Chapter 1. Why Python for Finance

Banks are essentially technology firms.

Hugo Banziger

The Python Programming Language

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:

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 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 other features, Python is:

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 macOS. It is used to build desktop as well as web applications, and it can be used on the largest clusters and most powerful servers as well as on such small devices as the Raspberry Pi.

Dynamically typed

Types in Python are in general inferred at 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 do it.

Although that way may not be obvious at first unless you're Dutch.

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!
```

A 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. In July 2018, van Rossum stepped down from this position after decades of being an active driver of the Python core development efforts. 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
- **Python 3.7** released in June 2018

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 tons 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 3.7 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 interactive 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, known as the *Python Standard Library*. 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)

        ---------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)
        <ipython-input-3-74f22a2fd43b> in <module>
        ----> 1 log(1)

        NameError: name 'log' is not defined

In [4]: import math

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

Without further imports, an error is raised.

After importing the `math` module, the calculation can be executed.

While `math` is a standard Python module available with any Python installation, there are many more packages that can be installed optionally and that can be used in the very same fashion as the standard modules. Such packages 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 use interactive Python environments: IPython and Jupyter, respectively. These are probably the most widely used interactive Python environments at the time of this writing. Although IPython started out as just an enhanced interactive Python shell, it today has many features typically found in integrated development environments (IDEs), such as support for profiling and debugging. Those features missing in IPython are typically provided by advanced text/code editors, like Vim, which can also be integrated with IPython. Therefore, it is not unusual to combine IPython with one's text/code editor of choice to form the basic toolchain for a Python development process.

IPython enhances the standard interactive shell in many ways. Among other things, 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 `?` before or after 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 proved so useful and popular that it evolved into an independent, language-agnostic project now called Jupyter. Given this background, it is no surprise that Jupyter Notebook inherits most of the beneficial features of IPython—and offers much more, for example when it comes to visualization.

Refer to VanderPlas (2016, Chapter 1) for more details on using IPython.

The 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 in Python all they might require 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 packages 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.[1] A major reason for this is that its basic syntax is easy to learn and easy to understand, even for the non-developer. In addition, it is helpful that Python supports almost all programming styles.[2]

The Scientific Stack

There is a certain set of packages 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 subpackages 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 package for Python, providing both 2D and 3D visualization capabilities.

`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 application programming interface (API) for many different ML algorithms, such as for estimation, classification, or clustering.

`PyTables`

`PyTables` is a popular wrapper for the `HDF5` data storage package; it is a package to implement optimized, disk-based I/O operations based on a hierarchical database/file format.

Depending on the specific domain or problem, this stack is enlarged by additional packages, which more often than not have in common that they build on top of one or more of these fundamental packages. However, the *least common denominator* or *basic building blocks* in general are the `NumPy ndarray` class (see Chapter 4) and 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 a popular language often compared to Python. The language's website describes Ruby as:

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

also provides by default anything that you would expect, say, as a seasoned web developer or systems administrator. In addition, Python is 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

With these "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 one 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 biotechnology 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:

FRAMINGHAM, Mass., June 14, 2018 – Worldwide spending on information technology (IT) by financial services firms will be nearly $500 billion in 2021, growing from $440 billion in 2018, according to new data from a series of Financial Services IT Spending Guides from International Data Corporation (IDC).

IDC

In particular, banks and other financial institutions are engaging in a race to make their business and operating models digital:

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 to maintain existing systems and build new ones. Large investment banks with heavy technological requirements often have technology budgets of several billion USD per year.

Technology as Enabler

The technological development has also contributed to innovations and efficiency improvements in the financial sector. Typically, projects in this area run under the umbrella of *digitalization*.

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” 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. Part IV covers topics related to the algorithmic trading of financial instruments.

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 today in certain areas in finance 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. With the development of the Python ecosystem, such efforts have become more efficient and budgets in this regard can be reduced significantly today compared to, say, 10 years ago. Part V covers derivatives analytics and builds a small but powerful and flexible derivatives pricing library with Python and standard Python packages alone.

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 or can be rented from a cloud provider based on a flexible fee plan. Chapter 2 shows how to set up an infrastructure in the cloud for interactive financial analytics, application development, and deployment with Python. The budgets for such a professional infrastructure start at a few USD per month. 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, and Data Volumes

The one dimension of the finance industry that has been influenced most by technological advances is the *speed and frequency* with which financial transactions are decided and executed. 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 time scales makes it necessary to react in real time. On the other hand, the increasing speed and frequency of trading makes the data volumes further increase. This leads to processes that reinforce each other and push the average time scale for financial transactions systematically down. This is a trend that had already started a decade ago:

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 closing quotes. This kind of data is what most of today's finance theory is based on. For example, modern or mean-variance

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 during a single trading hour the stock price of Apple Inc. (AAPL) may be quoted around 15,000 times—roughly twice the number of quotes compared to available end-of-day closing quotes over 30 years (see the example in "Data-Driven and AI-First Finance"). 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, and 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 and microsecond scales important as of today, consistent financial concepts and theories in the traditional sense that have proven to be somewhat robust over time are still missing.

All these challenges can in general 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, or the large-scale overnight calculation of credit valuation 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 as well as 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 central processing unit (CPU) side and massively parallel architectures on the general-purpose graphics processing unit (GPGPU) side. Current GPGPUs have computing cores in the thousands, 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 programming paradigms and techniques to harness the power of such hardware.

Python for Finance

The previous section described 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

This section analyzes how Python can help in addressing several of the challenges these imply. But first, on a more fundamental level, a brief analysis of Python for finance from a language and syntax point of view.

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, evaluate an integral, or simply visualize some data. In general, at this stage, little thought is given to topics like a formal development process, testing, documentation, or deployment. However, this especially seems to be the stage where people fall in love with Python. A major reason for this might be that Python syntax is generally quite close to the mathematical syntax used to describe scientific problems or financial algorithms.

This can be illustrated by a financial algorithm, namely the valuation of a European call option by Monte Carlo simulation. The example considers a Black-Scholes-Merton (BSM) setup in which the option's underlying risk factor follows a geometric Brownian motion.

Assume 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 $\sigma\sigma = 0.2$

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

$$S_T=S_0\exp((r-12\sigma^2)T+\sigma T--\sqrt{}z)ST=S0expr-12\sigma2T+\sigma Tz$$

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

1. Draw I pseudo-random numbers $z(i), i\in\{1,2,...,I\}$ $z(i), i\in\{1,2,...,I\}$, from the standard normal distribution.
2. Calculate all resulting index levels at maturity $S_T(i)$ ST(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 as given in Equation 1-2.

Equation 1-2. Monte Carlo estimator for European option

$$C_0\approx e^{-rT}1I\sum_I h_T(i)C0\approx e-rT1I\sum IhT(i)$$

This problem and algorithm must now be translated into Python. The following code implements the required steps:

```
In [6]: import math
        import numpy as np

In [7]: S0 = 100.
        K = 105.
```

```
T = 1.0
r = 0.05
sigma = 0.2
```

```
In [8]: I = 100000

In [9]: np.random.seed(1000)

In [10]: z = np.random.standard_normal(I)

In [11]: ST = S0 * np.exp((r - sigma ** 2 / 2) * T + sigma * math.sqrt(T) * z)


In [12]: hT = np.maximum(ST - K, 0)

In [13]: C0 = math.exp(-r * T) * np.mean(hT)

In [14]: print('Value of the European call option: {:5.3f}.'.format(C0))
         Value of the European call option: 8.019.
```

`NumPy` is used here as the main package.

The model and simulation parameter values are defined.

The seed value for the random number generator is fixed.

Standard normally distributed random numbers are drawn.

End-of-period values are simulated.

The option payoffs at maturity are calculated.

The Monte Carlo estimator is evaluated.

The resulting value estimate is printed.

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 or Jupyter Notebook. However, code that is meant to be reused regularly typically gets organized in so-called *modules* (or *scripts*), which are single Python files (technically 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, 2nd ed.
# (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)   # pseudo-random 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)   # payoff 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 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. One then has:

- **English** for *writing and talking* about scientific and financial problems, etc.
- **Mathematics** for *concisely, 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 in general straightforward to translate from the mathematical representation into the Pythonic implementation. This makes prototyping, development, and code maintenance in finance quite efficient with Python.

In some areas, it is common practice to use *pseudo-code* and therewith to introduce a fourth language family member. The role of pseudo-code 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, pseudo-code takes into account how computers work in principle.

This practice generally has its cause in the fact that with most (compiled) 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 *pseudo-code 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 one to do that alternative technologies do not allow for?

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 tremendously from such powerful tools as IPython, Jupyter Notebook, and packages like `pandas`.

Consider a finance student who is writing their master's thesis and is interested in S&P 500 index values. They want to analyze historical index levels for, say, a few years to see how the volatility of the index has fluctuated over time and hope 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. The student 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 volatility results

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

```
In [16]: import numpy as np
         import pandas as pd
         from pylab import plt, mpl

In [17]: plt.style.use('seaborn')
         mpl.rcParams['font.family'] = 'serif'
         %matplotlib inline

In [18]: data = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                            index_col=0, parse_dates=True)
         data = pd.DataFrame(data['.SPX'])
         data.dropna(inplace=True)
         data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2138 entries, 2010-01-04 to 2018-06-29
         Data columns (total 1 columns):
         .SPX    2138 non-null float64
         dtypes: float64(1)
         memory usage: 33.4 KB

In [19]: data['rets'] = np.log(data / data.shift(1))
         data['vola'] = data['rets'].rolling(252).std() * np.sqrt(252)

In [20]: data[['.SPX', 'vola']].plot(subplots=True, figsize=(10, 6));
```

This imports `NumPy` and `pandas`.

This imports `matplotlib` and configures the plotting style and approach for Jupyter.

`pd.read_csv()` allows the retrieval of remotely or locally stored data sets in comma-separated values (CSV) form.

A subset of the data is picked and `NaN` ("not a number") values eliminated.

This shows some metainformation about the data set.

The log returns are calculated in vectorized fashion ("no looping" on the Python level).

The rolling, annualized volatility is derived.

This 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, as well as visualization of the results. The example illustrates that `pandas` makes working with whole time series almost as simple as doing mathematical operations on floating-point numbers.

Translated to a professional finance context, the example implies that financial analysts can—when applying the right Python tools and packages that provide high-level abstractions—focus on their domain and not on the technical intrinsicalities. Analysts can also 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.

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

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

often 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:

Idioms and paradigms

: In general, many different ways can lead to the same result in Python, but sometimes with rather different performance characteristics; "simply" choosing the right way (e.g., a specific implementation approach, such as the judicious use of data structures, avoiding loops through vectorization, or the use of a specific package such as `pandas`) can improve results significantly.

Compiling

: Nowadays, there are several performance packages 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 make such functions orders of magnitude faster than pure Python code; popular ones are `Cython` and `Numba`.

Parallelization

: Many computational tasks, in particular in finance, can significantly 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 technologies.

This subsection sticks to a simple, but still realistic, example that touches upon all three strategies (later chapters illustrate the strategies in detail). 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 [21]: import math
             loops = 2500000
```

```
            a = range(1, loops)
            def f(x):
                return 3 * math.log(x) + math.cos(x) ** 2
            %timeit r = [f(x) for x in a]
            1.59 s ± 41.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop
each)
```

The Python interpreter needs about 1.6 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., *precompiled*) functions to handle such array-based operations:

```
In [22]: import numpy as np
            a = np.arange(1, loops)
            %timeit r = 3 * np.log(a) + np.cos(a) ** 2
            87.9 ms ± 1.73 ms per loop (mean ± std. dev. of 7 runs, 10
loops each)
```

Using `NumPy` considerably reduces the execution time to about 88 milliseconds. However, there is even a package specifically dedicated to this kind of task. It is called `numexpr`, for "numerical expressions." It *compiles* the expression to improve upon the performance of the general `NumPy` functionality by, for example, avoiding in-memory copies of `ndarray` objects along the way:

```
In [23]: import numexpr as ne
            ne.set_num_threads(1)
            f = '3 * log(a) + cos(a) ** 2'
            %timeit r = ne.evaluate(f)
            50.6 ms ± 4.2 ms per loop (mean ± std. dev. of 7 runs, 10
loops each)
```

Using this more specialized approach further reduces execution time to about 50 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 [24]: ne.set_num_threads(4)
            %timeit r = ne.evaluate(f)
            22.8 ms ± 1.76 ms per loop (mean ± std. dev. of 7 runs, 10
loops each)
```

Parallelization brings execution time further down to below 23 milliseconds in this case, with four threads utilized. Overall, this is a performance improvement of more than 90 times. Note, in particular, that this kind of improvement is possible without altering the basic problem/algorithm and without knowing any detail about compiling or parallelization approaches. The capabilities are accessible from a high

level even by non-experts. However, one has to be aware, of course, of which capabilities and options exist.

This example shows that Python provides a number of options to make more out of existing resources—i.e., to *increase productivity*. With the parallel approach, three times as many calculations can be accomplished in the same amount of time as compared to the sequential approach—in this case simply by telling Python to use multiple 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 for financial institutions. 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 still 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, deployment, access management, and version control, 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 compiled languages, such as C++ or Java, are used to fulfill the requirements for deployment and production. Also, a formal development process with professional tools, version control, etc., is generally 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; risks arise during translation

Diverse skill sets

: Different departments show different skill sets and use different languages to implement “the same things”; people not only program but also speak different languages

Legacy code

: Code is available and has to be maintained in different languages, often using different styles of implementation

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 analytics, financial application development, and algorithm implementation.

Data-Driven and AI-First Finance

Basically all the observations regarding the relationship of technology and the financial industry first formulated in 2014 for the first edition of this book still seem pretty current and important in August 2018, at the time of updating this chapter for the second edition of the book. However, this section comments on two major trends in the financial industry that are about to reshape it in a fundamental way. These two trends have mainly crystallized themselves over the last few years.

Data-Driven Finance

Some of the most important financial theories, such as MPT and CAPM, date as far back as to the 1950s and 1960s. However, they still represent a cornerstone in the education of students in such fields as economics, finance, financial engineering, and business administration. This might be surprising since the empirical support for most of these theories is meager at best, and the evidence is often in complete contrast to what the theories suggest and imply. On the other hand, their popularity is understandable since they are close to humans' expectations of how financial markets might behave and since they are elegant mathematical theories resting on a number of appealing, if in general too simplistic, assumptions.

The *scientific method*, say in physics, starts with *data*, for example from experiments or observations, and moves on to *hypotheses and theories* that are then *tested* against the data. If the tests are positive, the hypotheses and theories might be refined and properly written down, for instance, in the form of a research paper for publication. If the tests are negative, the hypotheses and theories are rejected and the search begins anew for ones that conform with the data. Since physical laws are stable over time, once such a law is discovered and well tested it is generally there to stay, in the best case, forever.

The history of (quantitative) finance in large parts contradicts the scientific method. In many cases, theories and models have been developed "from scratch" on the basis of simplifying mathematical assumptions with the goal of discovering elegant answers to central problems in finance. Among others, popular assumptions in finance are normally distributed returns for financial instruments and linear relationships between quantities of interest. Since these phenomena are hardly ever found in financial markets, it should not come as a surprise that empirical evidence for the elegant theories is often lacking. Many financial theories and models have been formulated, proven, and published first and have only later been tested empirically. To some extent, this is of course due to the fact that financial data back in the 1950s to the 1970s or even later was not available in the form that it is today even to students getting started with a bachelor's in finance.

The availability of such data to financial institutions has drastically increased since the early to mid-1990s, and nowadays even individuals doing financial research or getting involved in algorithmic trading have access to huge amounts of historical data down to the tick level as well as real-time tick data via streaming services. This allows us to return to the scientific method, which starts in general with the data before ideas, hypotheses, models, and strategies are devised.

A brief example shall illustrate how straightforward it has become today to retrieve professional data on a large scale even on a local machine, making use of Python and a professional data subscription to the Eikon Data APIs. The following example retrieves tick data for the Apple Inc. stock for one hour during a regular trading day. About 15,000 tick quotes, including volume information, are retrieved. While the symbol for the stock is `AAPL`, the Reuters Instrument Code (RIC) is `AAPL.O`:

```
In [26]: import eikon as ek

In [27]: data = ek.get_timeseries('AAPL.O', fields='*',
                                  start_date='2018-10-18 16:00:00',
                                  end_date='2018-10-18 17:00:00',
                                  interval='tick')

In [28]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 35350 entries, 2018-10-18 16:00:00.002000 to 2018-10-
18
          16:59:59.888000
         Data columns (total 2 columns):
         VALUE     35285 non-null float64
         VOLUME    35350 non-null float64
         dtypes: float64(2)
         memory usage: 828.5 KB

In [29]: data.tail()
Out[29]: AAPL.O                      VALUE  VOLUME
         Date
         2018-10-18 16:59:59.433  217.13    10.0
         2018-10-18 16:59:59.433  217.13    12.0
         2018-10-18 16:59:59.439  217.13   231.0
         2018-10-18 16:59:59.754  217.14   100.0
         2018-10-18 16:59:59.888  217.13   100.0
```

Eikon Data API usage requires a subscription and an API connection.

Retrieves the tick data for the Apple Inc. (`AAPL.O`) stock.

Shows the last five rows of tick data.

The Eikon Data APIs give access not only to structured financial data, such as historical price data, but also to unstructured data such as *news articles*. The next example retrieves metadata for a small selection of news articles and shows the beginning of one of the articles as full text:

```
In [30]: news = ek.get_news_headlines('R:AAPL.O Language:LEN',
                                      date_from='2018-05-01',
                                      date_to='2018-06-29',
                                      count=7)

In [31]: news
Out[31]:
                                        versionCreated  \
    2018-06-28 23:00:00.000 2018-06-28 23:00:00.000
    2018-06-28 21:23:26.526 2018-06-28 21:23:26.526
    2018-06-28 19:48:32.627 2018-06-28 19:48:32.627
    2018-06-28 17:33:10.306 2018-06-28 17:33:10.306
    2018-06-28 17:33:07.033 2018-06-28 17:33:07.033
    2018-06-28 17:31:44.960 2018-06-28 17:31:44.960
    2018-06-28 17:00:00.000 2018-06-28 17:00:00.000


text  \
    2018-06-28 23:00:00.000  RPT-FOCUS-AI ambulances and robot doctors:
Chi...
    2018-06-28 21:23:26.526  Why Investors Should Love Apple's (AAPL) TV
En...
    2018-06-28 19:48:32.627  Reuters Insider - Trump: We're reclaiming our
...
    2018-06-28 17:33:10.306  Apple v. Samsung ends not with a whimper but
a...
    2018-06-28 17:33:07.033  Apple's trade-war discount extended for
anothe...
    2018-06-28 17:31:44.960  Other Products: Apple's fast-growing island
of...
    2018-06-28 17:00:00.000  Pokemon Go creator plans to sell the tech
behi...

                                                                    storyId
\
    2018-06-28 23:00:00.000  urn:newsml:reuters.com:20180628:nL4N1TU4F8:6
    2018-06-28 21:23:26.526  urn:newsml:reuters.com:20180628:nNRA6e2vft:1
    2018-06-28 19:48:32.627  urn:newsml:reuters.com:20180628:nRTV1vNw1p:1
    2018-06-28 17:33:10.306  urn:newsml:reuters.com:20180628:nNRA6e1oza:1
    2018-06-28 17:33:07.033  urn:newsml:reuters.com:20180628:nNRA6e1pmv:1
    2018-06-28 17:31:44.960  urn:newsml:reuters.com:20180628:nNRA6e1m3n:1
    2018-06-28 17:00:00.000  urn:newsml:reuters.com:20180628:nL1N1TU0PC:3

                             sourceCode
    2018-06-28 23:00:00.000     NS:RTRS
    2018-06-28 21:23:26.526  NS:ZACKSC
    2018-06-28 19:48:32.627     NS:CNBC
```

```
     2018-06-28 17:33:10.306  NS:WALLST
     2018-06-28 17:33:07.033  NS:WALLST
     2018-06-28 17:31:44.960  NS:WALLST
     2018-06-28 17:00:00.000    NS:RTRS

In [32]: story_html = ek.get_news_story(news.iloc[1, 2])

In [33]: from bs4 import BeautifulSoup

In [34]: story = BeautifulSoup(story_html, 'html5lib').get_text()

In [35]: print(story[83:958])
         Jun 28, 2018 For years, investors and Apple AAPL have been beholden
to
          the iPhone, which is hardly a negative since its flagship product
is
          largely responsible for turning Apple into one of the world's
biggest
          companies. But Apple has slowly pushed into new growth areas,
with
          streaming television its newest frontier. So let's take a look at
what
          Apple has planned as it readies itself to compete against the
likes of
          Netflix NFLX and Amazon AMZN in the battle for the new age of
          entertainment.Apple's second-quarter revenues jumped by 16% to
reach
          $61.14 billion, with iPhone revenues up 14%. However, iPhone unit
sales
          climbed only 3% and iPhone revenues accounted for over 62% of
total Q2
          sales. Apple knows this is not a sustainable business model,
because
          rare is the consumer product that can remain in vogue for
decades. This
          is why Apple has made a big push into news,
```

Retrieves metadata for a small selection of news articles.

Retrieves the full text of a single article, delivered as an HTML document.

Imports the `BeautifulSoup` HTML parsing package and …

… extracts the contents as plain text (a `str` object).

Prints the beginning of the news article.

Although just scratching the surface, these two examples illustrate that structured and unstructured historical financial data is available in a standardized, efficient way via Python wrapper packages and data subscription services. In many circumstances, similar data sets can be accessed for free even by individuals who make use of, for instance, trading platforms such as the one by FXCM Group, LLC, that is introduced in Chapter 14 and also used in Chapter 16. Once the data is on the Python level—independent from the original source—the full power of the Python data analytics ecosystem can be harnessed.

DATA-DRIVEN FINANCE

Data is what drives finance these days. Even some of the largest and often most successful hedge funds call themselves "data-driven" instead of "finance-driven." More and more offerings are making huge amounts of data available to large and small institutions and individuals. Python is generally the programming language of choice to interact with the APIs and to process and analyze the data.

AI-First Finance

With the availability of large amounts of financial data via programmatic APIs, it has become much easier and more fruitful to apply methods from *artificial intelligence* (AI) in general and from *machine and deep learning* (ML, DL) in particular to financial problems, such as in algorithmic trading.

Python can be considered a first-class citizen in the AI world as well. It is often the programming language of choice for AI researchers and practitioners alike. In that sense, the financial domain benefits from developments in diverse fields, sometimes not even remotely connected to finance. As one example consider the `TensorFlow` open source package for deep learning, which is developed and maintained by Google Inc. and used by (among others) its parent company Alphabet Inc. in its efforts to build, produce, and sell self-driving cars.

Although for sure not even remotely related to the problem of automatically, algorithmically trading stock, `TensorFlow` can, for example, be used to predict movements in financial markets. Chapter 15 provides a number of examples in this regard.

One of the most widely used Python packages for ML is `scikit-learn`. The code that follows shows how, in a highly simplified manner, classification algorithms from ML can be used to predict the direction of future market price movements and to base an algorithmic trading strategy on those predictions. All the details are explained in Chapter 15, so the example is therefore rather concise. First, the data import and the preparation of the features data (directional lagged log return data):

```
In [36]: import numpy as np
         import pandas as pd

In [37]: data = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                            index_col=0, parse_dates=True)
         data = pd.DataFrame(data['AAPL.O'])
         data['Returns'] = np.log(data / data.shift())
         data.dropna(inplace=True)

In [38]: lags = 6

In [39]: cols = []
         for lag in range(1, lags + 1):
             col = 'lag_{}'.format(lag)
             data[col] = np.sign(data['Returns'].shift(lag))
             cols.append(col)
         data.dropna(inplace=True)
```

Selects historical end-of-day data for the Apple Inc. stock (`AAPL.O`).

Calculates the log returns over the complete history.

Generates `DataFrame` columns with directional lagged log return data (`+1` or `-1`).

Next, the instantiation of a model object for a *support vector machine* (SVM) algorithm, the fitting of the model, and the prediction step. Figure 1-2 shows that the prediction-based trading strategy, going long or short on Apple Inc. stock depending on the prediction, outperforms the passive benchmark investment in the stock itself:

```
In [40]: from sklearn.svm import SVC

In [41]: model = SVC(gamma='auto')

In [42]: model.fit(data[cols], np.sign(data['Returns']))
Out[42]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
```

```
            decision_function_shape='ovr', degree=3, gamma='auto',
kernel='rbf',
            max_iter=-1, probability=False, random_state=None, shrinking=True,
            tol=0.001, verbose=False)

In [43]: data['Prediction'] = model.predict(data[cols])

In [44]: data['Strategy'] = data['Prediction'] * data['Returns']

In [45]: data[['Returns', 'Strategy']].cumsum().apply(np.exp).plot(
                   figsize=(10, 6));
```

Instantiates the model object.

Fits the model, given the features and the label data (all directional).

Uses the fitted model to create the predictions (in-sample), which are the positions of the trading strategy at the same time (long or short).

Calculates the log returns of the trading strategy given the prediction values and the benchmark log returns.

Plots the performance of the ML-based trading strategy compared to the performance of the passive benchmark investment.

Figure 1-2. ML-based algorithmic trading strategy vs. passive benchmark investment in Apple Inc. stock

The simplified approach taken here does not account for transaction costs, nor does it separate the data set into training and testing subsets. However, it shows how straightforward the application of ML algorithms to financial data is, at least in a technical sense; practically, a number of important topics need to be considered (see López de Prado (2018)).

AI-FIRST FINANCE

AI will reshape finance in a way that other fields have been reshaped already. The availability of large amounts of financial data via programmatic APIs functions as an enabler in this context. Basic methods from AI, ML, and DL are introduced in Chapter 13 and applied to algorithmic trading in Chapters 15 and 16. A proper treatment of *AI-first finance*, however, would require a book fully dedicated to the topic.

AI in finance, as a natural extension of data-driven finance, is for sure a fascinating and exciting field, both from a research and a practitioner's point of view. Although this book uses several methods from AI, ML, and DL in different contexts, overall the focus lies—in line with the subtitle of the book—on the fundamental Python techniques and approaches needed for *data-driven finance*. These are, however, equally important for AI-first finance.

Conclusion

Python as a language—and even more so as an ecosystem—is an ideal technological framework for the financial industry as whole and the individual working in finance alike. It is characterized by a number of benefits, like an elegant syntax, efficient development approaches, and usability for prototyping as well as production. With its huge amount of available packages, 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.

In addition, Python has become the programming language of choice for artificial intelligence in general and machine and deep learning in particular. Python is therefore the right language for data-driven finance as well as for AI-first finance, two recent trends that are about to reshape finance and the financial industry in fundamental ways.

Further Resources

The following books cover several aspects only touched upon in this chapter in more detail (e.g., Python tools, derivatives analytics, machine learning in general, and machine learning in finance):

- Hilpisch, Yves (2015). *Derivatives Analytics with Python*. Chichester, England: Wiley Finance.
- López de Prado, Marcos (2018). *Advances in Financial Machine Learning*. Hoboken, NJ: John Wiley & Sons.
- VanderPlas, Jake (2016). *Python Data Science Handbook*. Sebastopol, CA: O'Reilly.

When it comes to algorithmic trading, the author's company offers a range of online training programs that focus on Python and other tools and techniques required in this rapidly growing field:

- *http://pyalgo.tpq.io*
- *http://certificate.tpq.io*

Sources referenced in this chapter are, among others, the following:

- Ding, Cubillas (2010). "Optimizing the OTC Pricing and Valuation Infrastructure." Celent.
- Lewis, Michael (2014). *Flash Boys*. New York: W. W. Norton & Company.
- Patterson, Scott (2010). *The Quants*. New York: Crown Business.