Saarland
Locality Name (eg, city) []:Voelklingen
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TPQ
GmbH
Organizational Unit Name (eg, section) []:Algorithmic Trading
Common Name (e.g. server FQDN or YOUR name) []:Jupyter Lab
Email Address []:pyalgo@tpq.io
(work) mac:cloud yves$
```

The two files `mykey.key` and `mycert.pem` need to be copied to the Droplet and need to be referenced by the `Jupyter Notebook` configuration file. This file is presented next.

Jupyter Notebook Configuration File

A public `Jupyter Lab` server can be deployed securely as explained on the Running a notebook server. Among others, `Jupyter Lab` shall be password protected. To this end, there is a password hash code-generating function called `passwd()` available in the `notebook.auth` sub-package. The code below generates a password hash code with `jupyter` being the password itself.

```
(work) mac:cloud yves$ ipython
Python 3.7.6 | packaged by conda-forge | (default, Jan  7 2020,
22:05:27)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.13.0 -- An enhanced Interactive Python. Type '?' for
help.

In [1]: from notebook.auth import passwd

In [2]: passwd('jupyter')
Out[2]:
'sha1:da3a3dfc0445:052235bb76e56450b38d27e41a85a136c3bf9cd7'

In [3]: exit
(work) mac:cloud yves$
```

This hash code needs to be placed in the `Jupyter Notebook` configuration file as presented in Example 2-3. The configuration file assumes that the RSA key files have been copied on the Droplet to the `/root/.jupyter/` folder.

Example 2-3. Jupyter Notebook configuration file

```
#
# Jupyter Notebook Configuration File
#
# Python for Algorithmic Trading
# (c) Dr. Yves J. Hilpisch
# The Python Quants GmbH
#
# SSL ENCRYPTION
```

```
# replace the following file names (and files used) by your choice/files
c.NotebookApp.certfile = u'/root/.jupyter/mycert.pem'
c.NotebookApp.keyfile = u'/root/.jupyter/mykey.key'

# IP ADDRESS AND PORT
# set ip to '*' to bind on all IP addresses of the cloud instance
c.NotebookApp.ip = '0.0.0.0'
# it is a good idea to set a known, fixed default port for server access
c.NotebookApp.port = 8888

# PASSWORD PROTECTION
# here: 'jupyter' as password
# replace the hash code with the one for your password
c.NotebookApp.password = \
         'sha1:da3a3dfc0445:052235bb76e56450b38d27e41a85a136c3bf9cd7'

# NO BROWSER OPTION
# prevent Jupyter from trying to open a browser
c.NotebookApp.open_browser = False

# ROOT ACCESS
# allow Jupyter to run from root user
c.NotebookApp.allow_root = True
```

CAUTION

Deploying `Jupyter Lab` in the cloud leads to a number of security issues since it is a full-fledged development environment accessible via a web browser. It is therefore of paramount importance to use the security measures that a `Jupyter Lab` server provides by default, like password protection and SSL encryption. But this is just the beginning and further security measures might be advised depending on what exactly is done on the cloud instance.

The next step is to make sure that Python and `Jupyter Lab` get installed on the Droplet.

Installation Script for Python and Jupyter Notebook

The bash script to install Python and `Jupyter Lab` is similar to the one presented in section "Using Docker Containers" to install Python via Miniconda in a Docker container. However, the script here needs to start the `Jupyter Lab` server as well. All major parts and lines of code are commented inline.

Example 2-4. Bash script to install Python and to run the `Jupyter Notebook` server

```
#!/bin/bash
#
# Script to Install
# Linux System Tools and Basic Python Components
# as well as to
# Start Jupyter Lab Server
#
# Python for Algorithmic Trading
# (c) Dr. Yves J. Hilpisch
# The Python Quants GmbH
#
# GENERAL LINUX
apt-get update  # updates the package index cache
apt-get upgrade -y  # updates packages
# install system tools
apt-get install -y gcc git htop  # system tools
apt-get install -y screen htop vim wget  # system tools
apt-get upgrade -y bash  # upgrades bash if necessary
apt-get clean  # cleans up the package index cache

# INSTALLING MINICONDA
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-
Linux-x86_64.sh \
                -O Miniconda.sh
bash Miniconda.sh -b  # installs Miniconda
rm -rf Miniconda.sh  # removes the installer
# prepends the new path for current session
export PATH="/root/miniconda3/bin:$PATH"
# prepends the new path in the shell configuration
cat >> ~/.profile <<EOF
```

```
export PATH="/root/miniconda3/bin:$PATH"
EOF

# INSTALLING PYTHON LIBRARIES
conda install -y jupyter  # interactive data analytics in the browser
conda install -y jupyterlab  # Jupyter Lab environment
conda install -y numpy  #  numerical computing package
conda install -y pytables  # wrapper for HDF5 binary storage
conda install -y pandas  #  data analysis package
conda install -y matplotlib  # standard plotting library
conda install -y seaborn  # statistical plotting library
conda install -y quandl  # wrapper for Quandl data API
conda install -y scikit-learn  # machine learning library
conda install -y openpyxl  # package for Excel interaction
conda install -y xlrd xlwt  # packages for Excel interaction
conda install -y pyyaml  # package to manage yaml files

pip install --upgrade pip  # upgrading the package manager
pip install q  # logging and debugging
pip install plotly  # interactive D3.js plots
pip install cufflinks  # combining plotly with pandas
pip install tensorflow  # deep learning library
pip install keras  # deep learning library
pip install eikon  # Python wrapper for the Refinitiv Eikon Data API
# Python wrapper for Oanda API
pip install git+git://github.com/yhilpisch/tpqoa

# COPYING FILES AND CREATING DIRECTORIES
mkdir /root/.jupyter
mkdir /root/.jupyter/custom
wget http://hilpisch.com/custom.css
mv custom.css /root/.jupyter/custom
mv /root/jupyter_notebook_config.py /root/.jupyter/
mv /root/mycert.pem /root/.jupyter
mv /root/mykey.key /root/.jupyter
mkdir /root/notebook
cd /root/notebook

# STARTING JUPYTER LAB
jupyter lab &
```

This script needs to be copied to the Droplet and needs to be started by the orchestration script as described in the next sub-section.

Script to Orchestrate the Droplet Set-up

The second bash script which sets up the Droplet is the shortest one. It mainly copies all the other files to the Droplet for which the respective IP address is expected as a parameter. In the final line, it starts the `install.sh` bash script which in turn does the installation itself and starts the `Jupyter Lab` server.

Example 2-5. `Bash` script to setup the Droplet

```
#!/bin/bash
#
# Setting up a DigitalOcean Droplet
# with Basic Python Stack
# and Jupyter Notebook
#
# Python for Algorithmic Trading
# (c) Dr Yves J Hilpisch
# The Python Quants GmbH
#

# IP ADDRESS FROM PARAMETER
MASTER_IP=$1

# COPYING THE FILES
scp install.sh root@${MASTER_IP}:
scp mycert.pem mykey.key jupyter_notebook_config.py
root@${MASTER_IP}:

# EXECUTING THE INSTALLATION SCRIPT
ssh root@${MASTER_IP} bash /root/install.sh
```

Everything now is together to give the set-up code a try. On DigitalOcean, create a new Droplet with options similar to these:

- **Operating system**: Ubuntu 20.04 LTS x64 (the newest version available at the time of this writing)
- **Size**: 2 core, 2GB, 60GB SSD (standard Droplet)
- **data center region**: Frankfurt (since your author lives in Germany)
- **SSH key**: Add a (new) SSH key for password-less login.[10]
- **Droplet name**: You can go with the pre-specified name or you can choose something like `pyalgo`.

Finally, clicking on the `Create` button initiates the Droplet creation process which generally takes about one minute. The major outcome for proceeding with the set-up procedure is the IP address which might be, for instance, 167.172.165.98 when you have chosen Frankfurt as your data center location. Setting up the Droplet now is as easy as follows:

```
(base) mac:cloud$ bash setup.sh 167.172.165.98
```

The resulting process, however, might take a couple of minutes. It is finished when there is a message from the `Jupyter Lab` server saying something like:

```
[I 10:24:22.324 LabApp] Serving notebooks from local directory:
/root/notebook
[I 10:24:22.324 LabApp] The Jupyter Notebook is running at:
[I 10:24:22.324 LabApp] https://pyalgo:8888/
```

In any current browser, visiting the following address accesses the running `Jupyter Notebook` server (note the `https` protocol):

```
https://167.172.165.98:8888
```

After maybe adding a security exception, the `Jupyter Notebook` login screen prompting for a password (in our case `jupyter`) should appear. Everything is now ready to start Python development in the browser via `Jupyter Lab`, the `IPython`-based console, via a terminal window or the text file editor. Other file management capabilities like file upload, deletion of files or creation of folders are also available.

> **TIP**
>
> Cloud instances like those from DigitalOcean and `Jupyter Lab` (powered by the `Jupyter Notebook` server) are a powerful combination for the Python developer and algorithmic trading practitioner to work on and make use of professional compute and storage infrastructure. Professional cloud and data center providers make sure that your (virtual) machines are physically secure and highly available. Using cloud instances also keeps the exploration and development phase at rather low costs since usage generally gets charged by the hour without the need to enter long term agreements.

Conclusions

Python is the programming language and technology platform of choice, not only for this book but for almost every leading financial institution. However, Python deployment can be tricky at best and sometimes even tedious and nerve wrecking. Fortunately, technologies are available today — all in general younger than ten years — that help with the deployment issue. The open source software `conda` helps with both Python package and virtual environment management. Docker containers go even further in that complete file systems and runtime environments can be easily created in a technically shielded "sandbox", that is the *container*. Going even one step further, cloud providers like DigitalOcean offer compute and storage capacity in professionally managed and secured data centers within minutes and billed by the hour. This in combination with a Python 3.7 installation and a secure `Jupyter Notebook/Lab` server installation provides a professional environment for

Python development and deployment in the context of Python for algorithmic trading projects.

Further Resources

For *Python package management*, consult the following resources:

- `pip` package manager page
- `conda` package manager page
- official Installing Packages page

For *virtual environment management*, consult these resources:

- `virtualenv` environment manager page
- `conda` Managing Environments page
- `pipenv` package and environment manager

Information about *Docker containers* is found, among others, here:

- Docker home page
- Matthias, Karl and Sean Kane (2018): *Docker: Up and Running*. 2nd ed., O'Reilly, Beijing et al.

Robbins (2016) provides a concise introduction to and overview of the `Bash` *scripting language*.

- Robbins, Arnold (2016): *Bash Pocket Reference*. 2nd ed., O'Reilly, Beijing et al.

How to *run a public Jupyter Notebook/Lab server securely* is explained under Running a notebook server. There is also `JupyterHub` available which allows the management of multiple users for a `Jupyter Notebook` server — see JupyterHub.

To sign up on DigitalOcean with a 10 USD starting balance in your new account visit the page *http://bit.ly/do_sign_up*. This pays for two months of usage for the smallest Droplet.

---

1 A recent project called `pipenv` combines the capabilities of the package manager `pip` with those of the virual environment manager `virtualenv`. See *https://github.com/pypa/pipenv*.

2 On Windows, you can also run the exact same commands in a Docker container (see *https://docs.docker.com/docker-for-windows/install/)*. Working on Windows directly requires some adjustments. See, for example, the book Matthias and Kane (2018) for further details on Docker usage.

3 Installing the meta package `nomkl`, such as in `conda install numpy nomkl`, avoids the automatic installation and usage of `mkl` and related other packages.

4 In the official documentation you find the following explanation: "Python *Virtual Environments* allow Python packages to be installed in an isolated location for a particular application, rather than being installed globally." See the Creating Virtual Environments page.

5 See the book Matthias and Kane (2018) for a comprehensive introduction to the Docker technology.

6 Consult the book by Robbins (2016) for a concise introduction to and a quick overview of `Bash` scripting. Also see see GNU Bash.

7 For those who do not have an account with a cloud provider yet, on this page *http://bit.ly/do_sign_up* new users get a starting credit of 10 USD for DigitalOcean.

8 Technically, `Jupyter Lab` is an extension of `Jupyter Notebook`. Both expressions are, however, sometimes used interchangebly.

9 With such a self-generated certificate you might need to add a security exception when prompted by the browser. On Mac OS you might even explicitely register the certificate as trustworthy.

10 If you need assistance, visit either How To Use SSH Keys with DigitalOcean Droplets or How To Use SSH Keys with PuTTY on DigitalOcean Droplets (Windows users).

Chapter 3. Working with Financial Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at hilpisch@gmail.com.

> *Clearly, data beats algorithms. Without comprehensive data, you tend to get non-comprehensive predictions.*[1]
>
> —Rob Thomas

In algorithmic trading, one generally has to deal with four types of data as illustrated in Table 3-1. Although simplifying the real data world, distinguishing data along the pairs *historical vs. real-time* and *structured vs. unstructured* proves often useful in technical settings.

Table 3-1. Types of financial data (examples)

	structured	unstructured
historical	end-of-day closing prices	financial news articles
real-time	bid/ask prices for FX	posts on Twitter

This book is mainly concerned with *structured data* (numerical, tabular data) of both historical and real-time types. This chapter in particular focuses on historical, structured data, like end-of-day closing values for the SAP SE stock traded at the Frankfurt Stock Exchange. However, this category also subsumes intraday data, like, for example, 1-minute-bar data for the Apple, Inc. stock traded at the NASDAQ stock exchange. The processing of real-time, structured data is covered in [Link to Come].

An algorithmic trading project typically starts with a trading idea or hypothesis which needs to be (back-)tested based on historical financial data. This is the context for this chapter, the plan for which is as follows. "Reading Financial Data From Different Sources" uses `pandas` to read data from different file- and web-based sources. "Working with Open Data Sources" introduces Quandl as a popular open data source platform. "Eikon Data API" introduces the Python wrapper for the Refinitiv Eikon Data API. Finally, "Storing Financial Data Efficiently" briefly shows how to store historical, structured data efficiently with `pandas` based on the HDF5 binary storage format.

The goal for this chapter is to have available financial data in a format with which the backtesting of trading ideas and hypotheses can be implemented effectively. The three major themes are the importing of data, the handling of the data and the storage of it. This and sub-sequent chapters assume a Python 3.7 installation with Python packages installed as explained in detail in Chapter 2. For the time being, it is not yet relevant on which infrastructure exactly this Python environment is provided. For more details on efficient input-output operations with Python, see Hilpisch (2018, ch. 9).

Reading Financial Data From Different Sources

This section makes heavy use of the capabilities of `pandas`, the popular data analysis package for Python (see `pandas` home page). `pandas` comprehensively supports the three main tasks this chapter is concerned with: *reading data*, *handling data* and *storing data*. One of its strengths is the reading of data from different types of sources as the remainder of this section illustrates.

The Data Set

In this section, we work with a fairly small data set for the Apple, Inc. stock price (with symbol `AAPL` and Reuters Instrument Code or RIC `AAPL.O`) as retrieved from the Eikon Data API for April 2020.

Having stored such historical financial data in a CSV file on disk, pure Python can be used to read and print its content.

```
In [1]: fn = '../data/AAPL.csv'  ❶

In [2]: with open(fn, 'r') as f:  #  ❶
            for _ in range(5):  ❷
                print(f.readline(), end='')  ❸
        Date,HIGH,CLOSE,LOW,OPEN,COUNT,VOLUME
        2020-04-01,248.72,240.91,239.13,246.5,460606.0,44054638.0
        2020-04-02,245.15,244.93,236.9,240.34,380294.0,41483493.0
        2020-04-
03,245.7,241.41,238.9741,242.8,293699.0,32470017.0
        2020-04-06,263.11,262.47,249.38,250.9,486681.0,50455071.0
```

❶ Open the file on disk (adjust path and filename if necessary).

❷ Sets up a `for` loop with 5 iterations.

❸ Prints the first 5 lines in the opened CSV file.

This approach allows for simple inspection of the data. One learns that there is a header line and that the single data points per row represent `Date, OPEN, HIGH, LOW, CLOSE, COUNT` and `VOLUME`, respectively. However, the data is not yet available in memory for further usage with Python.

Reading from a CSV File with Python

To work with data stored as a CSV file, the file needs to be parsed and the data needs to be stored in a Python data structure. Python has a built-in module called `csv` which supports the reading of data from

a CSV file. The first approach yields a `list` object containing other `list` objects with the data from the file.

```
In [3]: import csv ❶

In [4]: csv_reader = csv.reader(open(fn, 'r')) ❷

In [5]: data = [l for l in csv_reader] ❸

In [6]: data[:5] ❹
Out[6]: [['Date', 'HIGH', 'CLOSE', 'LOW', 'OPEN', 'COUNT',
'VOLUME'],
         ['2020-04-01',
          '248.72',
          '240.91',
          '239.13',
          '246.5',
          '460606.0',
          '44054638.0'],
         ['2020-04-02',
          '245.15',
          '244.93',
          '236.9',
          '240.34',
          '380294.0',
          '41483493.0'],
         ['2020-04-03',
          '245.7',
          '241.41',
          '238.9741',
          '242.8',
          '293699.0',
          '32470017.0'],
         ['2020-04-06',
          '263.11',
```

```
            '262.47',
            '249.38',
            '250.9',
            '486681.0',
            '50455071.0']]
```

❶ Imports the `csv` module.

❷ Instantiates a `csv.reader` iterator object.

❸ A `list` comprehension adding every single line from the CSV file as a `list` object to the resulting `list` object.

❹ Prints out the first five elements of the `list` object.

Working with such a nested `list` object — e.g. for the calculation of the average closing price — is possible in principle but not really efficient or intuitive. Using a `csv.DictReader` iterator object instead of the standard `csv.reader` object makes such tasks a bit more manageable. Every row of data in the CSV file (apart from the header row) is then imported as a `dict` object so that single values can be accessed via the respective key.

```
In [7]: csv_reader = csv.DictReader(open(fn, 'r'))  ❶

In [8]: data = [l for l in csv_reader]

In [9]: data[:3]
Out[9]: [OrderedDict([('Date', '2020-04-01'),
                      ('HIGH', '248.72'),
```

```
                    ('CLOSE', '240.91'),
                    ('LOW', '239.13'),
                    ('OPEN', '246.5'),
                    ('COUNT', '460606.0'),
                    ('VOLUME', '44054638.0')]),
             OrderedDict([('Date', '2020-04-02'),
                    ('HIGH', '245.15'),
                    ('CLOSE', '244.93'),
                    ('LOW', '236.9'),
                    ('OPEN', '240.34'),
                    ('COUNT', '380294.0'),
                    ('VOLUME', '41483493.0')]),
             OrderedDict([('Date', '2020-04-03'),
                    ('HIGH', '245.7'),
                    ('CLOSE', '241.41'),
                    ('LOW', '238.9741'),
                    ('OPEN', '242.8'),
                    ('COUNT', '293699.0'),
                    ('VOLUME', '32470017.0')])]
```

❶ Here, the `csv.DictReader` iterator object is instantiated which reads every data row into a `dict` object — given the information in the header row.

Based on the single `dict` objects, aggregations are now somewhat more easy to accomplish. However, one still cannot speak of a convenient way of calculating the mean of the Apple closing stock price when inspecting the respective Python code.

```
In [10]: sum([float(l['CLOSE']) for l in data]) / len(data)  ❶
Out[10]: 272.38619047619045
```

❶ First, a `list` object is generated via a list comprehension with all closing values; second, the sum is taken over all these values; third, the resulting sum is divided by the number of closing values.

This is one of the major reasons why `pandas` has gained such a popularity in the Python community. It makes the importing of data and the handling of, for example, financial time series data sets more convenient (and also often considerably faster) than pure Python.

Reading from a CSV File with pandas

From this point on, this section uses `pandas` to work with the Apple stock price data set. The major function used is `read_csv()` which allows for a number of customizations via different parameters (see the `read_csv()` API reference). `read_csv()` yields as a result of the data reading procedure a `DataFrame` object which is the central means of storing (tabular) data with `pandas`. The `DataFrame` class has many powerful methods that are particularly helpful in financial applications (refer to the `DataFrame` API reference).

```
In [11]: import pandas as pd  ❶

In [12]: data = pd.read_csv(fn, index_col=0,
                            parse_dates=True)  ❷

In [13]: data.info()  ❸
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 21 entries, 2020-04-01 to 2020-04-30
         Data columns (total 6 columns):
```

```
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   HIGH    21 non-null     float64
 1   CLOSE   21 non-null     float64
 2   LOW     21 non-null     float64
 3   OPEN    21 non-null     float64
 4   COUNT   21 non-null     float64
 5   VOLUME  21 non-null     float64
dtypes: float64(6)
memory usage: 1.1 KB

In [14]: data.tail()  ❹
Out[14]:                HIGH   CLOSE     LOW    OPEN     COUNT
VOLUME
         Date
         2020-04-24  283.01  282.97  277.00  277.20  306176.0
31627183.0
         2020-04-27  284.54  283.17  279.95  281.80  300771.0
29271893.0
         2020-04-28  285.83  278.58  278.20  285.08  285384.0
28001187.0
         2020-04-29  289.67  287.73  283.89  284.73  324890.0
34320204.0
         2020-04-30  294.53  293.80  288.35  289.96  471129.0
45765968.0
```

❶ The `pandas` package is imported.

❷ This imports the data from the CSV file, indicated that the first column shall be treated as the index column and letting the entries in that column be interpreted as date-time information.

❸

This method call prints out meta information regarding the resulting `DataFrame` object.

4. The `data.tail()` method prints out by default the five most recent data rows.

Calculating the mean of the Apple stock closing values now is only a single method call.

```
In [15]: data['CLOSE'].mean()
Out[15]: 272.38619047619056
```

[Link to Come] introduces more functionality of `pandas` for the handling of financial data. For details on working with `pandas` and the powerful `DataFrame` class also refer to the official `pandas` Documentation page and to the book McKinney (2017).

> **TIP**
>
> Although the Python standard library provides capabilities to read data from CSV files, `pandas` in general significantly simplifies and speeds up such operations. An additional benefit is that the data analysis capabilities of `pandas` are immediately available since `read_csv()` returns a `DataFrame` object.

Exporting to Excel and JSON

`pandas` is also strong in exporting data stored in `DataFrame` objects when this data needs to be shared in a non-Python specific format. Apart from being able to export to CSV files, `pandas` allows, for

example, also to do the export in the form of Excel spreadsheet files as well as JSON files which are both popular data exchange formats in the financial industry. Such an exporting procedure typically needs a single method call only.

```
In [16]: data.to_excel('data/aapl.xls', 'AAPL')  ❶

In [17]: data.to_json('data/aapl.json')  ❷

In [18]: ls -n data/
         total 24
         -rw-r--r--  1 501  20  3067 May  7 11:15 aapl.json
         -rw-r--r--  1 501  20  5632 May  7 11:15 aapl.xls
```

❶ Exports the data to an Excel spreadsheet file on disk.

❷ Exports the data to a JSON file on disk.

In particular when it comes to the interaction with Excel spreadsheet files, there are more elegant ways than just doing a data dump to a new file. `xlwings`, for example, is a powerful Python package allowing for an efficient and intelligent interaction between Python and Excel (visit the `xlwings` home page).

Reading from Excel and JSON

Now that the data is also available in the form of an Excel spreadsheet file and a JSON data file, `pandas` can read data from

these sources as well. The approach is as straightforward as with CSV files.

```
In [19]: data_copy_1 = pd.read_excel('data/aapl.xls', 'AAPL',
                                     index_col=0)  ❶

In [20]: data_copy_1.head()  ❷
Out[20]:                HIGH    CLOSE       LOW     OPEN    COUNT
VOLUME
         Date
         2020-04-01  248.72  240.91  239.1300  246.50  460606
44054638
         2020-04-02  245.15  244.93  236.9000  240.34  380294
41483493
         2020-04-03  245.70  241.41  238.9741  242.80  293699
32470017
         2020-04-06  263.11  262.47  249.3800  250.90  486681
50455071
         2020-04-07  271.70  259.43  259.0000  270.80  467375
50721831

In [21]: data_copy_2 = pd.read_json('data/aapl.json')  ❸

In [22]: data_copy_2.head()  ❹
Out[22]:                HIGH    CLOSE       LOW     OPEN    COUNT
VOLUME
         2020-04-01  248.72  240.91  239.1300  246.50  460606
44054638
         2020-04-02  245.15  244.93  236.9000  240.34  380294
41483493
         2020-04-03  245.70  241.41  238.9741  242.80  293699
32470017
         2020-04-06  263.11  262.47  249.3800  250.90  486681
50455071
         2020-04-07  271.70  259.43  259.0000  270.80  467375
```

```
50721831

In [23]: !rm data/*
```

❶ This reads the data from the Excel spreadsheet file to a new `DataFrame` object.

❷ The first five rows of the first in-memory copy of the data are printed.

❸ This reads the data from the JSON file to yet another `DataFrame` object.

❹ This then prints the first five rows of the second in-memory copy of the data.

`pandas` proves useful for reading and writing financial data from and to different types of data files. Often, the reading might be tricky due to non-standard storage formats (like a ";" instead of a "," as separator) but `pandas` generally provides the right set of parameter combinations to cope with such cases. Although all examples in this section use a small data set only, one can expect high performance input-output operations from `pandas` in the most important scenarios when the data sets are much larger.

Working with Open Data Sources

To a great extent, the attractiveness of the Python ecosystem stems from the fact that almost all packages available are open source and can be used for free. Financial analytics in general and algorithmic trading in particular, however, cannot live with open source software and algorithms alone — data plays a vital role as well, as the quote at the beginning of the chapter emphasizes. The previous section uses a small data set from a commercial data source. While there have been helpful open (financial) data sources available for some years (such as the ones provided by Yahoo! Finance or Google Finance), there are not too many left at the time of this writing in 2020. One of the more obvious reasons for this trend might be the ever-changing terms of data licensing agreements.

The one notable exception for the purposes of this book is **Quandl** (*http://quandl.com*), a platform that aggregates a large number of open as well as premium (= to-be-paid-for) data sources. The data is provided via a unified API for which a Python wrapper package is available.

The Python wrapper package for the Quandl data API (see the Python wrapper page on Quandl and the Github page of the package) is installed with `conda` through `conda install quandl`. The first example shows how to retrieve historical average prices for the BTC/USD exchange rate since the introduction of Bitcoin as a cryptocurrency. With Quandl, requests expect always a combination of the *database* and the specific *data set* desired. In the example, `BCHAIN` and `MKPRU`. Such information can generally be looked up on the Quandl platform. For the example, the relevant page on Quandl is BCHAIN/MKPRU.

By default, the `quandl` package returns a `pandas` `DataFrame` object. In the example, the `Value` column is also presented in annualized fashion, i.e. with year end values. Note that the number shown for 2020 is the last available value in the data set (from May 2020) and not necessarily the year end value.

While a large part of the data sets on the Quandl platform are free, some of the free data sets require an API key (such a key is required after a certain limit of free API calls too). Every user obtains such a key by signing up for a free Quandl account on the Quandl sign up page. Data requests requiring an API key expect the key to be provided as the parameter `api_key`. In the example, the API key (which is found on the account settings page) is stored as a string in the variable `quandl_api_key`. The concrete value for the key is read from a configuration file via the `configparser` module.

```
In [24]: import configparser
         config = configparser.ConfigParser()
         config.read('../pyalgo.cfg')
Out[24]: ['../pyalgo.cfg']

In [25]: import quandl as q  ❶

In [26]: data = q.get('BCHAIN/MKPRU', api_key=config['quandl']
['api_key'])  ❷

In [27]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 4144 entries, 2009-01-03 to 2020-05-08
         Data columns (total 1 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
```

```
 0   Value   4144 non-null   float64
dtypes: float64(1)
memory usage: 64.8 KB

In [28]: data['Value'].resample('A').last()  ❸
Out[28]: Date
         2009-12-31        0.000000
         2010-12-31        0.299999
         2011-12-31        4.995000
         2012-12-31       13.590000
         2013-12-31      731.000000
         2014-12-31      317.400000
         2015-12-31      428.000000
         2016-12-31      952.150000
         2017-12-31    13215.574000
         2018-12-31     3832.921667
         2019-12-31     7385.360000
         2020-12-31     9170.790000
         Freq: A-DEC, Name: Value, dtype: float64
```

❶ Imports the Python wrapper package for Quandl.

❷ Reads historical data for the BTC/USD exchange rate.

❸ Selects the `Value` column, resamples it to yearly values and defines the last available observation to be the relevant one.

Quandl also provides, for example, diverse data sets for single stocks, like end-of-day stock prices, stock fundamentals or data sets related to options traded on a certain stock.

```
In [29]: data = q.get('FSE/SAP_X', start_date='2018-1-1',
                          end_date='2020-05-01',
                          api_key=config['quandl']['api_key'])

In [30]: data.info()
          <class 'pandas.core.frame.DataFrame'>
          DatetimeIndex: 579 entries, 2018-01-02 to 2020-04-30
          Data columns (total 10 columns):
           #   Column                 Non-Null Count  Dtype
          ---  ------                     --------------   -----
           0   Open                       257 non-null
float64
           1   High                       579 non-null
float64
           2   Low                        579 non-null
float64
           3   Close                      579 non-null
float64
           4   Change                     0 non-null       object
           5   Traded Volume             533 non-null     float64
           6   Turnover                   533 non-null     float64
           7   Last Price of the Day   0 non-null       object
           8   Daily Traded Units      0 non-null       object
           9   Daily Turnover          0 non-null       object
          dtypes: float64(6), object(4)
          memory usage: 49.8+ KB
```

The API key can also be configured permanently with the Python wrapper via

```
q.ApiConfig.api_key = 'YOUR_API_KEY'
```

The Quandl platform also offers premium data sets for which a subscription or fee is required. Most of these data sets offer free samples. The example retrieves option implied volatilities for the Microsoft, Inc. stock. The free sample data set is quite large with more than 4,100 rows and many columns (only a subset is shown). The last line of code display the 30, 60 and 90 days implied volatility values for the five most recent days available.

```
In [31]: q.ApiConfig.api_key = config['quandl']['api_key']

In [32]: vol = q.get('VOL/MSFT')

In [33]: vol.iloc[:, :10].info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 1006 entries, 2015-01-02 to 2018-12-31
         Data columns (total 10 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   Hv10    1006 non-null   float64
          1   Hv20    1006 non-null   float64
          2   Hv30    1006 non-null   float64
          3   Hv60    1006 non-null   float64
          4   Hv90    1006 non-null   float64
          5   Hv120   1006 non-null   float64
          6   Hv150   1006 non-null   float64
          7   Hv180   1006 non-null   float64
          8   Phv10   1006 non-null   float64
          9   Phv20   1006 non-null   float64
         dtypes: float64(10)
         memory usage: 86.5 KB

In [34]: vol[['IvMean30', 'IvMean60', 'IvMean90']].tail()
Out[34]:              IvMean30  IvMean60  IvMean90
         Date
```

```
2018-12-24    0.4310    0.4112    0.3829
2018-12-26    0.4059    0.3844    0.3587
2018-12-27    0.3918    0.3879    0.3618
2018-12-28    0.3940    0.3736    0.3482
2018-12-31    0.3760    0.3519    0.3310
```

This concludes the overview of the Python wrapper package `quandl` for the Quandl data API. The Quandl platform and service is growing rapidly and proves to be a valuable source for financial data in an algorithmic trading context.

> **NOTE**
>
> Open source software is a trend that started many years ago. It has lowered the barriers of entry in many areas and also in algorithmic trading. A new, reinforcing trend in this regard are open data sources. In some cases, such as with Quandl, they even provide high quality data sets. It cannot be expected that open data will completely replace professional data subscriptions any time soon, but they represent a valuable means to get started with algorithmic trading in a cost efficient manner.

Eikon Data API

Open data sources are a blessing for algorithmic traders wanting to get started in the space and wanting to be able to quickly test hypotheses and ideas based on real financial data sets. Sooner or later, however, open data sets will not suffice anymore to satisfy the requirements of more ambitious traders and professionals.

Refinitiv is one of the biggest financial data and news providers in the world. Its current desktop flagship product is Eikon which is the equivalent to the Terminal by Bloomberg, the major competitor in the data services field. Figure 3-1 shows a screen shot of Eikon in the browser-based version. It provides access to peta bytes of data via a single access point.

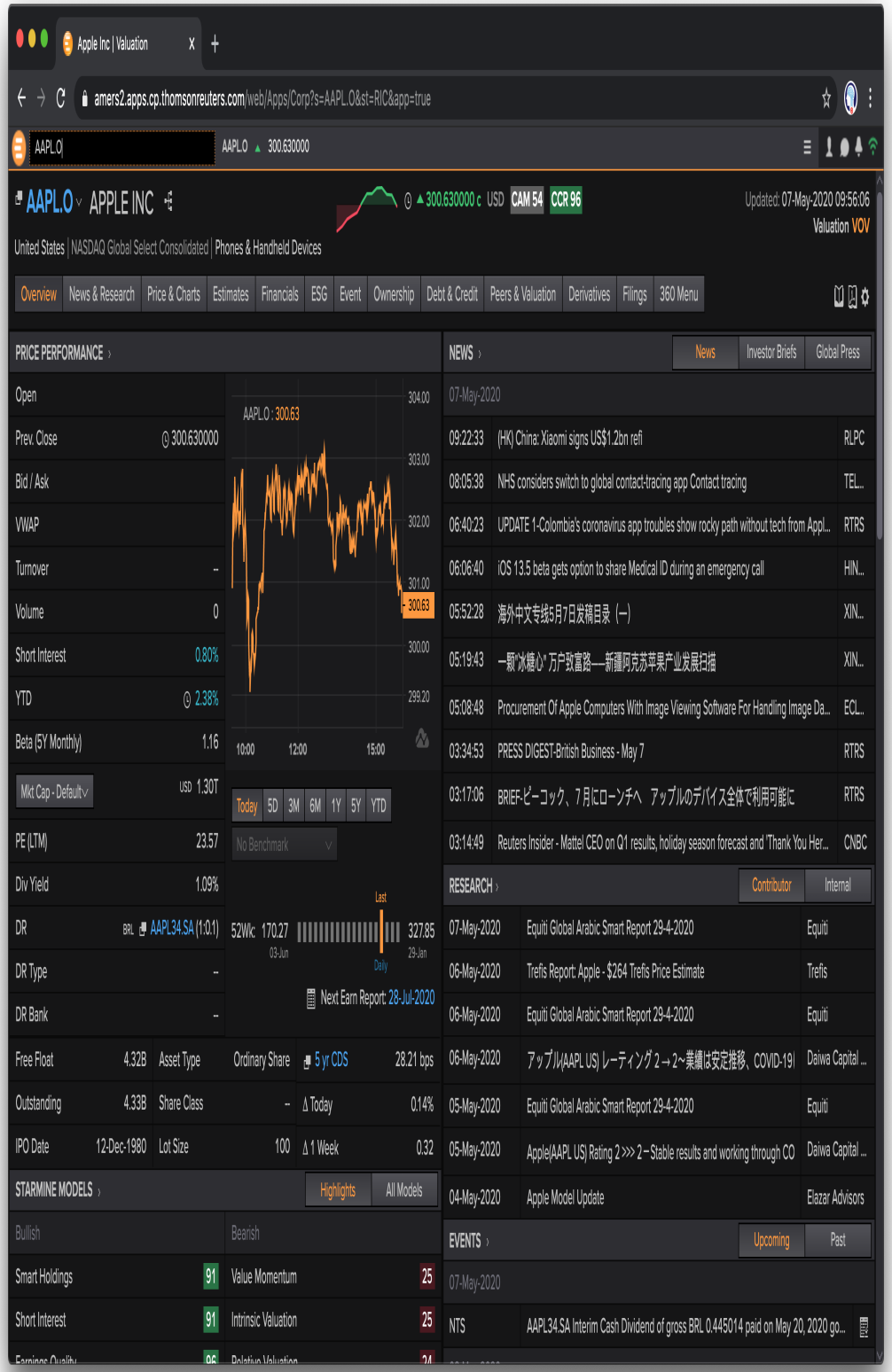
Apple Inc | Valuation
amers2.apps.cp.thomsonreuters.com/web/Apps/Corp?s=AAPL.O&st=RIC&app=true
AAPL.O
AAPL.O ▲ 300.630000
AAPL.O APPLE INC
▲ 300.630000 USD CAM 54 CCR 96
Updated: 07-May-2020 09:56:06
Valuation VOV
United States | NASDAQ Global Select Consolidated | Phones & Handheld Devices
Overview News & Research Price & Charts Estimates Financials ESG Event Ownership Debt & Credit Peers & Valuation Derivatives Filings 360 Menu
PRICE PERFORMANCE
Open
Prev. Close 300.630000
Bid / Ask
VWAP
Turnover -
Volume 0
Short Interest 0.80%
YTD 2.38%
Beta (5Y Monthly) 1.16
Mkt Cap - Default USD 1.30T
PE (LTM) 23.57
Div Yield 1.09%
DR BRL AAPL34.SA (1:0.1)
DR Type -
DR Bank -
AAPL.O : 300.63
304.00
303.00
302.00
301.00
300.63
300.00
299.20
10:00
12:00
15:00
Today 5D 3M 6M 1Y 5Y YTD
No Benchmark
Last
52Wk: 170.27
327.85
03-Jun
29-Jan
Daily
Next Earn Report: 28-Jul-2020
Free Float 4.32B
Asset Type Ordinary Share
5 yr CDS 28.21 bps
Outstanding 4.33B
Share Class -
Δ Today 0.14%
IPO Date 12-Dec-1980
Lot Size 100
Δ 1 Week 0.32
STARMINE MODELS
Highlights All Models
Bullish
Bearish
Smart Holdings 91
Value Momentum 25
Short Interest 91
Intrinsic Valuation 25
NEWS
News Investor Briefs Global Press
07-May-2020
09:22:33 (HK) China: Xiaomi signs US$1.2bn refi RLPC
08:05:38 NHS considers switch to global contact-tracing app Contact tracing TEL...
06:40:23 UPDATE 1-Colombia's coronavirus app troubles show rocky path without tech from Appl... RTRS
06:06:40 iOS 13.5 beta gets option to share Medical ID during an emergency call HIN...
05:52:28 海外中文专线5月7日发稿目录（一） XIN...
05:19:43 一颗"冰糖心"万户致富路——新疆阿克苏苹果产业发展扫描 XIN...
05:08:48 Procurement Of Apple Computers With Image Viewing Software For Handling Image Da... ECL...
03:34:53 PRESS DIGEST-British Business - May 7 RTRS
03:17:06 BRIEF-ピーコック、7月にローンチへ アップルのデバイス全体で利用可能に RTRS
03:14:49 Reuters Insider - Mattel CEO on Q1 results, holiday season forecast and 'Thank You Her... CNBC
RESEARCH
Contributor Internal
07-May-2020 Equiti Global Arabic Smart Report 29-4-2020 Equiti
06-May-2020 Trefis Report: Apple - $264 Trefis Price Estimate Trefis
06-May-2020 Equiti Global Arabic Smart Report 29-4-2020 Equiti
06-May-2020 アップル(AAPL US) レーティング 2→2〜業績は安定推移、COVID-19| Daiwa Capital ...
05-May-2020 Equiti Global Arabic Smart Report 29-4-2020 Equiti
05-May-2020 Apple(AAPL US) Rating 2 >>> 2 – Stable results and working through CO Daiwa Capital ...
04-May-2020 Apple Model Update Elazar Advisors
EVENTS
Upcoming Past
07-May-2020
NTS AAPL34.SA Interim Cash Dividend of gross BRL 0.445014 paid on May 20, 2020 go...

Figure 3-1. Browser version of Eikon terminal

Recently, Refinitiv have streamlined their API landscape and have released a Python wrapper package, called `eikon`, for the Eikon data API which is installed via `pip install eikon`. If you have a subscription to the Refinitiv Eikon data services, you can use the Python package to programmatically retrieve historical as well as streaming structured and unstructured data from the unified API. A technical prerequisite is that a local desktop application is running that provides a desktop API session. The latest such desktop application at the time of this writing is called Workspace (see Figure 3-2).


AAPLO, .SPX, BTC= CHT
MARKETS
NEWS
CHARTING
COMPANY
ANALYTICS
PORTFOLIOS & MONITORS
SEARCH TOOLS
MY LAYOUTS
INSTRUMENTS 3/20
Clear All
AAPLO
.SPX
BTC=
Add Instrument
PRICE SERIES
Rebasing
Base Value: 0 100
OPTIONS
Templates
Daily
Reset
APPLE INC 1D 300.63
S&P 500 INDEX 2848.42
FOREX BTC= 9253.74
128.00
124.00
120.00
116.76
112.00
108.00
104.00
99.45
96.00
92.58
88.00
84.00
80.00
76.00
72.00
68.00
64.00
60.00
56.00
52.00
48.00
15
Dec
16
2020
16
Feb
Mar
16
Apr
16
07 May 20
1D
5D
10D
1M
3M
6M
YTD
1Y
2Y
3Y
5Y
10Y
20Y
Max
06-Nov-2019 - 06-May-2020

Figure 3-2. Workspace application with desktop API services

If you are an Eikon subscriber and have an account for the Developer Community pages, you find an overview of the Python Eikon Scripting Library under Quick Start.

In order to use the Eikon Data API, the Eikon `app_key` needs to be set. You get it via the App Key Generator (APPKEY`) application in either Eikon or Workspace.

```
In [35]: import eikon as ek  ❶

In [36]: ek.set_app_key(config['eikon']['app_key'])  ❷
         2020-05-07 11:15:56,323 P[87076] [MainThread 4326989248]
Error on
          handshake port 9000 : ReadTimeout(ReadTimeout())

In [37]: help(ek)  ❸
         Help on package eikon:

         NAME
             eikon - # coding: utf-8

         PACKAGE CONTENTS
             Profile
             data_grid
             eikonError
             json_requests
             news_request
             streaming_session (package)
             symbology
             time_series
             tools
```

```
            SUBMODULES
                cache
                desktop_session
                istream_callback
                itemstream
                session
                stream
                stream_connection
                streamingprice
                streamingprice_callback
                streamingprices

            VERSION
                1.1.2

            FILE
                /Users/yves/Python/lib/python3.7/site-
packages/eikon/__init__.py
```

❶ Imports the `eikon` package as `ek`.

❷ Sets the `app_key`.

❸ Shows the help text for the main module.

Retrieving Historical Structured Data

The retrieval of historical financial time series data is as straightforward as with the other wrappers used before.

```
In [39]: symbols = ['AAPL.O', 'MSFT.O', 'GOOG.O']  ❶
```

```
In [40]: data = ek.get_timeseries(symbols,  ❷
                                  start_date='2020-01-01',  ❸
                                  end_date='2020-05-01',  ❹
                                  interval='daily',  ❺
                                  fields=['*'])  ❻

In [41]: data.keys()  ❼
Out[41]: MultiIndex([('AAPL.O',   'HIGH'),
                     ('AAPL.O',  'CLOSE'),
                     ('AAPL.O',    'LOW'),
                     ('AAPL.O',   'OPEN'),
                     ('AAPL.O',  'COUNT'),
                     ('AAPL.O', 'VOLUME'),
                     ('MSFT.O',   'HIGH'),
                     ('MSFT.O',  'CLOSE'),
                     ('MSFT.O',    'LOW'),
                     ('MSFT.O',   'OPEN'),
                     ('MSFT.O',  'COUNT'),
                     ('MSFT.O', 'VOLUME'),
                     ('GOOG.O',   'HIGH'),
                     ('GOOG.O',  'CLOSE'),
                     ('GOOG.O',    'LOW'),
                     ('GOOG.O',   'OPEN'),
                     ('GOOG.O',  'COUNT'),
                     ('GOOG.O', 'VOLUME')],
                    names=['Security', 'Field'])

In [42]: type(data['AAPL.O'])  ❽
Out[42]: pandas.core.frame.DataFrame

In [43]: data['AAPL.O'].info()  ❾
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 84 entries, 2020-01-02 to 2020-05-01
         Data columns (total 6 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
```

```
 0   HIGH    84 non-null     float64
 1   CLOSE   84 non-null     float64
 2   LOW     84 non-null     float64
 3   OPEN    84 non-null     float64
 4   COUNT   84 non-null     float64
 5   VOLUME  84 non-null     float64
dtypes: float64(6)
memory usage: 4.6 KB

In [44]: data['AAPL.O'].tail()  ❿
Out[44]: Field         HIGH   CLOSE     LOW    OPEN     COUNT
VOLUME
         Date
         2020-04-27  284.54  283.17  279.95  281.80  300771.0
29271893.0
         2020-04-28  285.83  278.58  278.20  285.08  285384.0
28001187.0
         2020-04-29  289.67  287.73  283.89  284.73  324890.0
34320204.0
         2020-04-30  294.53  293.80  288.35  289.96  471129.0
45765968.0
         2020-05-01  299.00  289.07  285.85  286.25  558319.0
60154175.0
```

❶ Defines a few symbols as a `list` object.

❷ The central line of code that retrieves data for the first symbol …

❸ … for the given start date and …

❹ … the given end date.

❺

The time interval is here chosen to be `daily`.

❻ All fields are requested.

❼ The function `get_timeseries()` returns a multi-index `DataFrame` object.

❽ The values corresponding to each level are regular `DataFrame` objects.

❾ This provides an overview of the data stored in the `DataFrame` object.

❿ The final five rows of data are shown.

The beauty of working with a professional data service API becomes evident when one wishes to work with multiple symbols and in particular with a different granularity of the financial data, i.e. other time intervals.

```
In [45]: %%time
         data = ek.get_timeseries(symbols,  ❶
                                  start_date='2020-05-05',  ❷
                                  end_date='2020-05-06',  ❸
                                  interval='minute',  ❹
                                  fields='*')
         CPU times: user 44.2 ms, sys: 2.87 ms, total: 47.1 ms
         Wall time: 10.7 s

In [46]: print(data['GOOG.O'].loc['2020-05-05 16:00:00':
```

```
                                    '2020-05-05
16:04:00'].round(1))  ❺
         Field                   HIGH     LOW    OPEN   CLOSE
COUNT  VOLUME
         Date
         2020-05-05 16:00:00  1367.4  1366.1  1366.4  1367.2
82.0  3308.0
         2020-05-05 16:01:00  1369.0  1367.2  1367.4  1368.8
210.0  5455.0
         2020-05-05 16:02:00  1369.5  1368.4  1368.8  1368.9
176.0  4900.0
         2020-05-05 16:03:00  1369.8  1368.4  1369.2  1368.4
162.0  6133.0
         2020-05-05 16:04:00  1369.3  1367.9  1368.4  1368.6
140.0  5080.0

In [47]: for sym in symbols:
             print('\n' + sym + '\n',
data[sym].iloc[-300:-295].round(1))  ❻

         AAPL.O
          Field                   HIGH    LOW   OPEN  CLOSE  COUNT
VOLUME
         Date
         2020-05-05 19:01:00  300.9  300.7  300.9  300.7  917.0
88059.0
         2020-05-05 19:02:00  300.9  300.7  300.7  300.8  634.0
60998.0
         2020-05-05 19:03:00  300.8  300.6  300.8  300.7  733.0
75349.0
         2020-05-05 19:04:00  300.8  300.6  300.7  300.8  632.0
56220.0
         2020-05-05 19:05:00  300.9  300.8  300.8  300.9  522.0
51194.0

         MSFT.O
```

```
          Field                HIGH    LOW   OPEN  CLOSE  COUNT
VOLUME
         Date
         2020-05-05 19:01:00  183.6  183.5  183.6  183.5  679.0
73198.0
         2020-05-05 19:02:00  183.6  183.5  183.5  183.6  595.0
79593.0
         2020-05-05 19:03:00  183.6  183.5  183.6  183.6  505.0
61200.0
         2020-05-05 19:04:00  183.6  183.5  183.6  183.6  357.0
27781.0
         2020-05-05 19:05:00  183.6  183.6  183.6  183.6  592.0
54057.0

         GOOG.O
          Field                 HIGH     LOW    OPEN   CLOSE
COUNT  VOLUME
         Date
         2020-05-05 19:01:00  1369.8  1368.7  1369.5  1369.0
158.0  5056.0
         2020-05-05 19:02:00  1369.6  1368.7  1368.9  1368.9
32.0   950.0
         2020-05-05 19:03:00  1369.5  1367.9  1369.3  1368.4
174.0  5186.0
         2020-05-05 19:04:00  1369.2  1368.0  1368.8  1368.7
76.0  2809.0
         2020-05-05 19:05:00  1369.6  1368.7  1368.7  1369.0
46.0  1186.0
```

❶ Data is retrieved for all symbols at once.

❷ The time interval …

❸

… is drastically shortened.

❹ The function call retrieves minute bars for the symbols.

❺ Prints five rows from the Google, Inc. data set.

❻ Prints three data rows from every `DataFrame` object.

The code above illustrates how convenient it is to retrieve historical financial time series data from the Eikon API with Python. By default, the function `get_timeseries` provides the following options for the `interval` parameter: `tick`, `minute`, `hour`, `daily`, `weekly`, `monthly`, `quarterly` and `yearly`. This gives all the flexibility needed in an algorithmic trading context — in particular, when combined with the resampling capabilities of `pandas` as shown in the code below.

```
In [48]: %%time
         data = ek.get_timeseries(symbols[0],
                                  start_date='2020-05-05
15:00:00',  ❶
                                  end_date='2020-05-05 16:00:00',
❷
                                  interval='tick',  ❸
                                  fields=['*'])
         CPU times: user 219 ms, sys: 16.1 ms, total: 235 ms
         Wall time: 7.34 s

In [49]: data.info()  ❹
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 39869 entries, 2020-05-05 15:00:00.012000
```

```
to 2020-05-05
          15:59:59.973000
         Data columns (total 2 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   VALUE   39818 non-null  float64
          1   VOLUME  39869 non-null  float64
         dtypes: float64(2)
         memory usage: 934.4 KB

In [50]: data.head()  ❺
Out[50]: AAPL.O                     VALUE  VOLUME
         Date
         2020-05-05 15:00:00.012  298.05   100.0
         2020-05-05 15:00:00.013  298.05   100.0
         2020-05-05 15:00:00.013  298.05    60.0
         2020-05-05 15:00:00.013  298.05    40.0
         2020-05-05 15:00:00.013  298.05    56.0

In [51]: resampled = data.resample('30s', label='right').agg(
                     {'VALUE': 'last', 'VOLUME': 'sum'}) ❻

In [52]: resampled.tail()  ❼
Out[52]:                          VALUE   VOLUME
         Date
         2020-05-05 15:58:00  298.8750  11129.0
         2020-05-05 15:58:30  298.9601  19818.0
         2020-05-05 15:59:00  298.9000  18750.0
         2020-05-05 15:59:30  298.9700  28733.0
         2020-05-05 16:00:00  298.8900  24757.0
```

❶ A time interval of …

❷ … one hour is chosen (due to data retrieval limits).

❸ The `interval` parameter is set to `tick`.

❹ More than 20,000 price ticks are retrieved for the interval.

❺ The time series data set shows highly irregular (heterogeneous) interval lengths between two ticks.

❻ The tick data is resampled to a 30 second interval length (by taking the last value and the sum, respectively) …

❼ … which is reflected in the `DatetimeIndex` of the new `DataFrame` object.

Retrieving Historical Unstructured Data

A major strength of working with the Eikon API via Python is the easy retrieval of unstructured data — which can then be parsed and analyzed with Python packages for natural language processing (NLP). Such a procedure is as simple and straightforward as for financial time series data. The code that follows retrieves news headlines for a fixed time interval which includes Apple, Inc. as a company as well as "iPhone" as a word. The five most recent hits are displayed as a maximum.

```
In [53]: headlines = ek.get_news_headlines(query='R:AAPL.O
macbook',  ❶
                                           count=5,  ❷
                                           date_from='2020-4-1',
❸
```

```
                                                date_to='2020-5-1')  ❹

In [54]: headlines  ❺
Out[54]:                                                versionCreated
\
         2020-04-20 21:33:37.332 2020-04-20 21:33:37.332000+00:00
         2020-04-20 10:20:23.201 2020-04-20 10:20:23.201000+00:00
         2020-04-20 02:32:27.721 2020-04-20 02:32:27.721000+00:00
         2020-04-15 12:06:58.693 2020-04-15 12:06:58.693000+00:00
         2020-04-09 21:34:08.671 2020-04-09 21:34:08.671000+00:00


text  \
         2020-04-20 21:33:37.332  Apple said to launch new
AirPods, MacBook Pro ...
         2020-04-20 10:20:23.201  Apple might launch upgraded
AirPods, 13-inch M...
         2020-04-20 02:32:27.721  Apple to reportedly launch new
AirPods alongsi...
         2020-04-15 12:06:58.693  Apple files a patent for
iPhones, MacBook indu...
         2020-04-09 21:34:08.671  Apple rolls out new software
update for MacBoo...


storyId  \
         2020-04-20 21:33:37.332
urn:newsml:reuters.com:20200420:nNRAble9rq:1
         2020-04-20 10:20:23.201
urn:newsml:reuters.com:20200420:nNRAbl8eob:1
         2020-04-20 02:32:27.721
urn:newsml:reuters.com:20200420:nNRAbl4mfz:1
         2020-04-15 12:06:58.693
urn:newsml:reuters.com:20200415:nNRAbjvsix:1
         2020-04-09 21:34:08.671
urn:newsml:reuters.com:20200409:nNRAbi2nbb:1
```

```
                                sourceCode
          2020-04-20 21:33:37.332  NS:TIMIND
          2020-04-20 10:20:23.201  NS:BUSSTA
          2020-04-20 02:32:27.721  NS:HINDUT
          2020-04-15 12:06:58.693  NS:HINDUT
          2020-04-09 21:34:08.671  NS:TIMIND

In [55]: story = headlines.iloc[0]  ❻

In [56]: story  ❼
Out[56]: versionCreated                    2020-04-20
21:33:37.332000+00:00
         text              Apple said to launch new AirPods,
MacBook Pro ...
         storyId
urn:newsml:reuters.com:20200420:nNRAble9rq:1
         sourceCode
NS:TIMIND
         Name: 2020-04-20 21:33:37.332000, dtype: object

In [57]: news_text = ek.get_news_story(story['storyId'])  ❽

In [58]: from IPython.display import HTML  ❾

In [59]: HTML(news_text)  ❿
Out[59]: <IPython.core.display.HTML object>
```

NEW DELHI: Apple recently launched its much-awaited affordable smartphone
iPhone SE. Now it seems that the company is gearing up for another launch.
Apple is said to launch the next generation of AirPods and the all-new
13-inch MacBook Pro next month.

In February an online report revealed that the Cupertino-based tech giant is working on AirPods Pro Lite. Now a tweet by tipster Job Posser has revealed that Apple will soon come up with new AirPods and MacBook Pro.
Jon Posser tweeted, "New AirPods (which were supposed to be at the March Event) is now ready to go.

Probably alongside the MacBook Pro next month." However, not many details about the upcoming products are available right now. The company was supposed to launch these products at the March event along with the iPhone SE.

But due to the ongoing pandemic coronavirus, the event got cancelled.
It is expected that Apple will launch the AirPods Pro Lite and the 13-inch MacBook Pro just like the way it launched the iPhone SE. Meanwhile, Apple has scheduled its annual developer conference WWDC to take place in June.

This year the company has decided to hold an online-only event due to the outbreak of coronavirus. Reports suggest that this year the company is planning to launch the all-new AirTags and a premium pair of over-ear Bluetooth headphones at the event. Using the Apple AirTags users will be able to locate real-world items such as keys or suitcase in the Find My app.

```
The AirTags will also have offline finding capabilities
that the company
introduced in the core of iOS 13.Apart from this, Apple
is also said to
unveil its high-end Bluetooth headphones. It is
expected that the Bluetooth
headphones will offer better sound quality and battery
backup as compared
to the AirPods.

For Reprint Rights: timescontent.com

Copyright (c) 2020 BENNETT,COLEMAN & CO.LTD.
```

❶ The `query` parameter for the retrieval operation.

❷ Sets the maximum number of hits to five.

❸ Defines the interval …

❹ … for which to look for news headlines.

❺ Gives out the results object (output shortened).

❻ One particular headline is picked …

❼ … and the `story_id` shown.

❽ This retrieves the news text as html code.

❾ In `Jupyter Notebook`, for example, the html code …


… can be rendered for better reading.

This concludes the illustration of the Python wrapper package for the Refinitiv Eikon data API.

Storing Financial Data Efficiently

In algorithmic trading, one of the most important scenarios for the management of data sets is “retrieve once, use multiple times”. Or from an input-output (IO) perspective, it is “write once, read multiple times”. In the first case, data might be retrieved from a web service and then used to backtest a strategy multiple times based on a temporary, in-memory copy of the data set. In the second case, tick data that is received continually is written to disk and later on again used multiple times for certain manipulations (like aggregations) in combination with a backtesting procedure.

This section assumes that the in-memory data structure to store the data is a `pandas` `DataFrame` object, no matter from which source the data is acquired (from a CSV file, a web service, etc.).

To have a somewhat meaningful data set available in terms of size, the section uses a sample financial data set generated by the use of pseudo-random numbers. “Python Scripts” presents the Python module with a function called `generate_sample_data` that accomplishes the task.

In principle, this function allows to generate a sample financial data set in tabular form of arbitrary size (available memory of course sets a limit).

```
In [60]: from sample_data import generate_sample_data  ❶

In [61]: print(generate_sample_data(rows=5, cols=4))  ❷
                                    No0         No1         No2
No3
         2021-01-01 00:00:00  100.000000  100.000000  100.000000
100.000000
         2021-01-01 00:01:00   99.965918  100.065050  100.063784
100.016058
         2021-01-01 00:02:00   99.896126   99.853106  100.051909
99.985001
         2021-01-01 00:03:00   99.893056   99.856924   99.959842
100.080701
         2021-01-01 00:04:00   99.954987   99.778804   99.927796
100.034123
```

❶ Imports the function from the Python script.

❷ Prints a sample financial data set with five rows and four columns.

Storing DataFrame Objects

The storage of a `pandas` `DataFrame` object as a whole is made simple by the `pandas` `HDFStore` wrapper functionality for the HDF5 binary storage standard. It allows to dump complete `DataFrame` objects in a single step to a file-based database object. To illustrate the

implementation, the first step is to create a sample data set of meaningful size — here the size of the `DataFrame` generated is about 420 MB.

```
In [62]: %time data = generate_sample_data(rows=5e6,
cols=10).round(4)  ❶
         CPU times: user 4.11 s, sys: 873 ms, total: 4.99 s
         Wall time: 5.02 s

In [63]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 5000000 entries, 2021-01-01 00:00:00 to
2030-07-05
          05:19:00
         Freq: T
         Data columns (total 10 columns):
          #   Column  Dtype
         ---  ------  -----
          0   No0     float64
          1   No1     float64
          2   No2     float64
          3   No3     float64
          4   No4     float64
          5   No5     float64
          6   No6     float64
          7   No7     float64
          8   No8     float64
          9   No9     float64
         dtypes: float64(10)
         memory usage: 419.6 MB
```

❶ A sample financial data set with 5,000,000 rows and ten columns is generated; the generation takes a couple of seconds.

The second step is to open a `HDFStore` object (i.e. HDF5 database file) on disk and to write the `DataFrame` object to it.[2] The size on disk of about 440 MB is a bit larger than for the in-memory `DataFrame` object. However, the writing speed is about five times faster than the in-memory generation of the sample data set. Working in Python with binary stores like HDF5 database files usually gets you writing speeds close to the theoretical maximum of the hardware available.[3]

```
In [64]: h5 = pd.HDFStore('data/data.h5', 'w')  ❶

In [65]: %time h5['data'] = data  ❷
         CPU times: user 313 ms, sys: 431 ms, total: 745 ms
         Wall time: 887 ms

In [66]: h5  ❸
Out[66]: <class 'pandas.io.pytables.HDFStore'>
         File path: data/data.h5

In [67]: ls -n data/data.*
         -rw-r--r--@ 1 501  20  440007240 May  7 11:16
data/data.h5

In [68]: h5.close()  ❹
```

❶ This opens the database file on disk for writing (and overwrites a potentially existing file with the same name).

❷ Writing the `DataFrame` object to disk takes less than a second.

❸

Print out meta information for the database file.

❹ Closes the database file.

The third step is to read the data from the file-based `HDFStore` object. Reading also generally takes place close to the theoretical maximum speed.

```
In [69]: h5 = pd.HDFStore('data/data.h5', 'r')  ❶

In [70]: %time data_copy = h5['data']  ❷
         CPU times: user 457 ms, sys: 420 ms, total: 876 ms
         Wall time: 881 ms

In [71]: data_copy.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 5000000 entries, 2021-01-01 00:00:00 to
2030-07-05
          05:19:00
         Freq: T
         Data columns (total 10 columns):
          #   Column  Dtype
         ---  ------  -----
          0   No0     float64
          1   No1     float64
          2   No2     float64
          3   No3     float64
          4   No4     float64
          5   No5     float64
          6   No6     float64
          7   No7     float64
          8   No8     float64
          9   No9     float64
         dtypes: float64(10)
```

```
        memory usage: 419.6 MB

In [72]: h5.close()

In [73]: rm data/data.h5
```

❶ Opens the database file for reading.

❷ Reading takes less than half of a second.

There is another, somewhat more flexible way of writing the data from a `DataFrame` object to an `HDFStore` object. To this end, one can use the `to_hdf()` method of the `DataFrame` object and sets the `format` parameter to `table` (see the `to_hdf` API reference page). This allows the appending of new data to the `table` object on disk and also, for example, the searching over the data on disk which is not possible with the first approach. The price to pay are slower writing and reading speeds.

```
In [74]: %time data.to_hdf('data/data.h5', 'data',
format='table')  ❶
         CPU times: user 3.65 s, sys: 576 ms, total: 4.22 s
         Wall time: 4.31 s

In [75]: ls -n data/data.*
         -rw-r--r--@ 1 501  20  446911683 May  7 11:16
data/data.h5

In [76]: %time data_copy = pd.read_hdf('data/data.h5', 'data')
❷
         CPU times: user 256 ms, sys: 309 ms, total: 565 ms
```

```
          Wall time: 565 ms

In [77]: data_copy.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 5000000 entries, 2021-01-01 00:00:00 to
2030-07-05
          05:19:00
         Freq: T
         Data columns (total 10 columns):
          #   Column  Dtype
         ---  ------  -----
          0   No0     float64
          1   No1     float64
          2   No2     float64
          3   No3     float64
          4   No4     float64
          5   No5     float64
          6   No6     float64
          7   No7     float64
          8   No8     float64
          9   No9     float64
         dtypes: float64(10)
         memory usage: 419.6 MB
```

❶ This defines the writing format to be of type `table`. Writing becomes slower since this format type involves a bit more overhead and leads to a somewhat increased file size.

❷ Reading is also slower in this application scenario.

In practice, the advantage of this approach is that one can work with the `table_frame` object on disk like with any other `table` object of

the `PyTables` package which is used by `pandas` in this context. This provides access to certain basic capabilities of the `PyTables` package — like, for instance, appending rows to a `table` object.

```
In [78]: import tables as tb  ❶

In [79]: h5 = tb.open_file('data/data.h5', 'r')  ❷

In [80]: h5  ❸
Out[80]: File(filename=data/data.h5, title='', mode='r',
root_uep='/',
          filters=Filters(complevel=0, shuffle=False,
bitshuffle=False,
          fletcher32=False, least_significant_digit=None))
         / (RootGroup) ''
         /data (Group) ''
         /data/table (Table(5000000,)) ''
           description := {
           "index": Int64Col(shape=(), dflt=0, pos=0),
           "values_block_0": Float64Col(shape=(10,), dflt=0.0,
pos=1)}
           byteorder := 'little'
           chunkshape := (2978,)
           autoindex := True
           colindexes := {
             "index": Index(6, medium, shuffle,
zlib(1)).is_csi=False}

In [81]: h5.root.data.table[:3]  ❹
Out[81]: array([(1609459200000000000, [100.    , 100.    , 100.
, 100.    ,
          100.    , 100.    , 100.    , 100.    , 100.    , 100.
]),
         (1609459260000000000, [ 99.9982,  99.9804, 100.1187,
99.9814,
```

```
          99.9845,  99.9969,  99.9546, 100.1686,  99.918 ,
99.9873]),
         (1609459320000000000, [ 99.9637, 100.0451, 100.0396,
99.8125,
          99.9327,  99.9475, 100.0082, 100.1155,  99.8387,
100.0953])],
               dtype=[('index', '<i8'), ('values_block_0', '<f8',
(10,))])

In [82]: h5.close()  ❺

In [83]: rm data/data.h5
```

❶ Imports the `PyTables` package.

❷ Opens the database file for reading.

❸ Shows the contents of the database file.

❹ Prints the first three rows in the table.

❺ Closes the database.

Although this second approach provides *more* flexibility, it does not open the doors to the full capabilities of the `PyTables` package. Nevertheless, the two approaches introduced in this sub-section are convenient and efficient when you are working with more or less *immutable data sets that fit into memory*. Nowadays, algorithmic trading, however, has to deal in general with continuously and rapidly growing data sets like, for example, tick data with regard to stock

prices or foreign exchange rates. To cope with the requirements of such a scenario, alternative approaches might prove useful.

> **TIP**
>
> Using the `HDFStore` wrapper for the HDF5 binary storage standard, `pandas` is able to write and read financial data almost at the maximum speed the available hardware allows. Exports to other file-based formats, like CSV, are generally much slower alternatives.

Using TsTables

The `PyTables` package — with import name `tables` — is a wrapper for the HDF5 binary storage library that is also used by `pandas` for its `HDFStore` implementation presented in the previous sub-section. The `TsTables` package (see Github page of the package) in turn is dedicated to the efficient handling of large financial time series data sets based on the HDF5 binary storage library. It is effectively an enhancement of the `PyTables` package and adds support for time series data to its capabilities. It implements a hierarchical storage approach that allows for a fast retrieval of data sub-sets selected by providing start and end dates and times, respectively. The major scenario supported by `TsTables` is "write once, retrieve multiple times".

The set up illustrated in this sub-section is that data is continuously collected from a web source, professional data provider, etc. and is stored interim and in-memory in a `DataFrame` object. After a while or a certain number of data points retrieved, the collected data is then

stored in a `TsTables table` object in a HDF5 database. First, the generation of the sample data.

```
In [84]: %%time
         data = generate_sample_data(rows=2.5e6, cols=5,
                                     freq='1s').round(4)  ❶
         CPU times: user 860 ms, sys: 190 ms, total: 1.05 s
         Wall time: 1.06 s

In [85]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2500000 entries, 2021-01-01 00:00:00 to
2021-01-29
          22:26:39
         Freq: S
         Data columns (total 5 columns):
          #   Column  Dtype
         ---  ------  -----
          0   No0     float64
          1   No1     float64
          2   No2     float64
          3   No3     float64
          4   No4     float64
         dtypes: float64(5)
         memory usage: 114.4 MB
```

❶ This generates a sample financial data set with 2,500,000 rows and five columns with a one second frequency; the sample data is rounded to two digits.

Second, some more imports and the creation of the `TsTables table` object. The major part is the definition of the `desc` class which

provides the description for the `table` object's data structure.

> **CAUTION**
>
> Currently, `TsTables` only works with the old `pandas` version 0.19. A friendly fork, working with newer versions of `pandas` is available under *http://github.com/yhilpisch/tstables* which can be installed via
>
> ```
> pip install
> git+https://github.com/yhilpisch/tstables.git
> ```

```
In [86]: import tstables  ❶

In [87]: import tables as tb  ❷

In [88]: class desc(tb.IsDescription):
             ''' Description of TsTables table structure.
             '''
             timestamp = tb.Int64Col(pos=0)  ❸
             No0 = tb.Float64Col(pos=1)  ❹
             No1 = tb.Float64Col(pos=2)
             No2 = tb.Float64Col(pos=3)
             No3 = tb.Float64Col(pos=4)
             No4 = tb.Float64Col(pos=5)

In [89]: h5 = tb.open_file('data/data.h5ts', 'w')  ❺

In [90]: ts = h5.create_ts('/', 'data', desc)  ❻

In [91]: h5  ❼
Out[91]: File(filename=data/data.h5ts, title='', mode='w',
root_uep='/',
```

```
           filters=Filters(complevel=0, shuffle=False,
bitshuffle=False,
           fletcher32=False, least_significant_digit=None))
          / (RootGroup) ''
          /data (Group/Timeseries) ''
          /data/y2020 (Group) ''
          /data/y2020/m05 (Group) ''
          /data/y2020/m05/d07 (Group) ''
          /data/y2020/m05/d07/ts_data (Table(0,)) ''
            description := {
            "timestamp": Int64Col(shape=(), dflt=0, pos=0),
            "No0": Float64Col(shape=(), dflt=0.0, pos=1),
            "No1": Float64Col(shape=(), dflt=0.0, pos=2),
            "No2": Float64Col(shape=(), dflt=0.0, pos=3),
            "No3": Float64Col(shape=(), dflt=0.0, pos=4),
            "No4": Float64Col(shape=(), dflt=0.0, pos=5)}
            byteorder := 'little'
            chunkshape := (1365,)
```

❶ `TsTables` (install it from *https://github.com/yhilpisch/tstables)* …

❷ … `PyTables` are imported.

❸ The first column of the table is a `timestamp` represented as an `int` value.

❹ All data columns contain `float` values.

❺ This opens a new database file for writing.

❻

The `TsTables` table is created at the root node, with name `data` and given the class-based description `desc`.

❼ Inspecting the database file reveals the basic principle behind the hierarchical structuring in years, months and days.

Third, the writing of the sample data stored in a `DataFrame` object to the `table` object on disk. One of the major benefits of `TsTables` is the convenience with which this operation is accomplished, namely by a simple method call. Even better, that convenience here is coupled with speed. With regard to the structure in the database, `TsTables` chunks the data into sub-sets of a single day. In the example case where the frequency is set to one second, this translates into 24 x 60 x 60 = 86,400 data rows per full day worth of data.

```
In [92]: %time ts.append(data)  ❶
         CPU times: user 549 ms, sys: 259 ms, total: 809 ms
         Wall time: 854 ms

In [93]: # h5  ❷
```

```
File(filename=data/data.h5ts, title='', mode='w',
root_uep='/',
        filters=Filters(complevel=0, shuffle=False,
bitshuffle=False,
        fletcher32=False,
least_significant_digit=None))
/ (RootGroup) ''
/data (Group/Timeseries) ''
/data/y2020 (Group) ''
/data/y2021 (Group) ''
```

```
/data/y2021/m01 (Group) ''
/data/y2021/m01/d01 (Group) ''
/data/y2021/m01/d01/ts_data (Table(86400,)) ''
  description := {
  "timestamp": Int64Col(shape=(), dflt=0, pos=0),
  "No0": Float64Col(shape=(), dflt=0.0, pos=1),
  "No1": Float64Col(shape=(), dflt=0.0, pos=2),
  "No2": Float64Col(shape=(), dflt=0.0, pos=3),
  "No3": Float64Col(shape=(), dflt=0.0, pos=4),
  "No4": Float64Col(shape=(), dflt=0.0, pos=5)}
  byteorder := 'little'
  chunkshape := (1365,)
/data/y2021/m01/d02 (Group) ''
/data/y2021/m01/d02/ts_data (Table(86400,)) ''
  description := {
  "timestamp": Int64Col(shape=(), dflt=0, pos=0),
  "No0": Float64Col(shape=(), dflt=0.0, pos=1),
  "No1": Float64Col(shape=(), dflt=0.0, pos=2),
  "No2": Float64Col(shape=(), dflt=0.0, pos=3),
  "No3": Float64Col(shape=(), dflt=0.0, pos=4),
  "No4": Float64Col(shape=(), dflt=0.0, pos=5)}
  byteorder := 'little'
  chunkshape := (1365,)
/data/y2021/m01/d03 (Group) ''
/data/y2021/m01/d03/ts_data (Table(86400,)) ''
  description := {
  "timestamp": Int64Col(shape=(), dflt=0, pos=0),
        ...
```

❶ This appends the `DataFrame` object via a simple method call.

❷ The `table` object shows 86,400 rows per day after the `append()` operation.

Reading sub-sets of the data from a `TsTables` `table` object is generally really fast since this is what it is optimized for in the first place. In this regard, `TsTables` supports typical algorithmic trading applications, like backtesting, pretty well. Another contributing factor is that `TsTables` returns the data already as a `DataFrame` object such that additional conversions are not necessary in general.

```
In [94]: import datetime

In [95]: start = datetime.datetime(2021, 1, 2)  ❶

In [96]: end = datetime.datetime(2021, 1, 3)  ❷

In [97]: %time subset = ts.read_range(start, end)  ❸
         CPU times: user 10.8 ms, sys: 4.87 ms, total: 15.7 ms
         Wall time: 14.1 ms

In [98]: start = datetime.datetime(2021, 1, 2, 12, 30, 0)

In [99]: end = datetime.datetime(2021, 1, 5, 17, 15, 30)

In [100]: %time subset = ts.read_range(start, end)
          CPU times: user 40.4 ms, sys: 25.7 ms, total: 66.1 ms
          Wall time: 66.4 ms

In [101]: subset.info()
          <class 'pandas.core.frame.DataFrame'>
          DatetimeIndex: 276331 entries, 2021-01-02 12:30:00 to
2021-01-05
           17:15:30
          Data columns (total 5 columns):
           #   Column  Non-Null Count   Dtype
          ---  ------  --------------   -----
           0   No0     276331 non-null  float64
```

```
 1   No1     276331 non-null  float64
 2   No2     276331 non-null  float64
 3   No3     276331 non-null  float64
 4   No4     276331 non-null  float64
dtypes: float64(5)
memory usage: 12.6 MB

In [102]: h5.close()

In [103]: rm data/*
```

❶ This defines the starting date and …

❷ … end date for the data retrieval operation.

❸ The `read_range()` method takes the start and end dates as input — reading here is only a matter of milliseconds.

New data that is retrieved during a day can be appended to the `TsTables table` object as illustrated before. The package is therefore a valuable addition to the capabilities of `pandas` in combination with `HDFStore` objects when it comes to the efficient storage and retrieval of (large) financial time series data sets over time.

Storing Data with SQLite3

Financial times series data can also be written directly from a `DataFrame` object to a relational database like `SQLite3`.[4] The use of a relational database might be useful in scenarios where the SQL query language is applied to implement more sophisticated analyses. With

regard to speed and also disk usage, relational databases cannot, however, compare with the other approaches that rely on binary storage formats like HDF5.

The `DataFrame` class provides the method `to_sql()` (see the `to_sql()` API reference page) to write data to a table in a relational database. The size on disk with 100+ MB indicates that there is quite some overhead overhead when using relational databases.

```
In [104]: %time data = generate_sample_data(1e6, 5,
'1min').round(4)  ❶
          CPU times: user 417 ms, sys: 92.9 ms, total: 510 ms
          Wall time: 537 ms

In [105]: data.info()  ❶
          <class 'pandas.core.frame.DataFrame'>
          DatetimeIndex: 1000000 entries, 2021-01-01 00:00:00 to
2022-11-26
           10:39:00
          Freq: T
          Data columns (total 5 columns):
           #   Column  Non-Null Count    Dtype
          ---  ------  --------------    -----
           0   No0     1000000 non-null  float64
           1   No1     1000000 non-null  float64
           2   No2     1000000 non-null  float64
           3   No3     1000000 non-null  float64
           4   No4     1000000 non-null  float64
          dtypes: float64(5)
          memory usage: 45.8 MB

In [106]: import sqlite3 as sq3  ❷

In [107]: con = sq3.connect('data/data.sql')  ❸
```

```
In [108]: %time data.to_sql('data', con)  ❹
          CPU times: user 4.86 s, sys: 496 ms, total: 5.36 s
          Wall time: 5.54 s

In [109]: ls -n data/data.*
          -rw-r--r--@ 1 501  20  105316352 May  7 11:16
data/data.sql
```

❶ The sample financial data set has 1,000,000 rows and five columns; memory usage is about 46 MB.

❷ This imports the `SQLite3` module.

❸ A connection is opened to a new database file.

❹ Writing the data to the relational database a couple of seconds.

One strength of relational databases is the ability to implement (out-of-memory) analytics tasks based on standardized SQL statements. As an example, consider a query that selects for column `No1` all those rows where the value in that row lies between 105 and 108.

```
In [110]: query = 'SELECT * FROM data WHERE No1 > 105 and No2 <
108'  ❶

In [111]: %time res = con.execute(query).fetchall()  ❷
          CPU times: user 77.2 ms, sys: 32.1 ms, total: 109 ms
          Wall time: 108 ms
```

```
In [112]: res[:5]  ❸
Out[112]: [('2021-01-02 14:56:00', 95.8818, 105.0073, 101.8703,
95.0334,
           100.2481),
          ('2021-01-02 15:02:00', 96.0003, 105.0244, 101.8224,
94.9175,
           100.2968),
          ('2021-01-02 15:03:00', 96.1241, 105.0055, 101.7528,
94.9619,
           100.3256),
          ('2021-01-02 15:04:00', 96.2036, 105.0295, 101.8097,
95.0345,
           100.3991),
          ('2021-01-02 15:05:00', 96.317, 105.0074, 101.8206,
95.0034,
           100.3331)]

In [113]: len(res)  ❹
Out[113]: 789

In [114]: con.close()

In [115]: rm data/*
```

❶ The SQL query as a Python `str` object.

❷ The query executed to retrieve all results rows.

❸ The first five results printed.

❹ The length of the results `list` object.

Admittedly, such simple queries are possible with `pandas` as well if the data set fits into memory. However, the SQL query language has proven use- and powerful for decades now and should be in the algorithmic trader's arsenal of data weapons.

Conclusions

This chapter covers the handling of financial time series data. It illustrates the reading of such data from different file-based sources, like CSV files. It also shows how to retrieve financial data from web services like the one of Quandl for end-of-day and options data. Open financial data sources are a valuable addition to the financial landscape. Quandl is a platform integrating thousands of open data sets under the umbrella of a unified API.

Another important topic covered in this chapter is the efficient storage of complete `DataFrame` objects on disk as well as of the data contained in such an in-memory object to databases. Database flavors used in this chapter include the HDF5 database standard as well the light-weight relational database `SQLite3`. This chapter lays the foundation for [Link to Come] which addresses vectorized backtesting, [Link to Come] which covers machine learning and deep learning for market prediction as well as [Link to Come] that discusses event-based backtesting of trading strategies.

Further Resources

You find more information about Quandl following these links:

- *http://quandl.org*

Information about the package used to retrieve data from that source is found here:

- Python wrapper page on Quandl
- Github page of the Quandl Python wrapper

You should consult the official documentation pages for more information on the packages used in this chapter:

- `pandas` home page
- `PyTables` home page
- `TsTables` fork on Github
- `SQLite` home page

Books cited in this chapter:

- Hilpisch, Yves (2018): *Python for Finance*. 2nd ed., O'Reilly, Beijing et al.
- McKinney, Wes (2017): *Python for Data Analysis*. 2nd ed., O'Reilly, Beijing et al.

Python Scripts

The following Python script generates sample financial time series data based on a Monte Carlo simulation for a geometric Brownian motion (see Hilpisch (2018, ch. 12)).

```
#
# Python Module to Generate a
# Sample Financial Data Set
#
# Python for Algorithmic Trading
# (c) Dr. Yves J. Hilpisch
# The Python Quants GmbH
#
import numpy as np
import pandas as pd

r = 0.05  # constant short rate
sigma = 0.5  # volatility factor


def generate_sample_data(rows, cols, freq='1min'):
    '''
    Function to generate sample financial data.

    Parameters
    ==========
    rows: int
        number of rows to generate
    cols: int
        number of columns to generate
    freq: str
        frequency string for DatetimeIndex

    Returns
    =======
    df: DataFrame
```

```
        DataFrame object with the sample data
    '''
    rows = int(rows)
    cols = int(cols)
    # generate a DatetimeIndex object given the frequency
    index = pd.date_range('2021-1-1', periods=rows, freq=freq)
    # determine time delta in year fractions
    dt = (index[1] - index[0]) / pd.Timedelta(value='365D')
    # generate column names
    columns = ['No%d' % i for i in range(cols)]
    # generate sample paths for geometric Brownian motion
    raw = np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt +
                   sigma * np.sqrt(dt) *
                   np.random.standard_normal((rows, cols)),
axis=0))
    # normalize the data to start at 100
    raw = raw / raw[0] * 100
    # generate the DataFrame object
    df = pd.DataFrame(raw, index=index, columns=columns)
    return df


if __name__ == '__main__':
    rows = 5  # number of rows
    columns = 3  # number of columns
    freq = 'D'  # daily frequency
    print(generate_sample_data(rows, columns, freq))
```

---

1 Source: “Bad Election Day Forecasts Deal Blow to Data Science — Prediction models suffered from narrow data, faulty algorithms and human foibles.” Wall Street Journal, 09. November 2016.

2 Of course, multiple `DataFrame` objects could also be stored in a single `HDFStore` object.

3 All values reported here are from the author's MacMini with Intel i7 hexa core processor (12 threads), 32 GB of random access memory (DDR4 RAM) and a 512 GB solid state drive (SSD).

4 `pandas` also supports database connections via `SQLAlchemy`, a Python abstraction layer package for diverse relational databases (refer to the `SQLAlchemy` home page). This in turn allows for the use of, for example, `MySQL` as the relational database backend.

About the Author

Dr. Yves J. Hilpisch is founder and managing partner of The Python Quants, a group focusing on the use of open source technologies for financial data science, artificial intelligence, algorithmic trading, and computational finance. He is also founder and CEO of The AI Machine, a company focused on AI-powered algorithmic trading via a proprietary strategy execution platform.

In addition to this book, he is the author of the following books:

- Artificial Intelligence in Finance (O'Reilly, forthcoming),
- Python for Finance (2nd ed., O'Reilly, 2018),
- Derivatives Analytics with Python (Wiley, 2015), and
- Listed Volatility and Variance Derivatives (Wiley, 2017).

Yves is Adjunct Professor for Computational Finance and lectures on Algorithmic Trading at the CQF Program. He is also the director of the first online training programs leading to University Certificates in Python for Algorithmic Trading and Python for Computational Finance, respectively.

Yves wrote the financial analytics library DX Analytics and organizes meetups, conferences, and bootcamps about Python for quantitative finance and algorithmic trading in London, Frankfurt, Berlin, Paris, and New York. He has given keynote speeches at technology conferences in the United States, Europe, and Asia.