Pawel Lachowicz, PhD

Python for Quants

Volume

Fundamentals of Python 3.5
Fundamentals of NumPy
Standard Library

QuantAtRisk eBooks

Table of Contents

Preface

This is the first part out of the *Python for Quants* trilogy, the book-series that provides you with an opportunity to commence your programming experience with **Python**—a very modern and dynamically evolving computer language. Everywhere.

This book is **completely different** than anything ever written on Python, programming, quantitative finance, research, or science. It became one of the greatest challenges in my career as a writer—being able to deliver a book that **anyone can** learn programming from in the most gentle but sophisticated manner—starting from absolute beginner. I made a lot of effort **not** to follow **any** rules in book writing, solely preserving the expected skeleton: chapters, sections, margins.

It is written from a standpoint of over 21 years of experience as a programmer, with a scientific approach to the problems, seeking pinpoint solutions but foremost blended with a heart and soul—two magical ingredients making this book so unique and alive.

It is all about **Python** strongly inclined towards **quantitative** and **numerical** problems. It is thought of quantitative analysts (also known as *quants*) occupying all rooms from bedrooms to Wall Street trading rooms. Therefore, it is written for traders, algorithmic traders, and financial analysts. All students and PhDs. In fact, **for anyone** who wishes to learn Python and apply its mathematical abilities.

In this book you will find numerous examples taken from finance, however the content is not strictly limited to that one single field. Again, it is all about Python. From the beginning to the end. From the tarmac to the stratosphere of dedicated programming.

Within **Volume I**, we will try to cover the quantitative aspects of *Fundamentals of Python* supplemented with most useful language's structures taken from the Python's *Standard Library*. We will be studying the numerical and algebraical concepts of *NumPy* to equip you with the best of **Python 3.5**. Yes, the newest version of the interpreter. **This book is up to date**.

If you hold a copy of this ebook it means you are very serious about learning Python quickly and efficiently. For me it is a dream to guide you from cover to cover, leaving you wondering "what's next?", and making your own coding in Python a truly remarkable experience. **Volume I** is thought of as a story on the quantitatively dominated side of Python for beginners which, I do hope, you will love from the very first page.

If I missed something or simply left anything with a room for improvement—please email me at pawel@quantatrisk.com. The 1st edition of Volume II will come out along with the 2nd edition of Volume I. Thank you for your feedback in advance.

Ready for **Python for Quants** fly-thru experience? If so, fasten your seat belt and adjust a seat to an upright position. We are now clear for take-off!

Enjoy your flight!

Paweł Lachowicz, PhD
November 26th, 2015

About the Author

Paweł Lachowicz was born in Wrocław, Poland in 1979. At the age of twelve he became captivated by programming capability of the Commodore Amiga 500. Over the years his mind was sharply hooked on maths, physics, and computer science, concurrently exploring the frontiers of positive thinking and achieving "the impossible" in life. In 2003 he graduated from Warsaw University defending his MSc degree in astronomy; four years later—a PhD degree in signal processing applied directly to black-hole astrophysics at Polish Academy of Sciences in Warsaw, Poland. His novel discoveries secured his post-doctoral research position at the National University of Singapore in 2007. In 2010 Pawel shifted his interest towards financial markets, trading, and risk management. In 2012 he founded QuantAtRisk.com portal where he continuously writes on quantitative finance, risk, and applied Python programming.

Today, Pawel lives in Sydney, Australia (dreaming of moving to Singapore or to the USA) and serves as a freelance financial consultant, risk analyst, and algorithmic trader. Worldwide. Relaxing, he fulfils his passions as a writer, motivational speaker, yacht designer, photographer, traveler, and (sprint) runner.

He never gives up.

Acknowledgments

To all my Readers and Followers.

To Dr. Ireneusz Baran for his patience in the process of waiting for this book. For weekly encouragement to go for what valuable I could do for people around the world. For his uplifting words injected into my subconscious mind. It all helped me. A lot.

To Aneta Glińska-Broś for placing a bar significantly higher than I initially anticipated. Expect the unexpected but never back down. You kept reminding me that all the time. I did listen to You. I rebuilt myself and re-emerged stronger. Thank You!

To Dr. Yves Hilpisch, Dr. Sebastian Raschka, and Stuart Reid, CFA for the boost of motivation I experienced from your side by providing the examples to follow.

To John Hedge for the effort of reading my book and truly great time we shared in Sydney. For courage you gave me.

To Lies Leysen, Dr. Katarzyna Tajs-Zielińska, Professor Iwona and Ireneusz Tomczak, and Armando Favrin for an amazing support, positive energy, long hours spent on memorable conversations, and for reminding me a true importance of accomplishing what I have started.

To Iain Bell and Dr. Chris Dandre for giving me a chance.

To Les Brown for motivation.

To all who believed in me.

And to all who did not. You made my jet engines full of thrust.

A superbly handy function in `math` arsenal is `trunc` which provides a stiff separation of the real number from its fractional part:

```
>>> e
2.718281828459045
>>> y = trunc(e)
2
>>> type(y)
<class 'int'>
>>> y == floor(e)
True
```

trunc()

In many cases the function of `trunc` can be replaced with the operation of the floor division:

```
from math import trunc
from random import random as r
x = r()*100  # float
print(x//1 == trunc(x))
```

```
True
```

thought the former returns a float and `trunc` an integer if `x` is float.

where the last logical test is positive despite the type of **y** is an integer and `floor(e)` is a float. It is always better to use `trunc` than `floor` function if you want to get the real value of your number. In certain cases (big numbers) rounding to the floor may fail. That is why the `trunc` function is a perfect choice.

2.1.6. Precise Maths with `decimal` Module

Python's Standard Library goes one step forward. It grants us access to its another little pearl known as `decimal` module. Now the game is all about **precision of calculations**. When we use floating-point mathematical operations usually we do not think how computer represents the floats in its memory. The truth is a bit surprising when you discover that:

You may obtain **a pure fractional part of the float** as follows, e.g:

```
>>> from math import pi
>>> pi
3.141592653589793
>>> pi - pi//1
0.14159265358979312
```

```
>>> r = 0.001
>>> print("r= %1.30f" % r)  # display 30 decimal places
r= 0.001000000000000000020816681712
```

instead of

```
r= 0.001000000000000000000000000000
```

It is just the way it is: the floats are represented in a binary format that involves a finite number of bits of their representation. When used in calculations, the floats provide us with a formal assurance up to 17 decimal places. However, the rest is not ignored and in case of heavy computations those false decimal digits may **propagate**. In order to "see" it—run the following code:

Code 2.6

```
r = 0.001
t = 0.002
print("r   = %1.30f" % r)
print("t   = %1.30f" % t)
print("r+t = %1.30f" % (r+t))
```

You should get:

```
r   = 0.001000000000000000020816681712
t   = 0.002000000000000000041633363423
r+t = 0.003000000000000000062450045135
```

It is now clear how those "happy endings" accumulate an error. The question you may ask is: "Should I be worried about that?"

2.1.8. `fractions` and Approximations of Numbers

If you are 11 years young and you are studying this book because you have some problems with solving fractions at school, I've got something for you too! Python is able to perform computations and display results in a form of nominator over denominator. Hurrah!

Let's say, for `x = 0.25 = 1/4` we want to calculate the value of a simple expression:

$$y = x + 5 \cdot \frac{x}{6} = \frac{1}{4} + \frac{5}{6} \cdot \frac{1}{4} = \frac{1}{4} + \frac{5}{24} = \frac{6+5}{24} = \frac{11}{24}$$

and display result exactly as a fraction of `11/24`. It is possible with the use of `fractions` module from the Standard Library:

Code 2.12

Fraction()

```
from fractions import Fraction as fr

x = 1./4          # float
xf = fr(str(x))   # fractional form

yf = xf + 5*xf/6

print("x  = %1.5f" % x)
print("xf = %s" % xf)     # use string
print("yf = %s" % yf)     #    for output
```

returning

```
x  = 0.25000
xf = 1/4
yf = 11/24
```

The input value of **x** for `Fraction` function needs to be firstly converted to a string-type variable. In the *4*th line of 2.12 we can see how easily then our calculations can be coded. Both **xf** and **yf** variables are recognised by Python as `<class 'fractions.Fraction'>` objects and we obtain the conversion of **yf** to float-type representation simply by writing:

```
y = float(yf)
print(y)

0.458333333333
```

Now, how accurate this outcome is?

```
from math import fabs
y0 = 11./24
print("|y-y0| = %1.30f" % fabs(y-y0))

|y-y0| = 0.000000000000000000000000000000
```

Well, as for rational numbers—so far, so good. How about irrational numbers? Can they be approximated by fractions and how would such approximations look?

You can also represent any float number without `fractions` module:

```
>>> x = 1.334
>>> y = x.as_integer_ratio(); y
(3003900951456121, 2251799813685248)

>>> x = 0.75
>>> y = x.as_integer_ratio(); y
(3, 4)
>>> type(y)
<class 'tuple'>
>>> (nom, den) = y
>>> nom
3
>>> den
4
>>> type(nom)
<class 'int'>
```

More on **tuples** in Section 2.5.1.

3.7. Essential Matrix and Linear Algebra

3.7.1. NumPy's ufuncs: Acceleration Built-In

When you buy a nice car with 300 horsepower under the hood, you expect it to perform really well in most road conditions. The amount of torque combined with a high-performance engine gives you a lot of thrust and confidence while overtaking. The same level of expectations arises when you decide, from now on, to use Python for your numerical computations.

Everyone heard about highly efficient engines of C, C++, or Fortran when it comes to the speed of the code execution. A magic takes place while the code is compiled to its machine-digestible version in order to gain the noticeable speed-ups. The trick is that within these languages every variable is declared, i.e. its type is known in advance before any mathematical operation begins. This is not the case of Python where variables are checked on-the-way as the interpreter reads the code. For example, if we declare:

```
r = 7
```

Python checks the value on the right-hand side first and if it does not have a floating-point representation, it will assume and remember that **r** is an integer. So, what does it have to do with the speed? Analyse the following case study.

Let's say we would like to compute the values of the function:

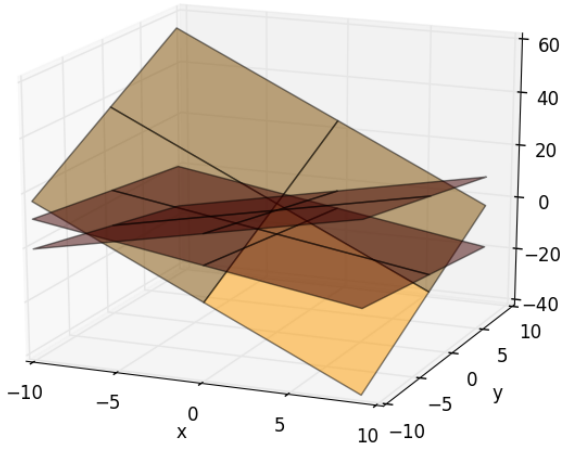
$$f(x) = \sqrt{|\sin(x-\pi)\cos^2(x+\pi)|\left(1+e^{\frac{-x}{\sqrt{2\pi}}}\right)}$$

for a grid of x defined between *0.00001* and *100* with the resolution of *0.00001*. Based on our knowledge till now, we can find all corresponding solutions using at least two methods: list and loop or list comprehension. The third method employs so-called NumPy's **universal functions** (or **ufunc**s for short). As we will see below, the former two methods are significantly slower than the application of ufuncs. The latter performs *vectorised* operations on arrays, i.e. a specific ufunc applies to each element. Since the backbone of ufuncs is CPython, the performance of our engine is optimised for speed.

Code 3.12 Compute *f(x)* as given above using three different methods: (1) list and loop, (2) list comprehension, and (3) NumPy's ufuncs. Measure and compare the time required to reach the end result.

```
from math import sin, cos, exp, pi, sqrt, pow
from time import time
import numpy as np
from matplotlib import pyplot as plt

def fun(x):
    return sqrt(abs(sin(x-pi)*pow(cos(x+pi), 2)) * (1+exp(-x/
                sqrt(2*pi))))
```



Even with a capability of the plot rotation in `matplotlib`'s default viewer, you may discover that it is tricky to "see" the crossing point at

```
[4.95, 8.56, -4.73]
```

what does not mean it's not there! ☺

Eigenvectors and Principal Component Analysis (PCA) for *N*-Asset Portfolio

Probably the most famous application of the algebra's concept of eigenvectors **in quantitative finance** is the **Principal Component Analysis** (PCA) for *N*-Asset Portfolio. The PCA delivers a simple, non-parametric method of extraction of the relevant information from often confusing data sets.

The real-world data usually hold some relationships among their variables and, as a good approximation, in the first instance we may suspect them to be of the linear (or close to linear) form. And the *linearity* is one of stringent, however, powerful assumptions standing behind PCA.

Let's consider a practical example everyone can use and reapply. Imagine we observe the daily change of prices of N stocks (being a part of your portfolio or a specific market index) over last L days. We collect the data in $\mathbf{X}$, the matrix $(N \text{x} L)$. Each of L-long vector lives in an N-dimensional vector space spanned by an orthonormal basis, therefore they all are a linear combination of the set of unit length basic vectors: $\mathbf{BX} = \mathbf{X}$ where a basis $\mathbf{B}$ is the identity matrix $\mathbf{I}$. Within PCA approach we ask a simple question: is there another basis which is a linear combination of the original basis that represents our data set? In other words, we look for a transformation matrix $\mathbf{P}$ acting on $\mathbf{X}$ in order to deliver its re-representation $\mathbf{PX} = \mathbf{X}$. The rows of $\mathbf{P}$ become a set of new basis vectors for expressing the columns of $\mathbf{X}$. This change of basis makes the row vectors of $\mathbf{P}$ in this transformation the **principal components** (PCs) of $\mathbf{X}$.