Python for Finance

Dr. Yves J. Hilpisch | <training@tpq.io>

The Python Quants GmbH | www.tpq.io

Table of Contents

Chapter 1. Why Python for Finance

> Banks are essentially technology firms.
>
> — Hugo Banziger

1.1. 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 Python syntax is generally quite close to the mathematical syntax used to describe scientific problems or financial algorithms.

This aspect 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 $\sigma = 0.2$

In the BSM model, the index level at maturity is a random variable, given by Black-Scholes-Merton (1973) index level at maturity with z being a standard normally distributed random variable.

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

Black-Scholes-Merton (1973) index level at maturity

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

1. draw I pseudo-random numbers $z(i), i \in \{1, 2, ..., I\}$, from the standard normal distribution
2. calculate all resulting index levels at maturity $S_T(i)$ for given $z(i)$ and Black-Scholes-Merton (1973) index level at maturity
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 Monte Carlo estimator for European option

$$C_0 \approx e^{-rT} \frac{1}{I} \sum_I h_T(i) \qquad (2)$$

Monte Carlo estimator for European option

This problem and algorithm is now to be translated into Python code. The code below implements the required steps.

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

In [2]: S0 = 100.  ②
        K = 105.  ②
        T = 1.0  ②
        r = 0.05  ②
        sigma = 0.2  ②

In [3]: I = 100000  ②

In [4]: np.random.seed(1000)  ③

In [5]: z = np.random.standard_normal(I)  ④

In [6]: ST = S0 * np.exp((r - sigma ** 2 / 2) * T + sigma * math.sqrt(T) * z)  ⑤

In [7]: hT = np.maximum(ST - K, 0)  ⑥

In [8]: C0 = math.exp(-r * T) * np.mean(hT)  ⑦

In [9]: print('Value of the European call option: {:5.3f}.'.format(C0))  ⑧
        Value of the European call option: 8.019.
```

① The model and simulation parameter values are defined.

② NumPy is used here as the main package.

③ The seed value for the random number generator is fixed.

④ Standard normally distributed random numbers are drawn.

⑤ End-of-period values are simulated.

⑥ The option payoffs at maturity are calculated.

⑦ The Monte Carlo estimator is evaluated.

⑧ 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 Monte Carlo valuation of European call option and could be saved as a file named `bsm_mcs_euro.py`.

Example 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, 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*.

1.2. 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 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.

1.2.1. 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, 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 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 code below shows how exactly this works — without worrying about syntax details at this stage (everything is explained in detail in subsequent chapters).

```
In [11]: import numpy as np  ①
         import pandas as pd  ①
         from pylab import plt, mpl  ②

In [12]: plt.style.use('seaborn')  ②
         mpl.rcParams['font.family'] = 'serif'  ②
         %matplotlib inline

In [13]: data = pd.read_csv('http://hilpisch.com/tr_eikon_eod_data.csv',
                            index_col=0, parse_dates=True)  ③
         data = pd.DataFrame(data['.SPX']) ④
         data.dropna(inplace=True)  ④
         data.info()  ⑤
         <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 [14]: data['rets'] = np.log(data / data.shift(1))  ⑥
         data['vola'] = data['rets'].rolling(252).std() * np.sqrt(252)  ⑦

In [15]: data[['.SPX', 'vola']].plot(subplots=True, figsize=(10, 6));  ⑧
         plt.savefig('../images/spx_volatility.png')
```

① This imports `NumPy` and `pandas`.

② This imports `matplotlib` and configures the plotting style and approach for `Jupyter`.

③ `pd.read_csv()` allows the retrieval of remotely or locally stored data sets in `CSV` form.

④ A sub-set of the data is picked and `NaN` ("not a number") values eliminated.

⑤ This shows some meta-information about the data set.

⑥ The log returns are calculated in vectorized fashion (“no looping” on the Python level).

⑦ The rolling, annualized volatility is derived.

⑧ This finally plots the two time series.

S&P 500 closing values and annualized volatility 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.

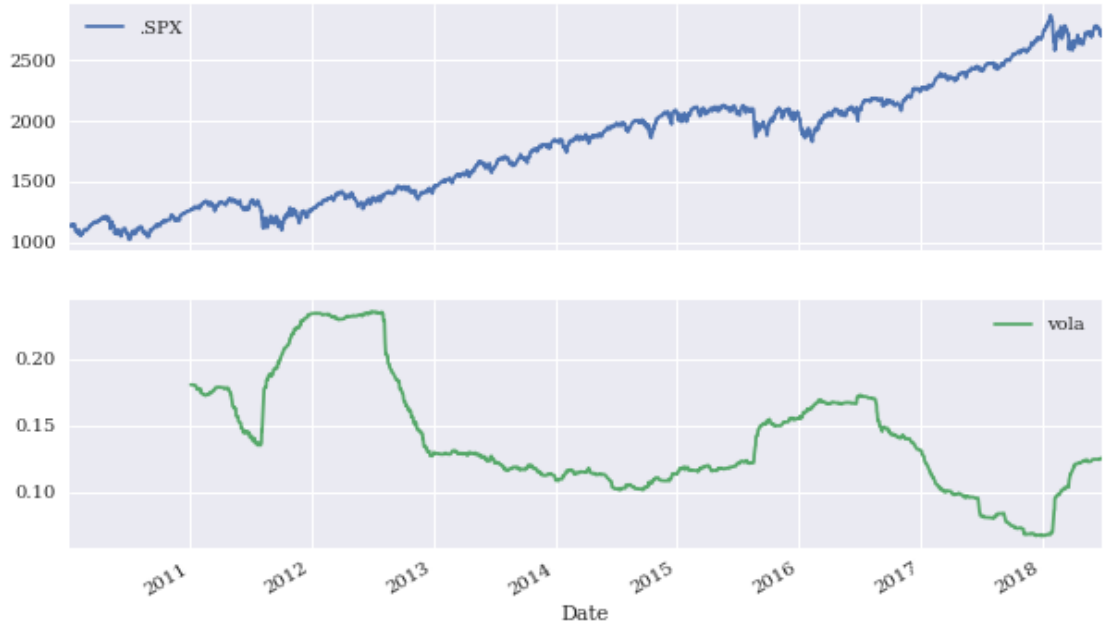

Figure 1. S&P 500 closing values and annualized volatility

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

1.2.2. 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.

Later chapters illustrate all three strategies in detail. This sub-section sticks to a simple, but still realistic, example that touches upon all three strategies. 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 [16]: 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]
         1.21 s ± 8.27 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The Python interpreter needs about 1.4 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 [17]: import numpy as np
         a = np.arange(1, loops)
         %timeit r = 3 * np.log(a) + np.cos(a) ** 2
         94.7 ms ± 1.55 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Using `NumPy` considerably reduces the execution time to about 80 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 [18]: import numexpr as ne
         ne.set_num_threads(1)
         f = '3 * log(a) + cos(a) ** 2'
         %timeit r = ne.evaluate(f)
         37.5 ms ± 936 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Using this more specialized approach further reduces execution time to about 40 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 [19]: ne.set_num_threads(4)
         %timeit r = ne.evaluate(f)
         12.8 ms ± 300 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Parallelization brings execution time further down to below 15 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.

The 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.

1.3. Conclusions

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 both 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.

1.4. 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, machine learning in finance):

- Hilpisch, Yves (2015): *Derivatives Analytics with Python*. Wiley Finance, Chichester,

England. http://dawp.tpq.io.

- López de Prado, Marcos (2018): *Advances in Financial Machine Learning*. John Wiley & Sons, Hoboken.
- VanderPlas, Jake (2016): *Python Data Science Handbook*. O'Reilly, Beijing et al.

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 — online training course
- http://certificate.tpq.io — online training program

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

- 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.