



Pawel Lachowicz, PhD

Python for Quants

Volume



Fundamentals of Python 3.5
Fundamentals of NumPy
Standard Library

QuantAtRisk eBooks

Python for Quants

Volume I

Paweł Lachowicz, PhD

QuantAtRisk eBooks

To My Mother, for Her Love

Cover design by Paweł Lachowicz, Adobe Photoshop Illustrator

Python for Quants. Volume I.
1st Edition, November 2015

Published by Paweł Lachowicz, ABN 58 495 201 605
Distributed solely by quantatrisk.com

QuantAtRisk eBooks
Sydney, Wrocław

Proofreading: John Hedge

Copyright © 2015 Paweł Lachowicz
All rights reserved.

All programming codes available in this ebook are being released under a BSD license as follows.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

The names QuantAtRisk, QaR, or Paweł Lachowicz may NOT be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER OF PAWEŁ LACHOWICZ AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OF PAWEŁ LACHOWICZ OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Paweł Lachowicz does not take any responsibility for the use and any consequences of the use of the codes available in this e-book in any form.

Table of Contents

Preface	11
About the Author	13
Acknowledgements	15
1. Python for Fearful Beginners.....	17
1.1. Your New Python Flight Manual	17
1.2. Python for Quantitative People	21
1.3. Installing Python	25
1.3.1. Python, Officially	25
1.3.2. Python via Anaconda (recommended)	29
1.4. Using Python	31
1.4.1. Interactive Mode	31
1.4.2. Writing .py Codes	31
1.4.3. Integrated Developments Environments (IDEs)	32
PyCharm	32
PyDev in Eclipse	34
Spyder	37
Rodeo	38
Other IDEs	39
2. Fundamentals of Python.....	41
2.1. Introduction to Mathematics	41
2.1.1. Numbers, Arithmetic, and Logic	41
Integers, Floats, Comments	41
Computations Powered by Python 3.5	43
N-base Number Conversion	44
Strings	45
Booleans	46
If-Elif-If	46

Comparison and Assignment Operators	47
Precedence in Python Arithmetics	48
2.1.2. Import, i.e. "Beam me up, Scotty!"	49
2.1.3. Built-In Exceptions	51
2.1.4. <code>math</code> Module	55
2.1.5. Rounding and Precision	56
2.1.6. Precise Maths with <code>decimal</code> Module	58
2.1.7. Near-Zero Maths	62
2.1.8. <code>fractions</code> and Approximations of Numbers	64
2.1.9. Formatting Numbers for Output	66
2.2. Complex Numbers with <code>cmath</code> Module	71
2.2.1. Complex Algebra	71
2.2.2. Polar Form of z and De Moivre's Theorem	73
2.2.3. Complex-valued Functions	75
References and Further Studies	77
2.3. Lists and Chain Reactions	79
2.3.1. Indexing	81
2.3.2. Constructing the Range	82
2.3.3. Reversed Order	83
2.3.4. Have a Slice of List	85
2.3.5. Nesting and Messing with List's Elements	86
2.3.6. Maths and <code>statistics</code> with Lists	88
2.3.7. More Chain Reactions	94
2.3.8. Lists and Symbolical Computations with <code>sympy</code> Module	98
2.3.9. List Functions and Methods	102
Further Reading	104
2.4. Randomness Built-In	105
2.4.1. From Chaos to Randomness Amongst the Order	105
2.4.2. True Randomness	106
2.4.3. Uniform Distribution and K-S Test	110
2.4.4. Basic Pseudo-Random Number Generator	114
Detecting Pseudo-Randomness with <code>re</code> and <code>collections</code> Modules	115
2.4.5. Mersenne Prime Numbers	123
2.4.6. Randomness of <code>random</code> . Mersenne Twister.	126
Seed and Functions for Random Selection	129
Random Variables from Non-Random Distributions	131
2.4.7. <code>urandom</code>	132
References	133
Further Reading	133

2.5. Beyond the Lists	135
2.5.1. Protected by Tuples	135
Data Processing and Maths of Tuples	135
Methods and Membership	137
Tuple Unpacking	138
Named Tuples	139
2.5.2. Uniqueness of Sets	139
2.5.3. Dictionaries, i.e. Call Your Broker	141
 2.6. Functions	 145
2.6.1. Functions with a Single Argument	146
2.6.2. Multivariable Functions	147
References and Further Studies	149
 3. Fundamentals of NumPy for Quants.....	 151
3.1. In the Matrix of NumPy	151
<i>Note on matplotlib for NumPy</i>	153
3.2. 1D Arrays	155
3.2.1. Types of NumPy Arrays	155
Conversion of Types	156
Verifying 1D Shape	156
More on Type Assignment	157
3.2.2. Indexing and Slicing	157
Basic Use of Boolean Arrays	158
3.2.3. Fundamentals of NaNs and Zeros	159
3.2.4. Independent Copy of NumPy Array	160
3.2.5. 1D Array Flattening and Clipping	161
3.2.6. 1D Special Arrays	163
Array–List–Array	164
3.2.7. Handling Infs	164
3.2.8. Linear and Logarithmic Slicing	165
3.2.9. Quasi-Cloning of Arrays	166
3.3. 2D Arrays	167
3.3.1. Making 2D Arrays Alive	167
3.3.2. Dependent and Independent Sub-Arrays	169
3.3.3. Conditional Scanning	170
3.3.4. Basic Engineering of Array Manipulation	172
3.4. Arrays of Randomness	177
3.4.1. Variables, Well Shook	177
Normal and Uniform	177
Randomness and Monte-Carlo Simulations	179
3.4.2. Randomness from Non-Random Distributions	183

3.5. Sample Statistics with <code>scipy.stats</code> Module	185
3.5.1. Downloading Stock Data from Yahoo! Finance	186
3.5.2. Distribution Fitting. PDF. CDF.	187
3.5.1. Finding Quantiles. Value-at-Risk.	189
3.6. 3D, 4D Arrays, and <i>N</i>-dimensional Space	193
3.6.1. Life in 3D	194
3.6.2. Embedding 2D Arrays in 4D, 5D, 6D	196
3.7. Essential Matrix and Linear Algebra	203
3.7.1. NumPy's ufuncs: Acceleration Built-In	203
3.7.2. Mathematics of ufuncs	205
3.7.3. Algebraic Operations	208
Matrix Transpositions, Addition, Subtraction	208
Matrix Multiplications	209
@ Operator, Matrix Inverse, Multiple Linear Regression	210
Linear Equations	213
Eigenvectors and Principal Component Analysis (PCA) for N-Asset Portfolio	215
3.8. Element-wise Analysis	223
3.8.1. Searching	223
3.8.2. Searching, Replacing, Filtering	225
3.8.3. Masking	227
3.8.4. Any, if Any, How Many, or All?	227
3.8.5. Measures of Central Tendency	229
Appendixes.....	231
A. Recommended Style of Python Coding	231
B. Date and Time	232
C. Replace VBA with Python in Excel	232
D. Your Plan to Master Python in Six Months	233

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 Paweł 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, Paweł 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.

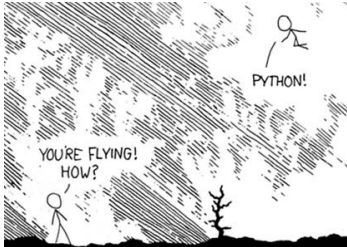
To Les Brown for motivation.

To all who believed in me.

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

1. Python for Fearful Beginners

1.1. Your New Python Flight Manual



Have you heard the news? With Python you can fly! ☺

For many reasons **Python** appears to be a perfect language to kick off your very first programming experience. Programming a machine to do what you want compute is truly an exciting thing! I still remember my very first code in BASIC language written in the early 90s. Nothing to be proud of today but at least I could find the solution for a system of two linear equations just by entering a couple of input parameters from the keyboard. It was really fun, believe me, and I was only twelve. I also learnt that one of the best ways to become a good programmer, if you actually start from the ground up, was to copy and re-run someone's else codes—the **beginner's basic instinct**. This is that moment when your imagination and logical thinking simultaneously kick in. Your mind creates a whole spectrum of ideas guiding you through the modification process of a given code. Instantly you notice that you can derive so many amazing things just by altering a few lines of code or feeding that code with different data sets.

Writing on programming is a **skill**. It is a combination of what is really essential to be grasped and injected into a bloodstream in order to become prepared for more advanced and demanding challenges. If you scan a list of books on Amazon.com devoted to only Python, its number reaches about thirty now. And what I mean by *a book* is a quality book that you can take and learn from it quite a lot. My writing this book-series has nothing to do with making a stunning impression of how great my command of Python is. In fact, I continuously learn something new every time I put my hands on it. What this book is about is more on what a man or a woman of a quantitative mind can do with an attractive programming language to make it work for his or her benefits.

This book series will be useful for all those who would like to start coding in Python straight away. It is intentionally **designed for** quantitative analysts with an interest in finance, algorithmic

trading, and risk management. But again, not limited to those fields whatsoever. It will require some knowledge of advanced maths, statistics, data analysis, time-series, and modelling. By studying this *trilogy* as a whole you will learn everything you need to turn your quantitative problems into a Python code. Efficiently, exhaustively, and effortlessly.

Many great books exist. As a reader you are strongly encouraged to cross-check and study them too. There always be something that I skipped or was not aware of. However, my goal is to take you on the journey through Python; to present a way of thinking you should develop in order to cope with any problem that could be solved by programming. You will be convinced yourself how beautifully Python does the job for you, shortly.

Please excuse my direct and personal tone in this book but I want you to feel comfortable with my style of communication. No stiff and boring writing. Not *another* book on programming language. I will try to make your reading time truly enjoyable.

Even if you have some idea of Python or you are pretty good at it, **you will** benefit from this Volume anyway. Interestingly, the introductory chapters are true mines of knowledge before heading to the Moon. We start small. We aim high!

The most recommended way of reading *Python for Quants* series is from **cover-to-cover**. I designed the text in an experimental way as no one has ever done it before. Just by reading it you learn and connect the dots. Nothing complicated disturbs you or frustrates you. Some people write books in the way that you look for another book, immediately. It will not be the case this time.

I dare to implement the wisdom of Prof. Andrzej A. Zdziarski who was my Master's thesis supervisor in 2003. He taught me the way you should always approach an unknown piece of knowledge and how to read scientific books or publications. He used to say: "When I read something and I do not understand it, I read it again. And again, and again, until I get it." Over the past ten years I turned this philosophy into a style of writing aimed at passing the knowledge on which avoids coming back to the same part of the material too frequently. Have I succeeded doing that? You will tell me at the end of this journey.

How to do "it" in-line?

An alternative way to define a function is via **lambda** keyword:

```
>>> g = lambda x: x**2
>>> f = g(3); f
9
```

Along the text, in the left margin you will find some extra **notes**, **tips**, or pieces of **supplementary Python codes**. It is intended to help you as you progress. Study the variations of all the codes. Try to capture what can be written differently and always think about the ways of how you could do the same — only better?! By doing so, you can bend your learning curve dramatically upwards. The pressure drops with altitude so do not worry: you will enjoy the view anyway!

English is my second language so please forgive me any mistakes. John made a great effort to polish my English. Some sentences I

build differently—on purpose. Katharine Hepburn said: “If you obey all the rules you miss all the fun!” ☺

This book-series may contain some traces of sexual content or references. A strong parental guidance is advised for the readers below the 12th year of age. ☺

I will keep my **writing style** for all Python codes to its absolute minimum, namely, this is a plain code:

```
# An exemplary program in Python
# ver. 1.0.0, by Johnny Cash

from bank import cash as c

wallet = c.amount
print("You have $%.2f in your wallet" % wallet)
```

and the output will be denoted as follows:

```
You have $98.50 in your wallet
```

while for all commands executed in an **interactive mode**:

```
>>> from bank import cash as c
>>> wallet = c.amount
>>> print("You have $%.2f in your wallet" % wallet)
You have $98.50 in your wallet
```

Some examples will be provided in the latter manner just to show you instant results by altering tiny things in the code itself. Study them carefully. All together will form a logical library of **ready-to-use methods** in your daily work with Python.

In addition, across all chapters, I will mark new **commands** or/and **functions** in the margin for your quick reference, for example:

Code 0.0

Given a vector of integer numbers, add 4.1 to each element of it, and sort all elements in ascending order.

```
>>> import numpy
>>> b = numpy.array([3, 2, 1, 4])
>>> a = b + 4.1
>>> a.sort(); a
array([ 5.1,  6.1,  7.1,  8.1])
```

This method comes from my working practice with MATLAB language some ten years ago. I found it pretty useful as a quick look at the **correctness of syntax**. I used to memorise when and where within my notes I made a comment on a specific **function** for the first time. Therefore, next time, I could jump into the corresponding page straight away. I hope the same method will work for you too.

Annotations on new problems or projects will be formulated explicitly as computational **challenges** denoted in the margin by **Code x.x**. Since we challenge ourselves in the process of learning,

a term “challenge” sounds more optimistic than a “problem”. Problems are all around us. Why would you like to read another book delivering you more problems?!

Lastly, some new features of the language need to appear in the text before their official "premiere" in the book. One of such examples is a custom function, e.g.

```
def canwedoit(a, b):
    c = a + b
    return (c/(c-1) < 2.5)

a = 2
b = 4
if canwedoit(a, b):
    print("Yes we can!")

Yes we can!
```

to be described in Section 2.6. Another example could be plotting with a help of `matplotlib` external library. Therefore, those actions are inevitable but making new Python's elements more memorable and user-friendly before their scheduled touchdown.

Having that in mind, let's address the question "Why Python?" first. There must be some "Because...", right?

1.2. Python for Quantitative People

Python language is not exactly like "emerging markets" in finance but one may think of it that way too. Its cornerstones are dated back to the early 80s and linked to its creator in the person of Guido van Rossum. All began at the Old Continent, in the Netherlands. A wishful thinking of making programming more personalised, touchy, and easily learnable. Fast and sophisticated. Attractive and sexy. I hope Guido would be able to confirm those adjectives looking back in time. Certainly he will.

Python displays **complexity** through **simplicity**. Somehow, this feature has been winning people's hearts for the past 20 years. It's readable. It's clear and concise. For sure it's free and downloadable as a part of an open source software. Anyone can contribute to the language itself by Python Enhancement Proposals (PEPs; see more at <https://www.python.org/dev/peps/>). The language blends object-oriented programming with procedural or functional solutions. It is high-level in its design and dynamically typed. You can use it in Windows, Linux, Mac OS X, and within other system environments. Python is a super-glue: it combines the power of simple operations with hacking possibilities across the Web (and not only that!).

It became strongly recognised as a tool for superbly efficient numerical computations leaving the same things in C++ more time consuming. By breaking the "sound barrier" in the form of Global Interpreter Lock (GIL) across many applications, now, Python can do things in a parallel mode. It gains more and more support for Graphical Processing Unit (GPU) computations and recent and ongoing improvements in its best supplementary data-processing/numerical libraries (e.g., [numpy](#), [scipy](#), [pandas](#), [numba](#), [scikit-learn](#), [blaze](#), [dask](#); just to name a few out of a very rich sample) shape the future of **quantitative Python** as extremely attractive to all newcomers. Once the student becomes a master—your own libraries may become accessible to all as an open-source contribution to the Python community. I believe that exactly right now is *your* time to convince yourself how far you can fly with Python on board. This book is your boarding pass.

For the past five years, the language has been gradually introduced into **computational finance**—a brave decision to make things more beautiful. My colleague, Dr. Yves Hilpisch, in his recent book *Python for Finance* (2014) concisely described and summarised the dawn of the Python era in quantitative finance: it all starts from a need, the need for speed and work efficiency. Lots of financial institutions, banks, and hedge funds continuously increase the spending as an investment in modern **technology**. Every four years the amount of data doubles and computers gain computing power in a notable way. The problem of having stronger and better technological facilities requires new ways of building applications and hiring people being able to act quickly and re-program software in a strictly limited window of time. The

global trends in the financial markets as viewed by front-end quants has been dramatically changing.

Dedicated solutions for trading and managing the financial risks on the daily basis commenced "the push" within the software development gently moving core developers from C++/C# platforms towards more user-friendly languages as JavaScript or Python. The key was to recognise the requirements of clients and for the products. The ease in creating computational tools meeting current demands for speed, volume of data, high-frequency, and accuracy of data transmission and processing became of the paramount importance. We entered the century dominated by new ways of creating financial and economical news, reports, and analyses powered by **machine learning** techniques and novel **visualisation methods**. This is a land where solid theoretical foundations of quantitative finance had been asked to comply with the vision of a future world that we will live in.

Sarah Butcher in her article *Goldman Sachs is replacing old school traders with junior finance grads who know how to code* pointed out:

Keen to maintain their employability (...) some senior traders are going back to university and studying masters programs in computer science after eight or nine years in the industry. There's an awareness that people need to keep up to date with new methods. High-frequency traders need to know C++, C# or Python. Systematic traders need to know scripting languages like MATLAB and SQL.

[/businessinsider.com, 16.09.2015/](#)

A serious evolution in the consciousness among those who already work in the field could not be overlooked any longer. In financial markets, the decisions need to be taken swiftly based on the most recent stream of data. The pressure increases. There is no room for human error. Survival in today's financial market place equals information plus unique skill. The better you adopt, the higher chances you've got to make your mark and become a valuable asset to the company.

Python offers a flexibility of programming numerous financial solutions: from time-series processing through real-time analytics down to CPU/GPU-based exhausting Monte-Carlo simulations for, e.g., new derivative products. Some of its core numerical libraries are written and optimised for speed (e.g. [numpy](#)). By putting your hands on the programming abilities of Python you experience an acceleration in turning your models into results delivered on screen nearly instantly. You cannot replace SQL with Python completely. However, MATLAB or R can be overtaken by speed and associated multi-level Python's techniques for data streaming and distributions.

The language would not gain such rapidly expanding popularity if not for the people using it so passionately. Worldwide. The Python community is one of the largest. We observe a linear growth in the private sector of high-tech companies offering dedicated **Web platforms** based on *inter alia* Python engine (see

for example: Quant Platform; <http://www.pythonquants.com/index.html#platform>). The same we observe in the **algo-trading** domain where Python is more than a tool. It is the main tool (e.g. Interactive Brokers; <https://www.interactivebrokers.com>). Leading world **quant bloggers** make use of Python more and more frequently not only as a great educational language but foremost as a demonstration of quantitative applicability for **finance** (see for instance: TuringFinance.com, QuantStart.com, QuantAtRisk.com).

Python captures people's minds going beyond financial territories. It is a superb tool for students, researchers, scientists, IT staff, and Web developers. There is no better way to prove it as by recalling a number of Python conferences and meet-ups organised around the world every year, from Sydney to Seattle.

Thinking about joining, following, and learning Python? Just type and start watching selected YouTube recordings of PyCon, SciPy Con, or PyData conferences. Already hooked? Sign up, fly, and attend one. Don't forget about **For Python Quants Conference** brought by CQF Institute and The Python Quants (<http://fpq.io>). Still hesitating? Don't worry. I've got three words for you:

Buy, my, book...

/Gordon Gekko in
Wall Street Money Never Sleeps 2010/

"Python for Quants" series—my all three ebooks. ☺

References

Yves Hilpisch, 2014, *Python for Finance: Analyze Big Financial Data*, 1st Ed., O'Reilly Media

Points of Interest

Subscribe for weekly newsletters: Python Software Foundation News (<http://pyfound.blogspot.com>) and Python Weekly (<http://www.pythonweekly.com>). Both constitute a great gateway to the latest tectonic movements of Python across the globe.

Visit my Twitter account (<https://twitter.com/quantatrisk>) and scan a whole list of people or organisations I follow. Choose and become the follower of all bodies tweeting on Python daily. You will learn a lot!

1.3. Installing Python

1.3.1. Python, Officially.

No matter if you are a proud owner of a machine running Microsoft Windows, Linux, or Mac OS X as your favourite operation system, you can enjoy Python in every galaxy at the same level of comfort. The official Python download site is <http://www.python.org/download> that offers you: (i) the Python interpreter, (ii) the Standard Library, and (iii) a couple of built-in modules (e.g. `cmath` for complex analysis) in one delivery to your doorsteps. You can treat that as the basic Python platform. You can go this way and install it or make use of one of the ready-to-install-and-manage Python distributions (e.g. Anaconda).

At the same webpage you will be offered with two choices between Python version of 2.7+ and 3.5+ (or older). Surprised? Well, there is an ongoing "battle" of what you should install first and start using as a newcomer to the Python's world. The former version of 2.7+ is still much more popular and has a wide spectrum of additional libraries (open source) while the release of **3.5+** is making its way to become a **new standard** for Python.

If, for some reasons, you need to deal with Python code written for 2.7+ version, the portability to version 3.5+ might require some extra effort and knowledge. Therefore, it is wise to get familiarised with the background information at <https://wiki.python.org/moin/Python2orPython3> referring to both versions if you have not done it so far.

This book will assume codes written and verified to be executable by **Python 3.5**. Underlined by many key world Python advocates, Python 3+ should be thought of as the future of Python. Version 3.5 grew in its richness of new language features and extended applicability since the introduction of Python 3.0.0 in 2008. You can track the change log in 3.5+ evolution at <https://docs.python.org/3.5/whatsnew/changelog.html#python-3-5-0-final>. Guido van Rossum in his keynotes at PyCon 2015 made a point that still ca. 10,000 open-source **batteries** had not been ported from Python 2 to Python 3 (<https://www.youtube.com/watch?v=G-uKNd5TSBw>). It will take time but eventually the Pythonists ought to make version 3 their new home and habit.

It became a custom to refer to all available open-source Python libraries as **batteries**. If you meet with a term *batteries included*, you will know what the Python guys are talking about.

In order to **install Python 3.5+** the official way, first, visit the website <http://www.python.org/download>, download the most appropriate version for your computer system (Windows, Linux, etc.) and install. Simple as that.

Let me show you **an example** of how it looks like within **Mac OS X**'s operating system. It takes a few minutes to make all ready to work. The official installer comes with a nice wizard guiding you painlessly through the whole installation process:



Once accomplished, you can verify the default location of Python 3.5+ by typing:

```
pawel — Python — 72x12
Pawels-MacBook-Pro:~ pawel$ which python3.5
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3.5
Pawels-MacBook-Pro:~ pawel$ python3.5
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

and you are ready to work with Python in its, so-called, interactive mode. In our case the directory storing all additional batteries is:

`/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/`

No matter what is inside it right now, there is always a good way to **install/upgrade** those libraries that you wish to use. You can achieve it by typing, e.g.:

```
$ python3.5 -m pip install --upgrade numpy scipy matplotlib seaborn
numexpr sympy pandas
```

```
Collecting numpy
  Downloading numpy-1.10.1-cp35-cp35m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl (3.7MB)
100% |#####| 3.7MB 128kB/s
Collecting scipy
  Downloading scipy-0.16.1-cp35-cp35m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl (19.7MB)
100% |#####| 19.7MB 27kB/s
Collecting matplotlib
  Downloading matplotlib-1.5.0-cp35-cp35m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl (49.6MB)
```

```

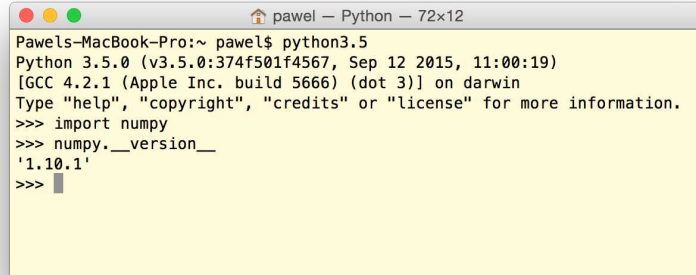
100% |████████████████████████████████████████| 49.6MB 10kB/s
Requirement already up-to-date: seaborn in /Library/Frameworks/
Python.framework/Versions/3.5/lib/python3.5/site-packages
Collecting numexpr
  Downloading numexpr-2.4.6-cp35-cp35m-
macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_
intel.macosx_10_10_x86_64.whl (133kB)
100% |████████████████████████████████████████| 135kB 1.8MB/s
Collecting sympy
  Downloading sympy-0.7.6.1.tar.gz (6.4MB)
100% |████████████████████████████████████████| 6.4MB 78kB/s
Collecting pandas
  Downloading pandas-0.17.0-cp35-cp35m-
macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_
intel.macosx_10_10_x86_64.whl (8.8MB)
100% |████████████████████████████████████████| 8.8MB 61kB/s
Collecting pyparsing!=2.0.0,!=2.0.4,>=1.5.6 (from matplotlib)
  Downloading pyparsing-2.0.5-py2.py3-none-any.whl
Collecting pytz (from matplotlib)
  Downloading pytz-2015.7-py2.py3-none-any.whl (476kB)
100% |████████████████████████████████████████| 479kB 436kB/s
Requirement already up-to-date: python-dateutil in /Library/
Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages
(from matplotlib)
Collecting cycler (from matplotlib)
  Downloading cycler-0.9.0-py2.py3-none-any.whl
Collecting six>=1.5 (from python-dateutil->matplotlib)
  Downloading six-1.10.0-py2.py3-none-any.whl
Installing collected packages: numpy, scipy, pyparsing, pytz, six,
cycler, matplotlib, numexpr, sympy, pandas
Found existing installation: numpy 1.9.2
Uninstalling numpy-1.9.2:
  Successfully uninstalled numpy-1.9.2
Found existing installation: scipy 0.16.0
Uninstalling scipy-0.16.0:
  Successfully uninstalled scipy-0.16.0
Found existing installation: pyparsing 2.0.3
Uninstalling pyparsing-2.0.3:
  Successfully uninstalled pyparsing-2.0.3
Found existing installation: pytz 2015.4
Uninstalling pytz-2015.4:
  Successfully uninstalled pytz-2015.4
Found existing installation: six 1.9.0
Uninstalling six-1.9.0:
  Successfully uninstalled six-1.9.0
Found existing installation: matplotlib 1.4.3
Uninstalling matplotlib-1.4.3:
  Successfully uninstalled matplotlib-1.4.3
Found existing installation: numexpr 2.4.3
Uninstalling numexpr-2.4.3:
  Successfully uninstalled numexpr-2.4.3
Found existing installation: sympy 0.7.6
Uninstalling sympy-0.7.6:
  Successfully uninstalled sympy-0.7.6
Running setup.py install for sympy
Found existing installation: pandas 0.16.2
Uninstalling pandas-0.16.2:
  Successfully uninstalled pandas-0.16.2
Successfully installed cycler-0.9.0 matplotlib-1.5.0 numexpr-2.4.6
numpy-1.10.1 pandas-0.17.0 pyparsing-2.0.5 pytz-2015.7 scipy-0.16.1
six-1.10.0 sympy-0.7.6.1

```

what initiates a download process or an update of [numpy](#), [scipy](#), [matplotlib](#), [seaborn](#), [numexpr](#), [sympy](#), [pandas](#) libraries (mentioned in this book). You may find that the installation of other modules requires some other (supporting) libraries. If so, use the same command and refer to the names of missing modules. If lost, look for solutions on the Web. There were people going the same

way before you stepped on that path. Someone left the clues. For sure. Find them.

Any time you wish to **check the version** of any **library** linked to your Python 3.5+, type, e.g.:



```
Pawel-MacBook-Pro:~ pawel$ python3.5
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> numpy.__version__
'1.10.1'
>>>
```

where `modulename.__version__` command requires the import of the module in first place. On the other hand, if you would like to obtain a full list of all modules that are at your disposal, write and execute the following piece of code:

```
>>> import pip
>>> lst = sorted(["%s %s" % (i.key, i.version) for i in
                  pip.get_installed_distributions()])
>>> for elem in lst: print(elem)
...     press Enter

ccycler 0.9.0
gnureadline 6.3.3
ipython 3.2.1
jinja2 2.8
jsonschema 2.5.1
markupsafe 0.23
matplotlib 1.5.0
nose 1.3.7
numexpr 2.4.6
numpy 1.10.1
pandas 0.17.0
patsy 0.4.0
pep8 1.6.2
pip 7.1.2
pyparsing 2.0.5
python-dateutil 2.4.2
pytz 2015.7
pyzmq 14.7.0
requests 2.7.0
scikit-learn 0.16.1
scipy 0.16.1
seaborn 0.6.0
setuptools 18.2
six 1.10.0
statsmodels 0.6.1
sympy 0.7.6.1
theano 0.7.0
tornado 4.2.1
urllib3 1.11
zmq 0.0.0
```

Update of Anaconda

From time to time it is advised to update your Anaconda Python distribution. It makes sure that all libraries are up to date. You can do it from the shell typing two commands:

```
$ conda update conda
$ conda update anaconda
```

Make sure that `pip` module has been installed prior to your action.

1.3.2. Python via Anaconda (recommended)

Installing Python via the "Official Way" can be more problematic. The devil resides in all the links (requirements) between different libraries. It takes a lot of time to install/update them all after every new version comes out.

A quick fix to that roadblock is the installation of Python 3.5 via **Anaconda Python distribution** provided by Continuum Analytics: <http://continuum.io/anaconda>. It is a completely free enterprise-ready distribution for large-scale data processing, predictive analytics, and scientific computing. It contains, all-in-one, over 300 most popular Python and third-party packages. Therefore, you do not need to worry about looking for any of them separately across the Web and installing on your own. Anaconda does it for you and keeps its eye on their most recent updates. The full list of extra modules (libraries) can be found here: <http://docs.continuum.io/anaconda/pkg-docs.html>. Anaconda Python 3.5 supports 317 packages (not every one has been included in the installer). Interestingly, in this book, we will make use of less than ten of them. That should give you an idea on the power that Python gains with every new update and uninterrupted Python community contributions.

The **installation process** has been greatly simplified since version 3.4. First, visit <https://www.continuum.io/downloads>. Second, choose the correct version for your computer (Mac OS X, Windows, Linux) and 32 or 64-bit release. Click and follow the graphical installer and its wisdom:

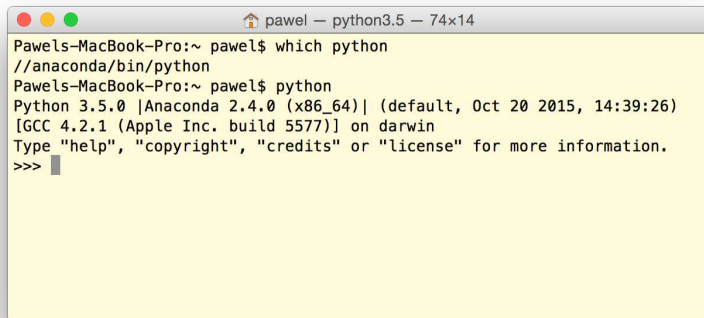
PYTHON 2.7	PYTHON 3.5
<div>Mac OS X 64-bit Graphical Installer</div> <div>274M (OS X 10.7 or higher)</div> <div>Mac OS X 64-bit Command-Line Installer</div> <div>239M (OS X 10.5 or higher)</div>	<div>Mac OS X 64-bit Graphical Installer</div> <div>267M (OS X 10.7 or higher)</div> <div>Mac OS X 64-bit Command-Line Installer</div> <div>233M (OS X 10.5 or higher)</div>

Again, in case of Mac OS X as an example, the downloadable file with Python 3.5 is 280.4 MB large and you will be asked to point at the destination (hard drive) of your choice.

Once the Anaconda 2.4+ installer has terminated with success, the PATH system variable within your `~/.bash_profile` file has also been updated:

```
# added by Anaconda3 2.4.0 installer
export PATH="//anaconda/bin:$PATH"
```

In this case, executing:

A screenshot of a macOS Terminal window titled 'pawel - python3.5 - 74x14'. The window has a yellow background. The text inside shows the user running 'which python' and 'python' commands. The output of 'python' shows the Python 3.5.0 environment with Anaconda 2.4.0, including the GCC version and build information. The prompt '>>>' is visible at the end of the last line.

```
Pawels-MacBook-Pro:~ pawel$ which python
//anaconda/bin/python
Pawels-MacBook-Pro:~ pawel$ python
Python 3.5.0 |Anaconda 2.4.0 (x86_64)| (default, Oct 20 2015, 14:39:26)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

in a "freshly" opened Terminal, a simple command:

```
$ python
```

enters Python 3.5 in its interactive mode. To **exit** the Python shell and come back to the bash, click **Control+D** or type:

```
>>> exit()
```

as a Python command.

Having that behind you, you are ready to use Python!

1.4. Using Python

1.4.1. Interactive Mode

Python is an interpreted language. That means it reads every line and “interprets” or “executes” it as specified and coded by a programmer. There are two ways you can use your Python interpreter. The first one is, as already mentioned earlier, the **interactive mode**. You launch Python in Terminal (Mac/Linux) or from the command prompt in Windows and start coding immediately. For example,

```
>>> x = 5
>>> y = x**2 # rise x to the power of 2
>>> print(y)
25
>>> for i in range(5): press Enter
...     print(i)      press Space 4x before 'print(i)'
...                  press Enter 2x to see the results
0
1
2
3
4
```

where sign >>> denotes a Python prompt sign and ... markers stand for the continuation of the Python structure of the **for** loop. The interactive mode is a great place to start your adventure with programming. It can be used directly to test results or the correctness of the code itself. One tiny advantage can be attributed to the possibility of entering a variable’s value from the keyboard, for instance:

```
raw_input >>> name = "John"
>>> mood = raw_input("How are you today %s? " % name)
How are you today John? Never been better!
>>> print(mood)
Never been better!
```

This option is nice if you really really have no clue of what to do with Python or wish to recall the good times of BASIC language where RAM was 64 kBytes on your Commodore 64 and nothing could be stored on the hard drive. Today, it is highly impractical to write the code that waits for a raw input. Moreover, the command of **raw_input** only works within Python 2.7.10. Therefore you see... Good times are gone.

1.4.2. Writing .py Codes

Dealing with complicated and complex programming problems both of quantitative and big-data nature requires writing more than a few lines of code. It can be re-run many times and modified freely. Programmers use a variety of solutions here ranging from simple text editors to more advanced and specially dedicated Integrated Development Environments (IDEs). For the former, you can write your Python code in a **plain text file**, e.g.

```
from cmath import sqrt
x = -1
y = sqrt(x)
print(y)
```

save it under the name of `isqrt.py` and run the code from the level of your operating system:

```
$ python isqrt.py
```

to see the outcome:

```
1j
```

i.e. a complex number as the result of taking a square root of -1. I do not have to tell you what `.py` stands for, right?

The second option seems to be more mature. Modern IDEs are stand-alone applications designed to facilitate computer programmers and their work in software development. A typical **IDE** consists of a source code editor, build automation tools, and debugger. The idea is to create a separate software project in IDE of your choice, keep all files of the project in one place and link them together (if required).

1.4.3. Integrated Development Environments for Python

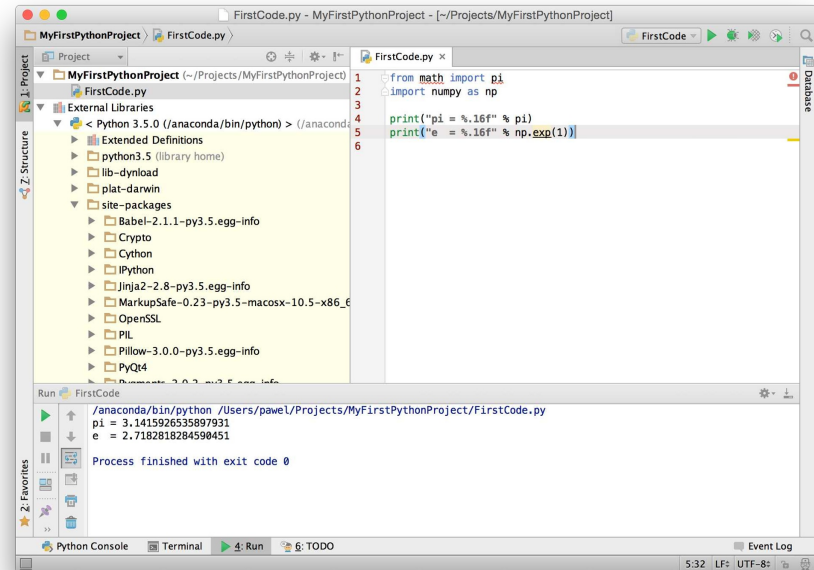
PyCharm

For the need of writing and running your Python codes you may choose among a few solid IDEs accessible across the Web. Here, I would recommend you to download and install **PyCharm** as your favourite IDE. Why? Well, it is elegant, free, and sufficient to cope with complex Python codes as you progress in time. It also offers an easy way to switch between Python interpreters (2.7.x and 3.4.x) for your code portability checks. You can **download PyCharm** for Mac OS X, Linux, and Windows visiting <https://www.jetbrains.com/pycharm/download/>. You can choose between PyCharm Community Edition (completely free) or a paid (recommended) Professional version.

When run for the first time, you will be welcomed with a *charm*. Click on “Create New Project” and when a window pops up define the project name, choose location, and select a desired interpreter (e.g. Anaconda’s python3.5.0, if already installed).

By clicking “OK” you enter into proper PyCharm’s interactive development environment which, from now on, will be your playground (see the figure on the next page).

The main workspace is, by default, divided into three panels: Project Files and External Libraries (left), Source Code Editor (right), and if you press `control+option+r` (Mac) you will see the results from executing your Python code (bottom).



Working within PyCharm project is fairly easy. From a main menu select “New...” and next “Python file”. In our example (above) it caused the opening of the source code editor’s field where we wrote a short Python program aimed at displaying on the screen the value of π and e . In the same way you can add more .py files to your project and switch between them or open and edit some plain text files (under the same source code editor) containing data (e.g. in the .csv file format).

If you click on and expand the item of site-packages (in left panel) you have direct access and preview of all modules recognisable by and linked to your /anaconda/bin/python interpreter.

In our example, above, we imported a numerical library of NumPy (more on NumPy in Chapter 3) by typing a command:

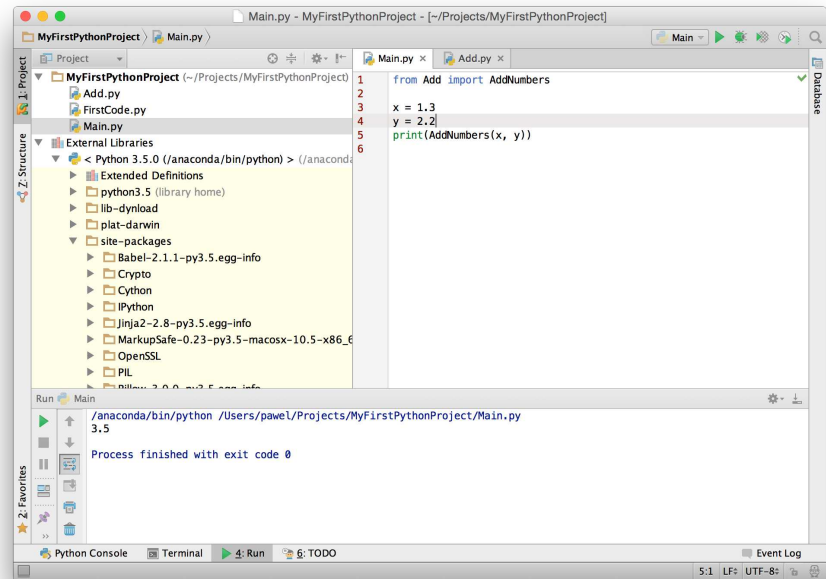
```
import numpy as np
```

It is obvious that importing a non-existing library or making a reference to a .py file that is not present in the Project directory will cause an error.

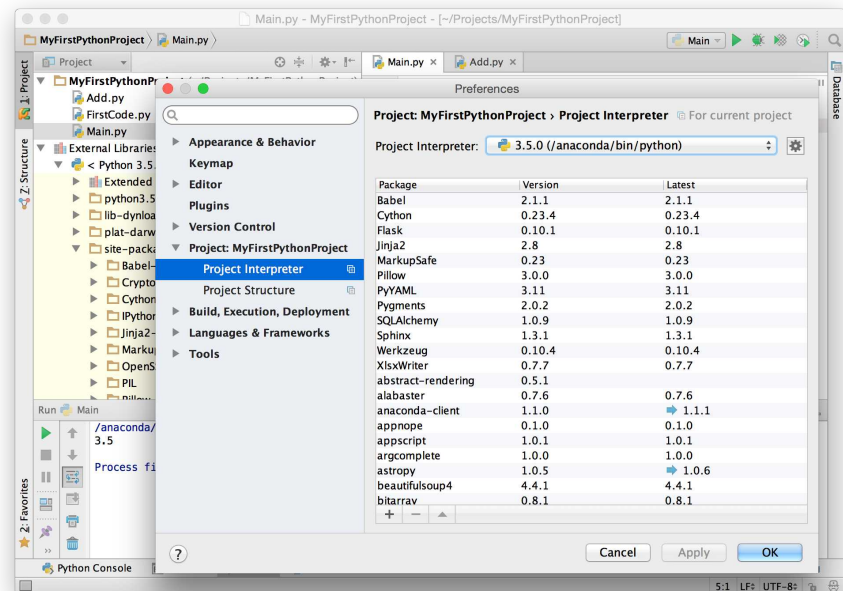
It is too early to mention that but if you work with two files within your project, say, **Main.py** and **Add.py**, the latter storing a simple Python function, **AddNumbers**, designed to add two numbers and return the result,

```
def AddNumbers(x, y):
    return x+y
```

in your main code of **Main.py** you can import **AddNumbers** function and use it in your calculation as follows:



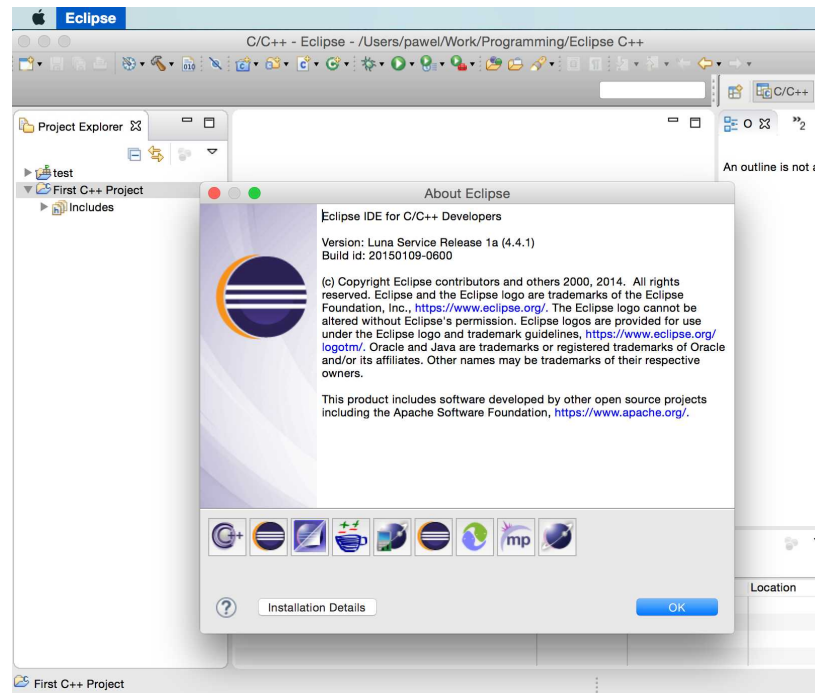
A full list of available modules plus the possibility to edit (the link/system path) of your current/desired Python interpreter is accessible by reaching Preferences > Project: *Name* > Python Interpreter from the main menu, e.g.:



PyDev in Eclipse

Some of Java or C/C++ software developers are used to working within **Eclipse** IDE. However not many are aware that it is possible to build Python projects in Eclipse too. The latter is available making use of **PyDev** module.

If you want to setup Eclipse as your preferable IDE for Python, first download and install *Eclipse IDE for C/C++ Developers* from <https://eclipse.org/downloads/>. In Mac OS X, as an example, you will see:



when your Eclipse IDE is launched.

Now, to turn it into Python factory, you need to make sure you have installed the newest version of **Java SE Runtime Environment** (8 or higher) from Oracle's page of <http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html> (e.g. `jre-8u66-macosx-x64.tar.gz` file for Mac OS X, etc.). Next, do the same but for **Java SE Development Kit 8** which is downloadable from <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html> webpage. These two URL addresses may change with a new release of JRE and JDK but at least you know that oracle.com website is the right place to look for their new versions.

Having that, adding **PyDev** into Eclipse IDE for C/C++ Developers requires following an illustrated manual provided at http://pydev.org/manual_101_install.html as an example for Microsoft Windows users. The process is identical for Mac and Linux programmers. Make sure you reset your computer to allow Eclipse link PyDev and set it up automatically.

From this moment you can begin your new Python project by selecting from the main menu `New > PyDev Project` and specifying its path and interpreter. `New > File` will create a first plain file (e.g. you can name it as `FirstCode.py`) containing the following listing:

[illegible]

Both IDEs, PyCharm and Eclipse, are standalone pillars of great programming software designed around professional developers. If you touch them for the first time you can be overwhelmed with

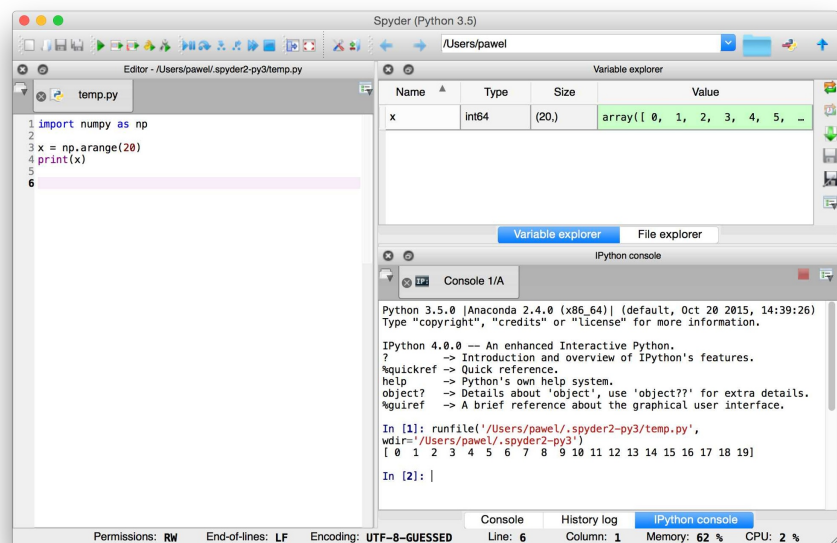
a number of submenus and hundreds of options and the settings available. Spend some time reading about them all. Watch some tutorials on YouTube.

As a challenge (or homework) try to work out the correct steps in the process of **debugging** of the Python code. Bugs will appear. Finding them and fixing your programs is a skill. With the right tools at your disposal you will kill the fear and calm down your anxiety.

Spyder

Every Anaconda distribution arrives equipped with Spyder. According to <http://spyder-ide.blogspot.com>, **Spyder** is a free open-source development environment for the Python programming language providing MATLAB-like features in a simple and light-weighted software, available for all major platforms (Windows, Linux, MacOS X).

You can install it independently following the procedures provided at <https://pythonhosted.org/spyder/installation.html>. The main screen of Spyder appears to resemble PyCharm, i.e.



By default, its console is set to IPython. The code in a current editor, when run, returns the output in the IPython shell as in the example above. Spyder is very easy in use and its unique feature that has been winning people's heart is a MATLAB-like **variable explorer** (e.g. still missing in PyCharm!).

In long-haul, Spyder seems to display delays between code executions and during frequent interactions with the IDE itself.

Rodeo

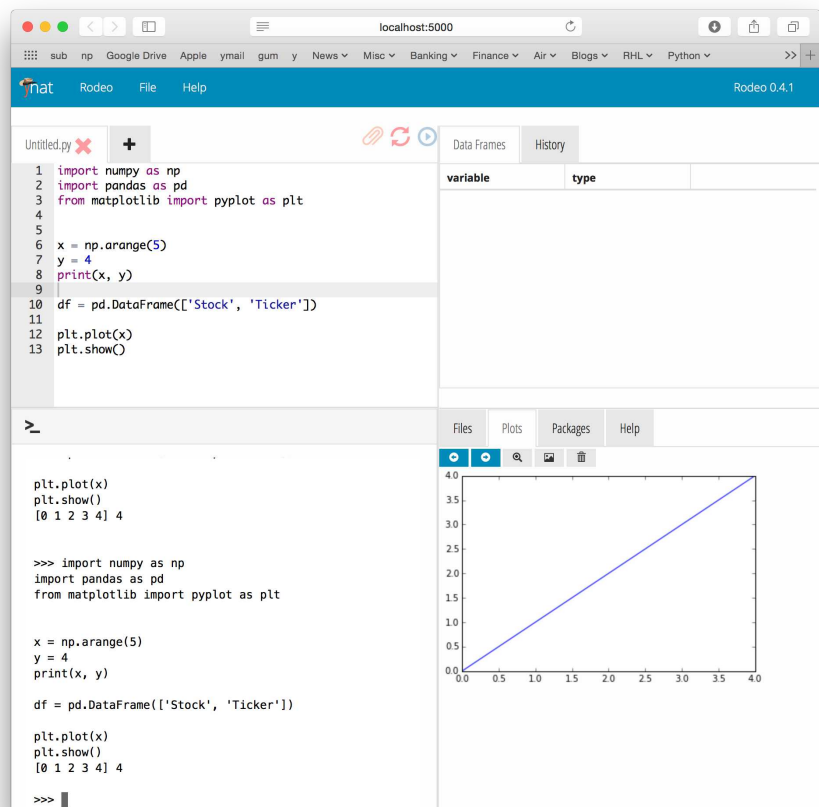
One alternative to Spyder may become **Rodeo** brought to you by yhat (<http://blog.yhathq.com/posts/introducing-rodeo.html>).

Rodeo is an IDE that is built expressly for doing data science in Python. Think of it as a light weight alternative to the IPython Notebook. Given the minimal requirements, i.e. a terminal with autocomplete, a text editor, and the ability to see plots—Rodeo is a project under construction attracting a fair circle of least-demanding Python programmers who wish to work with Python quickly and not in a sophisticated way.

You can install and run it in the Terminal by typing:

```
$ pip3.5 install rodeo
$ rodeo .
```

Rodeo launches itself in a browser:



and provides a user with a minimalistic working environment. Zen of Python says *Simple is better than complex*. If the IDEs could follow PEP20 (<https://www.python.org/dev/peps/pep-0020/>), Rodeo would gain more interest. For sure.

Other IDEs

As one goes deeper into the Web, you discover more possibilities for running your projects. Those ones that I mentioned here are plausible alternatives you may wish to consider.

The first one is **NetBeans** (<https://netbeans.org>) and its IDE's Python plugin (<http://plugins.netbeans.org/plugin/56795/python4netbeans802>). But why would you go for a major player that plays in the different leagues (Java, HTML5, C++)? The second one would be **KomodoIDE** for Python (<http://komodoide.com/python/>). 21 day free trial, nice layout; simple. The third and forth option you've got is the same class of plain but tuned code editor with the Python starters: **TextMate** (<http://macromates.com>; for Mac only) and **Sublime Text 2** (<http://www.sublimetext.com/2>). I loved the latter because it is so clean and sexy, elegant and intelligent. The fifth could be **KDevelop** (<https://www.kdevelop.org>) aimed for Linux and Mac users. Somehow, oldish today.

What attracts more attention, especially for MS Windows fans using Visual Studio, is **Python Tools for Visual Studio** (<https://www.visualstudio.com/en-us/features/python-vs.aspx>). Their latest overview video on YouTube (<https://www.youtube.com/watch?v=GtN73hfXsXM>) may redirect your curiosity towards this brand new product.

You are also strongly encouraged to explore the solution of IPython as well as **IronPython** (Python for the .NET Framework; <http://ironpython.net>) and **Cython** (a superset of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes; see <http://cython.org>).

In this book we will skip a discussion of **IPython** (<http://ipython.org>) that offers a supplementary workplace in the form of so-called, **IPython Notebooks**. It will be introduced fully within *Python for Quants. Volume II*, when we will be working with financial data and time-series employing **pandas** module as the main engine. For many reasons it is a very convenient way of starting coding however let's keep our focus clear: The Fundamentals of Python.

Shall we?

2. Fundamentals of Python

2.1. Introduction to Mathematics

Do you remember your first maths lesson at primary school? I guess it was a while ago. Let me rephrase my question: Would you like to enjoy your very “first time” once again? The good news is that you can do *it* with Python anytime you want. When you learn programming in a new language, somehow, you always start from the same level: the level of fundamental **mathematics** and the concepts of **variables** within a computer language. This is where our journey begins.

2.1.1. Numbers, Arithmetic, and Logic

Integers, Floats, Comments

It is an excessive extravagance to install and use Python as a simple calculator. However, it is possible: to add, subtract, perform any calculations that follows the rules of **arithmetic**. Run the following code using Python 2.7.10 interpreter first.

Code 2.1

Basic arithmetical operations in Python 2.7.10 vs Python 3.5.

Comments in Python

If you would like to add some lines of comments to your code, there are three ways to do it:

- (1) a full line comment:

```
# girls just wanna have
# fun
```
- (2) a comment as a 'suffix':

```
sex=True # go Tiger, go!
```
- (3) a block-comment:

```
x=7
...
x=x+1
...
print(x)
```

will ignore all lines between triple quotes returning:

7

```
# Here we go!
x = 2
y = 9
z = 13.513

a = x + x
b = (x - 3*x)*7.32 + x/y
c = y**(1/x)          # ** stands for 'rise to the power of'
txt = 'is derived'    # string

print(a)
print("b= ", b        # runnable only in Python 2.7+
print('The value of a = %g while c= %f %s' % (a, c, txt))
```

Here we defined three variables **x**, **y**, and **z** and assigned some random values. Next, we performed simple arithmetical operations

and the derived values of right-hand side expressions are assigned to a set of three new variables, namely, **a**, **b**, and **c**. Finally, using a Python command of `print` we check the outcomes:

```
4
b= -29.28
The value of a = 4 while c= 1.000000 is derived
```

This is your first lesson on what Python assumes about your style of coding. It resembles a never-ending discussion among women: *does the size matter?* I don't know. Just ask them! But all we know is that for Python the **type** of its variables **does matter**. Have a closer look at variable **a**. It returns 4 as expected. How about **b**? It displays a number with two-digit precision. Why there are no decimal numbers in case of `print(a)`? Good question! Python deduces, based on the input of **x** that **a** must be of the same type, i.e. to be an **integer** number.

When at least one number in the arithmetic statement is explicitly defined with some decimal precision then the result of all operations will be of the **floating-point** type (or **float**, in short). That is why **b** returns -29.28 instead of -29.

Do not be misled by the operation of division in case of **c** variable. Consider the following code:

```
>>> print(y/x)
4                                     # if run in Python 2.7.10
```

Therefore memorise the rule:

```
>>> print(9.0/2.0, 9.0/2, 9./2, 9/2.0, 9/2.)
4.5 4.5 4.5 4.5 4.5
```

Anytime, if you lose a track of types, feel free to check the type of your variables making use of `type`, e.g:

```
type()
>>> print(type(a), type(b), type(c))
(<type 'int'>, <type 'float'>, <type 'int'>)
```

Now, let's have a look at **c** once again. Consider the code:

```
float()
int()
print(c)
c = float(y**(1/float(x)))
print(c)
c_i = int(c)
print(c_i)
```

returning in Python 2.7.10 the following:

```
1
3.0
3
```

The previous value of **c** was equal 1 of the integer type. By applying the external function of `float()` acting upon the same expression we force **c** to be of the float type. A transition from float to integer numbers we achieved with `int()` function.

sys.maxint

If you are using 32-bit Python runtime the full range for integers will be between $[-2e31, +2e31]$. For the sake of curiosity you can find the maximal integer value for the current platform as follows:

```
>>> import sys
>>> print("%.5e3" %
          float(sys.maxint))
9.22337e+183
```

sys.float_info

Similarly, a rich information on the current floating-point number representation you find by typing:

```
>>> sys.float_info
sys.float_info(max=1.797693134
8623157e+308, max_exp=1024,
max_10_exp=308,
min=2.2250738585072014e-308,
min_exp=-1021,
min_10_exp=-307, dig=15,
mant_dig=53,
epsilon=2.220446049250313e-16,
radix=2, rounds=1)
```

Python handles **long integers** very well. When displayed in the interactive mode, a letter “L” may be added to mark the long type:

```
>>> p = 219
>>> r = 3**p
>>> r
30871290436659589031997859934150465556725743116640932942113508973
2346509723631627417356013073234099809467L

>>> type(r)
<type 'long'>
```

but a direct comparison with the float-like version returns negative result:

```
>>> s = 3.**float(p)
>>> s == y # is s equal y ?
False      # the same result is obtained for s = pow(3.0, 219.0)
```

due to the difference in number representation in the memory. Note that it is easier to store a correct long integer number than its floating-point equivalent though it is not so obvious. We will discuss the precision of mathematical operations with floats, shortly.

If for some reasons you would like to obtain information on the number of bits that any given **integer** consumes in memory, simply use:

.bit_length()
It works with integer and long
integers only.

```
>>> r_bytes = r.bit_length()
348
```

Computations Powered by Python 3.5

Rerunning the [Code 2.1](#) and all following operations (described above) in Python 3.5 will deliver completely different results. A classical test entitled *Are we working with Python 3?* is a two-liner:

```
>>> print 5
>>> print 5/2
```

For the former command, if your Python returns an error:

```
File "<stdin>", line 1
print 5
    ^
SyntaxError: Missing parentheses in call to 'print'
```

and for the latter the result of a correct form, i.e.

```
2.5
```

you can be sure that, in general, you’re dealing with Python 3.x. The command of **print** requires the use of parentheses (round brackets) that is optional in Python 2.x. The division of integer by integer in Python 3.x has been programmed to perform the float-point type arithmetic by default.

Floor Division
Additional arithmetic operation in Python is a floor division, i.e. the division of operands where the result is the quotient in which the digits after the decimal point are removed:
>>> 5/2
2.5
>>> 5//2
2
however
>>> 5.0//2
2.0

Pay attention to your results if you use Python 2.7.x. A lack of care in handling integer-involved operations could be the source of unnecessary mistakes and frustration.

N-base Number Conversion

This section would leave some of you disappointed if we had skipped a mention of **conversion** of the integer numbers from “our” decimal system **to n-base systems** and **back**. As you may guess, Python delivers handy solutions also in the field.

For the **binary** representation of decimal numbers we have a whole spectrum of ready-to-use tools. Analyse the following cases:

int(x, n)
The same function of `int()` can be used to convert any number `x` specified in `n`-base system.

format(x, code)
`code` is 'b', 'o', and 'x' for binary, octal, and hexadecimal integers, respectively.

```
>>> t = 10
>>> b = bin(t); b
'0b1010' # 0b is added as a prefix for binaries
>>> type(b)
<class 'str'>

>>> B = bin(t+t); B # never: b+b nor bin(t)+bin(t)
'0b10100'
>>> int(B, 2)
20

>>> B = bin(t+int(6./2.*4.)); int(B, 2)
22

>>> x = format(t+int(6./2*4), 'b') # use format with 'b'
>>> x
'10110'
>>> type(x)
<class 'str'>
```

hex()

Similarly to decimal-to-binary number conversion, we may use `oct()` and `hex()` functions to achieve the same step for **octal** and **hexadecimal** integers, respectively, e.g.

oct()

```
>>> x = oct(2015); x # 0o is added as a prefix for octals
'0o3737'
>>> y=int(x, 8); y
2015

>>> print("%g in hex is %s" % (int(hex(y), 16), format(y,'x')))
>>> 2015 in hex is 7df
```

The abovementioned methods limit our options down to three of the most frequently used systems. For more advanced conversions we need to program a dedicated solution with a little help of a special custom function, e.g.:

Breaking the Line
If the line of your Python code exceeds 80 characters or it needs to be broken at any point—use backslash. The use of a good IDE software, for example PyCharm, helps you to follow the rules defined by PEP8 (see Appendix).

```
def int2base(num, b):
    numerals = "0123456789abcdefghijklmnopqrstuvwxyz"
    return ((num == 0) and numerals[0]) or \
        (int2base(num//b, b).lstrip(
            numerals[0]) + numerals[num % b])

x = 2015
x2 = int2base(x, 2)
x8 = int2base(x, 8)
x16 = int2base(x, 16)
x24 = int2base(x, 24)

print("binary %12s = %g decimal" % (x2,int(x2, 2)))
```

```
print("octal  %12s = %g decimal" % (x8,int(x8, 8)))
print("hex    %12s = %g decimal" % (x16,int(x16, 16)))
print("24     %12s = %g decimal" % (x24,int(x24, 24)))
```

The output produced by our program is:

```
binary 11111011111 = 2015 decimal
octal   3737 = 2015 decimal
hex     7df = 2015 decimal
24      3bn = 2015 decimal
```

We will cover the construction of custom functions soon. As for now, try to grasp the beauty and elegance of blending a supplementary function with the main code and its calling.

Interestingly, `int(x24, 24)` handles the conversion back to decimal system from 24-base system quite effortlessly. Pay attention that the function of `int2base` has a limitation as it should raise an error if not $(2 \leq \text{base} \leq 36)$.

Strings

In [Code 2.1](#) we defined a `txt` variable. A quick check reveals that:

```
>>> type(txt)
<class 'str'>
```

what stands for **string**. Any non-numerical or semi-numerical sequence of characters embedded in single (`'`) or double quotes (`"`) is treated as a string. Do you remember a Hollywood movie “No Strings Attached”? Just by placing that title between two double quotes I created a string (for more on `print` and `format` and their use in formatting numbers for output see [Section 2.1.9](#)).

There exists a more formal way of checking whether any given variable is of the string type or not. Analyse the following code:

Code 2.2

`str()`

`if-elif-else`

Testing multiple logical conditions in Python never been easier. Don't forget about colon (`:`) at the end. That's the part of the syntax. Embedding conditions-to-check in round brackets helps Python in their evaluation (and order).

```
test = False
x = 3
y = 4
k = y**(1/float(x))
k = float(k)

c = str(k)
print(c)
print(type(c))

if isinstance(c, int):
    print("c is an integer")
elif isinstance(c, str):
    print("c is a string")
    test = True
else:
    print("c is probably of a float type")
```

displaying:

```
1.5874010519681994
<class 'str'>
c is a string
```

`isinstance()`

In the beginning we force an integer operation using **y** and **x** to be **c** converted to the float via **k**. Applying `str` function we convert float-to-string and next with the help of `isinstance` function we compare **c** with `int` or `str` class-type in order to determine its type.

Boolean

Python, as nearly all modern computer language, offers a logical type for its variables: a **boolean** type. We created such variable (**test**) in [Code 2.2](#). Its value is either `True` or `False`. As you can convince yourself in the interactive mode, `True` corresponds to 1 whereas `False` to 0. Examine the following:

A False Truth

If you work with Python 2.7.10 you may be trapped by a false side of truth:

```
>>> True = False
>>> False = True
>>> True
False
>>> False
False
```

Fortunately, Python 3.5 prevents such behaviour returning:

SyntaxError: can't assign to keyword

```
>>> test = True
>>> type(test)
<class 'bool'>
>>> test2 = False
>>> test*3 + test2
3
```

It is important to remember that spelling of `true` or `false` is incorrect and Python returns a `NameError`:

```
>>> test = false
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'false' is not defined
```

Therefore *choose your words wisely Spartan as they may be your last!*

If-Elif-Else

As we speak about logic, it is difficult to omit a few additional comments on `if-elif-else` construction. Women say: “**If** I have a headache today we won’t have sex *or else* (**elif**) you better have protection *otherwise* (**else**) leave me alone!” The same logical thinking can be coded in Python for any statement under examination. If its value has been evaluated to either `True` or `False`, in first place, a body of indented code beneath **if** is executed. Otherwise the alternatives are considered instead.

A classical way of the representation of the following function of $f(n)$, defined as,

$$f(n) = \begin{cases} n/3 & \text{if } n \geq 0 \\ -(n+5)/6 & \text{if } n < 0 \end{cases}$$

would be:

```
n = 2.1 # a random value
if(n >= 0):
    f = n/3
else:
    f = -(n+5)/6

print(f)
```

since we have only two options to consider. How would you go about that one...?

$$f(x) = \begin{cases} \sqrt{x} & \text{if } x < 0 \\ 4 \cos(2\pi x) & \text{if } x \geq 0 \text{ and } x < 1 \\ x \log_2[e^x \tanh(x)] & \text{if } x \geq 1 \end{cases}$$

Some of the functions which we need to use here can be imported directly from the Python's Standard Library. In our case that requires uploading of the `math` module. In addition, we need a function which can handle a square root for negative numbers—`cmath`. An exemplary code defining $f(x)$ could take the following form:

Code 2.3

```
from math import cos, log, exp, tanh
from cmath import sqrt as csqrt

x = -4.11    # a random value

if(x < 0):
    f = csqrt(x)
elif((x >= 0) and (x < 1)):
    f = 4*cos(2*pi*x)
else:
    f = x*log((exp(x)*tanh(x)),2)

print(f)
```

The library of `cmath` is a special collection of functions designed for complex analysis. We will cover its usefulness in Section 2.2.

Between `if` and `else` you can place more than one `elif`. There is even no need to end a block with `else`. Keep that in mind. Test on your own.

Comparison and Assignment Operators

The reason lot of people fall for Python is its English-like way of writing the code. The best example showing and proving that are Python **logical expressions**. Every day, we human, process information without particularly thinking about the logic of sentences or statements that reach our ears. *If* something is *not* right for you, you *do* something else *or* look for help *and* support. Programming in Python, intuitively, follows the same rules when it comes to the construction of logical conditions.

Try to assimilate the way of expressing the following logical statements to be evaluated to boolean values. As homework check the outcomes in the interactive mode by yourself:

```
>>> a = 2
>>> b = 9
>>> c = -20
>>> not(a <= 0) or (b == (10-1))
?
>>> (c != -20)
?
```

```
>>> (c == 20)
?
>>> ((a+b > 10) or (not(c-9) >= 30)) and (not(a) < 0)
?
```

It is good habit is to group your conditions in round brackets in order to determine the priority for their evaluation. Some IDEs may suggest the use of parentheses to be omitted however, personally I do think their inclusion improves the overall logic and readability of your code. *Sometimes, it's good to break the rules. The PEP8 rules.*

In terms of simple arithmetic keep in mind some handy shortcuts engaging so-called **assignment operators**. You may know them already from C or C++ or elsewhere:

```
>>> a = 2
>>> a += 2;           # equivalent to a = a+2
4
>>> a -= 6; a         # equivalent to a = a-6
-2
>>> a *= 13; a        # equivalent to a = a*13
-26
>>> a /= 2; a         # equivalent to a = a/2
-13.0
>>> a += 20; a %= 5; a # equivalent to a = a%5 (modulo)
2.0
>>> a -= 6; a //= 5; a # floor division
-1.0
```

Precedence in Python Arithmetic

What is worth noticing is the **precedence** for all Python operators, i.e. the order the interpreter will evaluate the expression if no parentheses have been used. Here is a complete list of operators from the highest to the lowest precedence:

Operator	Description
**	Exponentiation
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise AND
^	Bitwise exclusive OR and regular OR
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

We will cover some types of Python operators from that list later in the book. Again, the use of parentheses forces higher priority.

2.1.2. Import, i.e. “Beam me up, Scotty”

Have a look again at [Code 2.3](#) where we made use of `cos(expr)`, `log(expr, base)`, and `tanh(expr)` functions. Before we explore all goodies of the `math` module, it is an excellent time to make a side note on the `import` function in Python.

As you have probably already noticed, Python allows us to import existing functions from various libraries (modules) in a very flexible way. `math` and `cmath` is a good starting point towards exploration of all existing variants. In order to get an access to all functions included in any module, it is sufficient to specify, e.g.:

```
import      import math
```

Having that, you can use a function computing the cosine for any argument (expressed in radians) and to call a pre-defined constant storing the value of π as follows:

```
x = math.cos(math.pi/2)
```

Convince yourself that the following line of code will not work:

```
x = cos(pi/2)
```

We need to provide a prefix pointing at the module’s name. The latter becomes possible if the import is defined as:

```
import *      from math import *
x = cos(pi/2)
```

You can read Python syntax in a number of instances nearly as you read plain English, here in this example: “from math import **all** functions”.

It is also possible to write:

```
import .. as ..      import math as mth
x = mth.cos(mth.pi/2)
```

what we have seen earlier, e.g.:

```
import numpy as np
```

A problem can take place when we try to import two different libraries containing functions of the same name. Analyse the following:

```
>>> import cmath, math
>>> x1 = math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> x2 = cmath.sqrt(-1)
1j
```

In this case we have implicitly specified the `sqrt` function to be taken from the `math` and `cmath` module, respectively. `ValueError` occurred for the former as expected. However, if by mistake you specify two `import` functions as below:

```
>>> from cmath import *
>>> from math import *
>>> x = sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

an error will occur again. Why? Well, just by importing all functions from the `math` module we have overwritten the functions imported from the `cmath` module. Any solution? There are couple as you might suspect.

We can limit a number of functions to be imported from a specific library, e.g.

`from .. import .. as ..`

```
>>> from cmath import sqrt as csqrt
>>> from math import exp, sqrt as sq, log10
>>> x = csqrt(-2)
>>> x
1.4142135623730951j
```

or write a dedicated function that examines the input and returns the correct solution, e.g.

```
def Sqrt(x):
    if(x < 0.0):
        from cmath import sqrt
        return sqrt(x)
    else:
        from math import sqrt
        return sqrt(x)

print(Sqrt(1))
print(Sqrt(-1))
```

displaying

```
1.0
-1j
```

In general, PEP8 guideline does not recommend the use of

```
>>> from module import *
```

By doing that we run the increased risk among all functions to be misused, that are imported from different sources to be misused. As shown above.

Therefore, by marking only those functions which we intentionally want to use, it resembles beaming the individuals up from one to another place of the spacetime as we know it well from the *Star Trek* movie series. Importing `antigravity` in Python is also possible!

Lastly, `Code 2.3` still has not been exhausted in its description yet. Can you tell me why? Try to re-run the code with:

```
x = 10000
```

You will get an `OverflowError`:

```
Traceback (most recent call last):
  File "<>", line 11, in <module>
    f = x*log((exp(x)*tanh(x)),2)
```

Indentation in Python

Python assumes four (4) white spaces as its default indentation. Unlike some other programming languages, this style of writing of the Python code allows its simpler and quicker readability.

Try to memorise the way how we format the main code and the functions. Everything what has been indented is treated as a block of statements and executed accordingly. There is no need to close those blocks with any “end”-like functions (as required in MATLAB or FORTRAN) nor curly braces (as in C/C++).

```
OverflowError: math range error
```

which causes your program to be terminated. No so great, isn't it?

2.1.3. Built-in Exceptions

So far, we have familiarised ourselves with three kinds of errors in Python, namely: `NameError`, `ValueError`, and `OverflowError`. They all belong to a class of **Built-in Exceptions** in the Python module of `exceptions`. This module never needs to be imported explicitly with the use of the `import` function.

Python provides us with a handy way to capture any error that might occur while running a code. Analyse the following:

Code 2.4

```

from math import sqrt

a = -1.0
b = 1.0

try:
    c = sqrt(a/b)
    test = True
except Exception as e:
    # Something went wrong!
    print("Detected error: %s" % e)
    test = False
finally:
    if(test):
        print(c)

```

an equivalent of "if test is True"

When executed the result will be:

```
Detected error: math domain error.
```

Now, by changing:

```
a = -1.0
b = 0.0
```

we get

```
Detected error: float division by zero
```

while for

```
a = -1.0j
b = 0.0
```

the bad news is:

```
Detected error: complex division by zero.
```

As you can see `try-except-as-finally` block defined above allows us to understand what sort of error occurred. If so, the message with a name of an error is displayed for our knowledge. Playing safe, we add a boolean variable, `test`, in order to proceed if all went okay.

The inclusion of the `finally` block segment is optional. Consider the following variant of the same code:

Code 2.4a

```

from math import sqrt

a = 1.0
b = 1.0

try:
    c = sqrt(a/b)
    test = True
except Exception as e:
    # Something went wrong!
    print("Detected error: %s" % e)
    test = False

if(test):
    print(c)

```

It will return a value of **c** in the screen because the **try-except** test has been passed. In general, here we use a smart way of controlling the flow based on actions that were taking place along the lines. Though tempting, be cautious about overusing them. Python offers more elegant control over your code's behaviour and the use of such boolean-based logic of the flow should be limited to specific and local fragments of your code.

Python recognises the following built-in exceptions:

```

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        |
        +-- BufferError
        +-- ArithmeticError
            |
            +-- FloatingPointError
            +-- OverflowError
            +-- ZeroDivisionError
        +-- AssertionError
        +-- AttributeError
        +-- EnvironmentError
            |
            +-- IOError
            +-- OSError
                |
                +-- WindowsError (Windows)
                +-- VMSError (VMS)
        +-- EOFError
        +-- ImportError
        +-- LookupError
            |
            +-- IndexError
            +-- KeyError
        +-- MemoryError
        +-- NameError
            |
            +-- UnboundLocalError
        +-- ReferenceError
        +-- RuntimeError
            |
            +-- NotImplementedError
        +-- SyntaxError
            |
            +-- IndentationError
            +-- TabError
        +-- SystemError
        +-- TypeError
        +-- ValueError
            |
            +-- UnicodeError
                |
                +-- UnicodeDecodeError
                +-- UnicodeEncodeError
                +-- UnicodeTranslateError

```

```

+-- Warning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning

```

I've marked those we have encountered so far in red. Having that list you are able to supplement your Python code with extra lines of a *forced* protection which would vouchsafe its smoothness of running in the future.

Consider the following example:

Code 2.5

while-else

A classical loop for the execution of indented block of code *while* a defined condition is `True`. Otherwise, well, you know what.

It is easy to make *while-else* loop executable endlessly. Therefore, keep an eye on a rolling ball.

```

from math import sin, erf, sqrt, fabs
import random

i = 1
while(i <= 10):
    x = random.randint(-100, 100)
    y = random.randint(-1000, 1000)
    try:
        z = sin(x/y)*erf(x-y)*1./sqrt(fabs(x**y))
        print(i, x, y, z)
        i += 1
    except ValueError:
        print(" ValueError: math domain error")
    except ZeroDivisionError:
        print(" ZeroDivisionError: integer division or modulo by zero")
    except OverflowError:
        print(" OverflowError: long int too large to convert to float")

```

The heart of this `code` is to derive a mathematical expression for **z** given as:

$$z = \sin\left(\frac{x}{y}\right) \operatorname{erf}(x - y) \frac{1}{\sqrt{|x^y|}}$$

where *erf* denotes an error function and the input variables of **x** and **y** are drawn randomly from a uniform distribution of integer numbers in the range [-100; 100] and [-1000; 1000], respectively. Ignore for a second the details related to the Python's Standard Library module of `random` as we will spend some time with it a little bit later. Instead, pay attention to `try-except-except-except` block in action.

The code that follows a `try` keyword aims at derivation of **z** first, next printing **i**, **x**, **y**, and **z** on the screen, and (if that block is executed without errors), increasing **i** by 1. In other cases, which now have been tested for a specific error type separately, the corresponding message on error that occurred will be displayed for your knowledge.

An exemplary outcome we might obtain is:

```

OverflowError: long int too large to convert to float
OverflowError: long int too large to convert to float

```

```

ZeroDivisionError: integer division or modulo by zero
ZeroDivisionError: integer division or modulo by zero
1, 66, 58, 1.4396768283572388e-53
OverflowError: long int too large to convert to float
ZeroDivisionError: integer division or modulo by zero
ZeroDivisionError: integer division or modulo by zero
ZeroDivisionError: integer division or modulo by zero
2, -26, -111, 0.0
3, 55, 4, 0.00013889819399227798
OverflowError: long int too large to convert to float
OverflowError: long int too large to convert to float
ZeroDivisionError: integer division or modulo by zero
ZeroDivisionError: integer division or modulo by zero
OverflowError: long int too large to convert to float
4, -24, -17, -453768871475.1213
OverflowError: long int too large to convert to float
5, -2, 781, 2.359531820868905e-118
6, -4, -183, 0.0
OverflowError: long int too large to convert to float
ZeroDivisionError: integer division or modulo by zero
ZeroDivisionError: integer division or modulo by zero
7, -1, 690, 0.8414709848078965
ZeroDivisionError: integer division or modulo by zero
OverflowError: long int too large to convert to float
OverflowError: long int too large to convert to float
ZeroDivisionError: integer division or modulo by zero
OverflowError: long int too large to convert to float
8, 54, -28, -1.6301163358314728e+24
ZeroDivisionError: integer division or modulo by zero
9, 74, 69, 2.732403385235102e-65
10, -99, 98, 1.48792632248559e-98

```

The bad news is that errors in finding **z** did occur. We designed the code in the way to see what sort of errors took place but skipping information for which **x** and **y** that happened. The good news is that our program terminated with success and we found ten pairs of **x** and **y** for which **z** could be calculated. That was possible due to application of **while** loop.

A quick modification of the same code that suppresses information on errors and returns digestible results could be shortened to:

Code 2.5a

```

from math import sin, erf, sqrt, fabs
import random

i = 1
while(i < 11):
    x = random.randint(-100, 100)
    y = random.randint(-1000, 1000)
    try:
        z = sin(x/y)*erf(x-y)*1./sqrt(abs(x**y))
        print(i, x, y, z)
        i += 1
    except:
        pass
pass

```

where, as in poker, **pass** means “pass”. In that case, we get, e.g.

```

1, 92, 79, 2.2668352909869005e-78
2, 33, 93, -0.0
3, -96, -156, 0.0
4, 98, -38, -9.613555342337808e+36
5, 27, -17, -1334436638310.783
6, 86, 76, 2.5947648025557632e-74
7, 56, -49, -6.15619862771058e+42
8, 10, 266, -0.0

```

9, -71, 103, 3.847992229849534e-96
10, -25, -130, 0.0

Putting all together, by now you should grasp the essence of protection against potential errors. In this book we limit its presence to a required minimum. And as in life—some people prefer sex without condoms, right? ☺

2.1.4. math Module

The `math` module from the Python's Standard Library provides us with a range of both fundamental and useful functions. It is initially thought to assume non-complex-analysis maths. If this is not the case, `cmath` should be imported instead.

Below we display a list of most frequently used functions from the `math` library. Within some examples so far, we have seen how one can get access to any of them.

Merchant ID: 49129
Trans Type: Auth

Amount: 73.86

Tip:

Total:

77.00

Constants

`pi` $\pi = 3.141592653589793$
`e, exp(1)` $e = 2.718281828459045$

Powers and Logarithms

`exp(x)` e^x
`log(x)` the natural logarithm of x (to base e)
`log(x, base)` the natural logarithm of x (to base $base$)
 $\log(4, 12) = \log(4)/\log(12)$
`log1p(x)` the natural logarithm of $1+x$ (to base e)
`log10(x)` the base-10 logarithm;
more accurate than `log(x, 10)`
`pow(x,y)` x^y ; converts x and y to floats, unlike $x**y$
`sqrt(x)` the square root of x

Trigonometric Functions

`sin(x)` the sine of x , in radians
`cos(x)` the cosine of x , in radians
`tan(x)` the tangent of x , in radians
`asin(x)` the arc sine of x , in radians
`acos(x)` the arc cosine of x , in radians
`atan(x)` the arc tangent of x , in radians
`atan2(y, x)` `atan(y/x)`, the result is between $-\pi$ and π
`degrees(x)` converts x from radians to degrees
`radians(x)` converts x from degrees to radians
 $\sin(\text{radians}(30)) = 0.5$
`hypot(x,y)` the Euclidean norm, $=\sqrt{x^2 + y^2}$

Hyperbolic Functions

`sinh(x)` the hyperbolic sine of x
`cosh(x)` the hyperbolic cosine of x
`tanh(x)` the hyperbolic tangent of x
`asinh(x)` the inverse hyperbolic sine of x
`acosh(x)` the inverse hyperbolic cosine of x
`atanh(x)` the inverse hyperbolic tangent of x

Special Functions

`erf(x)` the error function of x
`erfc(x)` the complementary error function of x
`gamma(x)` the Gamma functions of x

<code>lgamma(x)</code>	the natural logarithm of <code> gamma(x) </code>
Other Functions	
<code>fabs(x)</code>	<code> x </code>
<code>factorial(n)</code>	<code>n!</code>
<code>x % y</code>	<code>x mod y</code> ; apply for integers
<code>fmod(x, y)</code>	<code>x mod y</code> ; apply for floats
<code>frexp(x)</code>	returns the mantissa and exponent of <code>x</code> as <code>(m,e)</code> such <code>x=m*2**e</code> <code>frexp(1)=(0.5,1)</code>
<code>ldexp(x)</code>	for above does the opposite <code>ldexp(0.5, 1)=1</code>
<code>modf(x)</code>	returns the fractional and integer part of <code>x</code> <code>modf(pi)=(0.14159265358979312, 3.0)</code>
<code>trunc(x)</code>	the real value of <code>x</code> truncated to integer <code>trunc(pi)=3</code> <code>trunc(3.91)=3</code>

2.1.5. Rounding and Precision

So far we have not mentioned anything about **rounding** of float numbers nor their **precision**. As we know, the RAM memory and a domination of 64-bit (sometimes 128-bit) number representation extends our abilities to use larger and larger numbers. However, the upper boundary still exists. Python may display a number which is not quite right, i.e. it differs substantially from our logical expectation. Therefore, it is advised to understand the how, the what, the why, and the when.

As an example, let's repeat a historical test of the Intel Pentium FPU (Floating Point Unit; coprocessor) reported in 1994 by Dr. Thomas Nicely in his email:

<http://www.trnicely.net/pentbug/bugmail1.html>

where a simple derivation of the following expression:

$$(824633702441.0) * (1/824633702441.0)$$

was supposed to return 1 but returned:

```
0.999999996274709702
```

to the surprise of a new Pentium processor user. At that time that event became huge and embarrassing. Today, twenty one years later, employing Python 2.7.10 run in Mac OS X 10.10 of my MacBook Pro with the 2.6 GHz Intel Core i7 processor, I get:

```
>>> x = (824633702441.0)*(1/824633702441.0)
>>> x
0.9999999999999999
```

which is 1, right? Well,

```
>>> print(x)
1.0
```


et voila! The use of the `print` command facilitates inspecting numbers and their precision. We will understand its full syntax and possibilities within the next sections. For now, let's see what we can obtain:

```
>>> from math import pi, tan
>>> x = tan(2/pi); x
0.7393029504866041
print("%1.1f" % x )
0.7
print("%1.2f" % x )
0.74
print("%1.6f" % x )
0.739303
```

If we specify the printing format as `%1.2f` then `x`, which is a float, will be displayed with 2-digit precision. Moreover, 0.739 has been rounded by the `print` command to 0.74. If this is a desired result, we are fine with that. However, what if we would like to round `x` in a more controlled manner?

It is tempting to begin your adventure with rounding of floats with the use of the `round` command. Hold on and check this out:

```
round      >>> round(0.438)           # returns a float in Python 2.7.10
0                                                    # of <class 'int'>
>>> round(0.4999999)
0
>>> round(0.5)
0                                                    # 1.0 in Python 2.7.10
>>> round(0.501)
1
```

It seems that 0.4999999 is approximately equal to 0.5 but somehow Python's `round` function has a different opinion about that. The same for 0.5 itself.

The `math` module equips us with two dedicated functions to handle rounding in the intended direction. Analyse the following:

```
>>> from math import floor, ceil
>>> from math import e, trunc

floor      >>> floor(0.499999)      # returns an integer in Python 3.5
0
>>> floor(0.5)
0
>>> floor(1.98)
1

ceil       >>> ceil(0.0021)         # returns an integer too
1
>>> ceil(3.49999999999)
4
>>> ceil(9.50000)
10
```

In other words, what goes down must go down, what goes up must go up. Keep these difference in mind.

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

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.

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
```

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

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:

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

instead of

```
r= 0.0010000000000000000000000000000000
```

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 = 0.0020000000000000000041633363423
r+t = 0.0030000000000000000062450045135
```

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

Well, I think that in some instances you ought to be (at least) aware of what is going on and why.

Let's try to take the same variable of $r = 0.001$ and inside of a simple loop of **for** add it 10,000,000 times. The expected value, $E(s)$, is 10000. Really? Have a look:

Code 2.7

Another classical loop across various computer languages. With its help we control the exact number of times the underlying indented block of commands will be executed. Here we make use of the `range` function to specify that `i` will run from 1 to 10000000 or/i.e. the loop will be repeated ten million times.

[illegible]

The code returns:

```
r      =    0.00100000000000000000000020816681712
E(s) =   10000.000000000000000000000000000000000000
s      =   10000.0000015785171854076907038688866
```

From an initial false precision detected at the 20th decimal place in `r` we end up with an error in our sum at the 6th decimal place! This is an excellent example of how much trust you can put in **floats**—not only in Python but in any computer language today.

Is there any cure for that? Of course! This is exactly where Python's `decimal` module earns its place in a spotlight.

Briefly speaking, the module has been created to handle imprecise floating-point accuracy. It is very complex in its structure and settings (see its full doc at <https://docs.python.org/3.5/library/decimal.html>). However, for our needs of precise computations we are going to use only two functions out of its full package, namely, `Decimal` and `getcontext`.

Code 2.8

Here is a modification of the previous code:

getcontext

Decimal

```
from decimal import Decimal as d
from decimal import getcontext as df

r = 0.001          # float!
rfloat = r

df().prec = 3      # set decimal precision for 'r'
r = d(str(r))

df().prec = 30     # set decimal precision for 's'
s = d(0.0)
for i in range(1, 10000001):
    s = s + r

print("r      = %40.30f" % rfloat)
print("E(s) =   10000.000000000000000000000000000000")
print("s      = %40.30f" % s)
print("s'     = %40.30f" % float(s))
```

By now the first two lines should be well understood. We import both functions but shorten their names to **d** and **df**, respectively.

The exact formula for solving this problem is:

$$\text{compR} = \left(1 + \frac{0.0365}{365}\right)^{3650} - 1$$

or

$$\text{compR} = R_t[k] = \prod_{j=0}^{m-1} (1 + R_{t-j}) - 1 = \prod_{j=0}^{m-1} \left(1 + \frac{0.0365}{365}\right) - 1$$

if we hold our deposit for **m** periods (i.e. 3650 days) between dates (*t-m*). The Python code that finds requested compound return utilising both methods is presented below. First, we use floats to solve the puzzles. Next, we re-write everything with the application of the `decimal` module and compare precision of all results.

```

from decimal import getcontext as df
from decimal import Decimal as d

pa = 0.0365      # interest rate p.a.
m = 3650         # number of payments (365 days times 10 years)
Ri = pa/(m/10)   # interest rate per period p.a.

print("Ri = %1.30f" % Ri)

# formula (floats)
compR0 = pow(1.+Ri, m) - 1.0

tmp = 1.0
for i in range(1, m+1):
    tmp = tmp*(1.0+Ri)

# compound return (floats)
compR = tmp - 1.0

df().prec = 5
dRi = d(pa) / (d(m)/d(10.))
print("dRi = %s\n" % repr(dRi))

# formula (Decimals)
df().prec = 30
dcompR0 = (d('1.0')+dRi)**d(str(m)) - d('1.0')

tmp = d('1.0')
for i in range(1, m+1):
    tmp = tmp * (d('1.0')+dRi)

# compound return (Decimals)
dcompR = tmp - d('1.0')

print("(formula) compR = %1.30f" % compR0)
print("(loop) compR = %1.30f" % compR)
print()
print("(formula) dcompR = %1.30f" % dcompR0)
print("(loop) dcompR = %1.30f" % dcompR)

```

Our computations deliver:

We have found that our \$10,000 would grow by a bit more than 44% in 10 years, i.e. up to \$14,404.88. The absolute difference between floats and Decimals would be \$0.00000000058 then. So, would you care about those fractions of cents? Nah... ☺

Taking into account a limited precision of floats we may encounter some problems when calculating standard mathematical function like `log` or `exp` for very small values. Again, you must become aware of that fact but keep your sanity intact. Python's `math` module takes care of tiny details. Therefore, if you are a perfectionist below there are some candies you may love. Let's analyse the `log` case first:

log10()

```
from math import *

a = 1.001e-20    # a very small number

x = log(a, 10)
y = log10(a)
print("x      = %1.30f" % x)
print("y      = %1.30f" % y)
print("|y-x|   =  %1.30f = %1.1e" % (fabs(y-x), fabs(y-x)))
print()
```

$$\begin{aligned} x &= -19.999565922520677219154094927944 \\ y &= -19.999565922520680771867773728445 \\ |y-x| &= 0.000000000000003552713678800501 = 3.6e-15 \end{aligned}$$

A very similar case we may observe for \mathbf{a} to be very small:

Code 2.11

log1p()

expm1()

```
a = 1.0047e-21

x = log(a+1)
y = log1p(a)
print("x = %1.30f = %1.5E" % (x, x))
print("y = %1.30f = %1.5E" % (y, y))
print()

a = 1 - exp(y)
a2 = expm1(y)
print("a = %1.30f = %1.5E" % (a, a))
print("a' = %1.30f = %1.5E" % (a2, a2))
print()
```

where `expm1(y)` is the inverse of `log1p(a)`. Therefore for the latter function we expect (as promised) higher accuracy than for a standard use of `log(a+1)` and `expm1(y)` should derive `a` as initially defined, i.e. `1.0047e-21` in the beginning. We check:

[illegible]

Indeed!

The same idea of improved accuracy for very small \mathbf{x} 's had been standing for a **complementary error function**, `erfc(x)`, used heavily in statistics:

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

where $\text{erf}(x)$ denotes a normal error function. The name of the **error function** has nothing to do with any errors nor error measurements — the same as *The October Revolution* leading to creation of Soviet Russia took place in November and not in October of 1917 according to the Gregorian calendar ☺. Thus,

erf(), erfc()

```
x = -3.0006e-16
a = 1 - erf(x)
b = erfc(x)
print("%1.30f" % a)
print("%1.30f" % b)
```

we find that,

1.0000000000000000444089209850063
1.0000000000000000444089209850063

a dedicated function of `erfc(x)` does not improve the performance somehow. Don't ask me why. I don't know. However, you are welcome to use it anytime you wish.

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
```

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.

The input value of **x** for `Fraction` function needs to be firstly converted to a string-type variable. In the 4th 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?

Let's analyse the case of **the square root of 3**. Hold on as we are increasing the complexity of coding! This is what will make you a better programmer. Arnold Schwarzenegger said: *When it burns, it grows!* Do you get my analogy? ☺

Code 2.13

Find the **approximation** of the irrational number `sqrt(3)` by fractions changing their dominator values using upper limits.

```

from math import sqrt, fabs
from fractions import Fraction as fr

x = sqrt(3)
xf = fr(str(x))

print("x = %1.30f" % x)
print("xf = %s" % xf)

tmp = 0
for i in range(10, int(1e6)+1, 10):
    xf_approx = xf.limit_denominator(i)
    xf_float = float(xf_approx)
    diff = fabs(x-xf_float)

    # Convert Fraction(nom,den) to Floats
    xf_string = str(xf_approx)           # covert Fraction to string
    j = xf_string.find('/')              # find index/location of "/"
    nom = float(xf_string[0:j])          # extract nominator; float
    den = float((xf_string[j+1:]))       # extract denominator; float

    if(den != tmp):
        print("    = %15s = %1.15f (%1.0e)" % (xf_approx,
                                                xf_float, diff))
        tmp = den

```

.limit_denominator()

.find()

Notice that in Python you cannot use `range(10, 1e6, 10)` as this function accepts input parameters defined by integer numbers. That is why in 2.13 we are converting `1e6` with `int(1e6)` function into `1000000`.

First, we define **x** and express it as fraction (Fraction object) of **xf**. The default output for that is:

```

x = 1.732050807568877193176604123437
xf = 4330127018922193/2500000000000000

```

Such fractional representation (as displayed) seems to have a finite precision. The denominator is `2.5e15` therefore accuracy would be `2.5e-15`.

What comes next in the code is a loop. Here we use the function of `limit_denominator(i)` out of `fractions` module. What it does is it puts the upper limit of **i** for the denominator. We employ `range(from, to, step)` function (as discussed previously) to select all numbers from 10 to 1,000,000 with a step of 10 so 10, 20, etc. A new variable of `xf_approx` is still of Fraction-like type but its denominator is as close to **i** as possible.

We convert such *i*-th approximation of **x** to float-type number and compare the absolute difference, `diff`. Since Fraction object is neither not iterable nor callable, we write a manual conversion of both nominator and denominator to floats. We call a standard Python function of `string.find(token)` which by acting upon the string of `xf_approx` returns the location (index) where the first appearance of the `"/"` token is. That allows us to extract and

```
>>> from fractions import \
... Fraction as fr
>>> x = 0.25
>>> y = fr(x) # or y=fr(str(x))
>>> y
Fraction(1,4)
>>> type(y)
<class 'fractions.Fraction'>
>>> s = str(y) # convert to string
>>> s
'1/4'
>>> j = s.find('/'); j
1
```

Just for now keep in mind that you can have an access to any letter or any slice of the string type variable by making a reference via indexing, for example:

```
>>> z = "sex4free"
>>> z[0:3]
'sex'
>>> z[-2:]
'ee'
>>> z[0:3]+y+z[-1:]*2
'sexyee'
```

We will talk more on **operations on strings** in *Volume II*.

separate nominator and denominator, respectively, and turn them into floats.

As you can verify, the results may produce the same output for several the same values of **i**. Therefore, we add one additional condition to check where the approximation has been already displayed for a given denominator or not. Usually, such shortcuts in programming are pretty handy.

The loop generates the final output:

```
=          12/7 = 1.714285714285714 (2e-02)
=          26/15 = 1.733333333333333 (1e-03)
=          45/26 = 1.730769230769231 (1e-03)
=          71/41 = 1.731707317073171 (3e-04)
=          97/56 = 1.732142857142857 (9e-05)
=         168/97 = 1.731958762886598 (9e-05)
=         265/153 = 1.732026143790850 (2e-05)
=         362/209 = 1.732057416267943 (7e-06)
=         627/362 = 1.732044198895028 (7e-06)
=         989/571 = 1.732049036777583 (2e-06)
=        1351/780 = 1.732051282051282 (5e-07)
=        2340/1351 = 1.732050333086603 (5e-07)
=        3691/2131 = 1.732050680431722 (1e-07)
=        5042/2911 = 1.732050841635177 (3e-08)
=        8733/5042 = 1.732050773502578 (3e-08)
=       13775/7953 = 1.732050798440840 (9e-09)
=       18817/10864 = 1.732050810014728 (2e-09)
=       32592/18817 = 1.732050805123027 (2e-09)
=       51409/29681 = 1.732050806913514 (7e-10)
=       70226/40545 = 1.732050807744481 (2e-10)
=      121635/70226 = 1.732050807393273 (2e-10)
=      191861/110771 = 1.732050807521824 (5e-11)
=      262087/151316 = 1.732050807581485 (1e-11)
=      453948/262087 = 1.732050807556270 (1e-11)
=      716035/413403 = 1.732050807565499 (3e-12)
=      978122/564719 = 1.732050807569782 (9e-13)
=     1694157/978122 = 1.732050807567972 (9e-13)
```

where in brackets we display the precision of our approximations for $\sqrt{3}$. This is the first example on that topic. We will come back to the approximations of mathematical functions later on.

2.1.9. Formatting Numbers for Output

Up till now, you should familiarise yourself with the syntax of the **print** function which we have been using intensively. We also have seen that numbers can be displayed making use of the **format** function. Understanding the calling of both functions and their usability will allow you to adjust your output according to your intentions.

print

Let's begin with the **print** function. Its basic use is:

```
>>> i = 7
>>> f = 7.00001
>>> b = (i == f) # boolean-type based on comparison
>>> l = 2**128
>>> print(i)
7
>>> print(i, f, b, l)
7 7.00001 False 340282366920938463463374607431768211456
```

where the last result has been displayed in the form of a tuple-type variable (more on tuples in Section 2.5.1).

Usually we aim at a proper formatting of numbers when using `print`. The history dates back to the origins of `printf` command in C language and its syntax widely adopted by many new languages since then. In certain sense it established a benchmark. In today's Python its final form does not differ a lot from those implementations which you could meet in Matlab, C++, etc. The starting point is displaying a string in its closed form confined by single or double quotes:

```
>>> print("x= ")          # alternatively print('x= ')
x=
```

By default, when the string has been redirected to the output (e.g. screen) the cursor jumps to the next line below. You can use a special character of `\n` as many times in as many lines as you want to skip, for example:

```
>>> print("x= \n3")
x=
3
>>> print("\nx=\n\n4")

x=

4
```

On the other hand, **keeping the cursor in the same line** requires additional parameter:

```
>>> print("pi = 3.141592", end=" "); print("roughly")
pi = 3.141592 roughly
```

what has been differently solved in Python 2.7.10, i.e. by placing a comma character right after `print` or variable:

```
>>> print("pi = 3.141592"),; print("roughly")
pi = 3.141592 roughly
```

where the semi-colon separates two commands in the same line. The same example is much better presented when you run it as a part of your Python code, e.g.

```
print("pi = 3.141593", end=" ")
print("roughly")
e = 2.71828
print("\tand", end=" ")
print(e, end=" ")
print("is more or less equal to exp(1)")

pi = 3.141593 roughly
    and 2.71828 is more or less equal to exp(1)
```

Though it is still possible to use a notation of `print var` in Python 2.7.x, make a habit to use `print(var)` instead (a new standard in Python 3.x).

A special character of `\t` acts as a tab, i.e. moves the cursor position by fixed four (4) spaces.

The **general formatting** for `print` function can be summarised as presented in the following table:

Format	Description with examples (a=9, b=4.1, c=4.3e-5)
<code>%d</code>	integer <pre>>>> print("\$%d" % a) \$9</pre>
<code>%xd</code>	integer right-adjusted in a field of width x <pre>>>> print("%10d\$" % a) 9\$</pre>
<code>%-xd</code>	integer left-adjusted in a field of width x <pre>>>> print("%-10d\$" % a) 9 \$</pre>
<code>%0xd</code>	integer padded with x leading zeros <pre>>>> print("%010d\$" % a) 0000000009\$</pre>
<code>%f</code>	decimal notation with six decimals <pre>>>> print("%f" % a) 9.000000 >>> print("%f" % b) 4.100000</pre>
<code>%xf</code>	decimal notation right-adjusted in a field of width x <pre>>>> print("%10f\$" % c) 0.000043\$</pre>
<code>%-xf</code>	decimal notation left-adjusted in a field of width x <pre>>>> print("%-10f\$" % c) 0.000043 \$</pre>
<code>%.yz</code>	format z with y decimals <pre>>>> print("%.8f\$" % c) 0.00004300\$</pre>
<code>%x.yz</code>	format z with y decimals in a field of width x <pre>>>> print("%12.7f\$" % c) 0.0000430\$ >>> print("%12.7g\$" % b) 4.1\$</pre>
<code>%e, %E</code>	compact scientific notation, with e or E in the exponent <pre>>>> print("%e" % b) 4.100000e+00 >>> print("%E" % c) 4.300000E-05 >>> print("%12.10E" % c) 4.3000000000E-05</pre>
<code>%g, %G</code>	compact decimal or scientific notation, with e or E <pre>>>> d=61.7775e9 >>> d 61777500000.0 >>> print("%g" % d) 6.17775e+10 >>> print("%14G" % d) 6.17775E+10</pre>
<code>%s, %xs</code>	string, string in a field of width x <pre>>>> s="Python for Quants"</pre>

Regardless of a sign, you can place "+" or "-" as a prefix. Analyse the following lines:

```
>>> x = 25.981
>>> y = -x
>>> print("%+10.3f" % x)
+25.981
>>> print("%+10.3f" % y)
-25.981
```

For displaying **boolean** results use formatting for strings, e.g.

```
>>> x = 3; y = 4.1
>>> test = not(y <= x)
>>> print("test: %s" % test)
test: True
```

```
>>> print("%20s" % s)
Python for Quants
>>> print("%s" % s[-6:])
Quants
```

%% the percentage sign

```
>>> print("%d is %.2f%% of 11" % (a, a/11.))
9 is 0.82% of 11
```

many numbers in one calling

```
>>> print("%d %f %G %.2e" % (a,b,c,d))
9 4.100000 4.3E-05 6.18e+10
```

In all examples we use % sign to inform Python's `print` that all defined formats of numbers/strings within precedent string-type expression will correspond to specific variables as provided after % token. Simple? I believe so.

`format`

A similar but slightly different case of formatting numbers for output we achieve making use of the `format` function. Analyse the following exemplary cases:

```
>>> c = 4.3e-5
>>> format(c, "1.6f")
'0.000043' # the returned value is of a string-type

>>> txt = "We find that 8*c = " + format(8*c, "1.6f")
>>> print(txt)
We find that 8*c = 0.000344

>>> format(8*c, ">12.6f")
'      0.000344' # right-adjusted

>>> format(8*c, "<12.6f")
'0.000344      ' # left-adjusted

>>> format(8*c, "^11.6f")
' 0.000344      ' # centered

>>> format(8*c, ">15.2E")
'      3.44E-04'

>>> format(-8*c, "g")
'-0.000344'

>>> t = "interest rate"
>>> r = 0.046
>>> "at " + format(r*100, "1.1f") + "% " + t
'at 4.6% interest rate'
```

where a quite useful option,

With this method the largest number "nicely" formatted is 9999999999.99, displaying:

£99,999,999,999.99

Adding 0.01 makes the output to be a bit incorrect:

£100,000,000,000.0

```
sav = 21409.657239
sav = float(format(sav, ".2f"))
print("You have £%s in your account" % format(sav, ","))
```

You have £21,409.66 in your account

delivers `format(x, ",")` function which adds thousands separator and returns any float in so-called **bank format** for numbers.

2.2. Complex Numbers with `cmath` Module

A set of complex numbers is introduced where no real number satisfying the polynomial equation

$$1 + x^2 = 0$$

is found. We consider z as a complex number having the form of

$$z = a + bi$$

where both a and b are real numbers and i denotes the so-called the **imaginary unit** such $i^2 = -1$. In this notation, a and b are called the real and imaginary parts of z denoted by $\text{Re}\{z\}$ and $\text{Im}\{z\}$, respectively. Two complex numbers are equal if their real and imaginary parts are equal, respectively. If the imaginary part of the complex number z is zero, then we consider z as a real number. If $\text{Re}\{z\}=0$ we talk about the **pure imaginary number**. The **complex conjugate** for any z is defined usually as

$$\bar{z} \equiv z^* = a - bi$$

The Python's module of `cmath` allows you for the fundamental operations on complex numbers. In this Section we will see how to use them efficiently for solving mathematical problems.

2.2.1. Complex Algebra

With no bigger difference, if compared to the algebra of real numbers, the fundamental operations with complex numbers become available in a similar manner. For four basic operations (addition, subtraction, multiplication, division) as given for two numbers of

$$z_1 = a + bi$$

$$z_2 = c + di$$

it is easy to prove that:

$$z_1 + z_2 = (a + bi) + (c + di) = (a + c) + (b + d)i$$

$$z_1 - z_2 = (a + bi) - (c + di) = (a - c) + (b - d)i$$

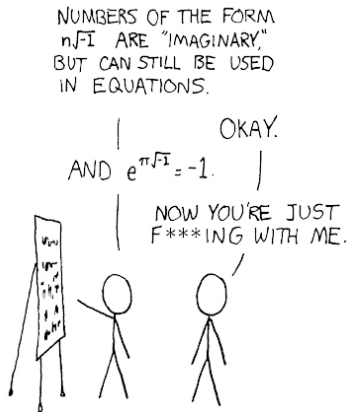
$$z_1 z_2 = (a + bi)(c + di) = (ac - bd) + (ad + bc)i$$

$$z_1/z_2 = \frac{(a + bi)}{(c + di)} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$$

$$\text{for } c \neq 0, d \neq 0$$

In Python we create complex variables in the following way:

```
>>> z = 3+2j; z # j, instead of i, is used
(3+2j)
>>> type(z)
<class 'complex'>
```



```
>>> from cmath import e, pi, sqrt
>>> e**(pi*sqrt(-1))
(-1+1.2246467991473532e-16j)
```

Indeed!

where a pure imaginary number would look like:

```
>>> z0 = 1j; z0 # z0 = j will raise an error!
1j
```

For $z = 5-2.17j$ we can find its $\text{Re}\{z\}$ and $\text{Im}\{z\}$ parts:

```
.real    >>> z.real
5.000
.imag    >>> z.imag
-2.17
```

both returned as floats.

Code 2.14 Solve the following cases and check their correctness in Python.

(a) $(3 + 2i) + (-7 - i) = 3 - 7 + 2i - i = -4 + i$

We derive:

```
>>> (3+2j)+(-7-1j)
(-4+1j)
```

(b) $\frac{3-2i}{-1+i} = \frac{3-2i}{-1+i} \cdot \frac{-1-i}{-1-i} = \frac{-3-3i+2i+2i^2}{1-i^2} = -\frac{5}{2} - \frac{1}{2}i$

```
>>> (3-2j)/(-1+1j)
(-2.5-0.5j)
```

(c)
$$\begin{aligned} \frac{3i^{30} - i^{19}}{-1 + 2i} &= \frac{3(i^2)^{15} - (i^2)^9 i}{-1 + 2i} = \frac{3(-1)^{15} - (-1)^9 i}{-1 + 2i} \\ &= \frac{-3 + i}{-1 + 2i} \cdot \frac{-1 - 2i}{-1 - 2i} = \frac{5 + 5i}{5} = 1 + i \end{aligned}$$

```
>>> (3*1j**30-1j**19)/(-1+2j)
(1+1j)
```

(d) For $z = -\frac{1}{2} + \frac{\sqrt{3}}{2}i$ derive:

$$\begin{aligned} (z^*)^4 &= \left(-\frac{1}{2} - \frac{\sqrt{3}}{2}i\right)^4 = \left[\left(-\frac{1}{2} - \frac{\sqrt{3}}{2}i\right)^2\right]^2 \\ &= \left[\frac{1}{4} + \frac{\sqrt{3}}{2}i + \frac{3}{4}i^2\right]^2 = \left(-\frac{1}{2} + \frac{\sqrt{3}}{2}i\right)^2 \\ &= \frac{1}{4} - \frac{\sqrt{3}}{2}i + \frac{3}{4}i^2 = -\frac{1}{2} - \frac{\sqrt{3}}{2}i \end{aligned}$$

At this point, make sure you group the float number ($\text{Im}\{\mathbf{z}\}$) in brackets, then multiply it by i . Otherwise, you may arrive at a different destination!

```
>>> from math import sqrt
>>> z = -0.5+(sqrt(3)/2)*1j # not: -0.5+sqrt(3)/2j, etc.
>>> z
(-0.5+0.8660254037844386j)

.conjugate() >>> z.conjugate()**4
(-0.5-0.8660254037844384j)
```

2.2.2. Polar Form of z and De Moivre's Theorem

A complex number, $z=x+y*1j$, can be represented graphically in the plane with two axis: real and imaginary. That allows us to express it in **polar form**, namely:

$$z = x + iy = r(\cos \theta + i \sin \theta)$$

where

$$x = r \cos \theta, \quad y = r \sin \theta$$

the **modulus** (absolute value) of \mathbf{z} is

$$\text{mod} z \equiv |z| = r = \sqrt{x^2 + y^2} = |x + iy|$$

and the **argument** of \mathbf{z} is often denoted as:

$$\theta = \text{Arg } z$$

For any two complex numbers one can show that:

$$\begin{aligned} z_1 z_2 &= r_1 r_2 [\cos(\theta_1 + \theta_2) + i \sin(\theta_1 + \theta_2)] \\ z_1 / z_2 &= r_1 / r_2 [\cos(\theta_1 - \theta_2) + i \sin(\theta_1 - \theta_2)] \end{aligned}$$

what in case of multiplication of \mathbf{z} n -times

$$z_1 z_1 \cdots z_1 = [r(\cos \theta + i \sin \theta)]^n$$

leads to so-called **De Moivre's theorem** expressed as:

$$z^n = r^n (\cos n\theta + i \sin n\theta)$$

Now, if we assume the infinite series expansion of:

$$e^x = 1 + x + (x^2/2!) + (x^3/3!) + \dots \quad \text{then for } x = i\theta$$

we arrive at:

$$e^{i\theta} = \cos \theta + i \sin \theta$$

what constitutes a well known **Euler's formula**. With its help, De Moivre's theorem reduces to (modulus skipped)

$$(e^{i\theta})^n = e^{in\theta}.$$

Python's module of `cmath` equips us with special functions that help to represent complex numbers both in polar and rectangular forms. Let's have a look at exemplary implementation.

Code 2.15

Express the following number in polar form and ask Python to display it using Euler's formula, additionally.

$$z = 2 + 2\sqrt{3}i$$

First, we find modulus and next $\text{Arg}\{z\}$. We will also need to import the function of `degrees` from the `math` module. To accomplish the task a nice custom function would be most welcomed, don't you think? Therefore,

$$r = |2 + 2\sqrt{3}i| = \sqrt{4 + 12} = 4$$

$$\text{Arg } z = \sin^{-1}(2\sqrt{3}/4) = \sin^{-1}(\sqrt{3}/2) = 60^\circ = \pi/3 \text{ (rad)}$$

The polar form of z is

$$\begin{aligned} z &= r(\cos \text{Arg } z + i \sin \text{Arg } z) = 4(\cos 60^\circ + i \sin 60^\circ) = \\ &= 4(\cos \pi/3 + i \sin \pi/3) = 4e^{i\pi/3} \end{aligned}$$

Now, let's utilise our current knowledge of Python programming and number formatting for output, and design a lovely function returning a string holding the solution as above.

```
from cmath import polar, exp
from math import sqrt, degrees, pi
```

`polar()`

```
def zpolar(z):
    r, theta_rad = polar(z)

    t = format(z.real, 'g')
    if(z.imag < 0):
        t = t + format(z.imag, 'g') + "j"
    elif(z.imag == 0):
        pass
    else:
        t = t + "+" + format(z.imag, 'g') + "j"

    Pi = u"\u03C0" # unicode for symbol of pi; string

    t2 = t + " = \n\t" + "\t\ttr[cos(Arg z)+isin(Arg z)]" + \
        "\n\t\t\t" + \
        "\n\t\t\t"
    t2 += format(r, "g")
    t2 += "[cos(" + format(degrees(theta_rad), "g") + "deg)"
    t2 += "+isin(" + format(degrees(theta_rad), "g") + "deg)]"
    t2 += "\n\t\t\t"
```

```
t2 += format(r, "g")+"exp(i"+format(theta_rad, "g")  
t2 += ") = \n\t\t\t\t"  
t2 += format(r, "g")  
t2 += "exp(i"+format(theta_rad/pi, "g")+Pi+") ="  
t2 += "\n\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t"  
  
ze = r*exp(theta_rad*1j)  
  
t2 += format(str(ze), "s")  
  
return t2  
  
# main program  
  
z = 2+(2*sqrt(3))*1j      # provide any complex number  
print(zpolar(z))
```

Our program returns the following output:

$$\begin{aligned} 2+3.4641j &= r[\cos(\text{Arg } z)+j\sin(\text{Arg } z)] = \\ &= 4[\cos(60\text{deg})+j\sin(60\text{deg})] = \\ &= 4\exp(j1.0472) = \\ &= 4\exp(j0.33333\pi) = \end{aligned}$$

We obtain a complex number based on its polar form making use of `rect` function, for example:

```
>>> from cmath import *
>>> z = -5.1+sqrt(2)*1j
>>> z
(-5.1+1.4142135623730951j)
>>> r, arg = polar(z)
>>> z2 = rect(r,arg)
>>> z2
(-5.1+1.414213562373096j)
```

Thus, we wrote a simple custom function of `zpolar`. Its main body could be given as an integrated part of the main program but for sake of elegance, it is more handsome to keep its beauty individually preserved. We made use of the `polar` function from the `cmath` module. Based on an input parameter to be a complex number that function returns modulus and argument of \mathbf{z} in a form of a tuple. By assigning two variables (`r` and `theta_rad`) we exactly control what is what. The latter holds $\text{Arg}\{\mathbf{z}\}$ expressed in radians. What follows is a longer series of formatted text where you can analyse a practical application of the `format` function discussed in the previous Section. I realise it is not the most charming way of programming but it delivers results in the way that we asked for. Finally, at the end of `zpolar` we engage a **complex** function of `exp()` to make sure that the use of Euler's formula indeed works and returns the same complex number of \mathbf{z} where we started our journey from.

Please also note the order of both `import` functions. The first one imports all available functions from `cmath`. By executing the second line of the code we overwrite a complex function of `sqrt`, only(!) (you can use complex `sqrt` alternatively too; see a side note).

2.2.3. Complex-valued Functions

The examples provided within precedent subsections are sufficient to give you the right tools to start programming Python solutions for mathematical problems involving complex numbers. Complex Analysis may require the use of series expansions, calculations of sums, limits, etc. We can achieve this with the application of loops or Python's list.

For now, let's note down the remaining functions from the `cmath` library. Their application in **quantitative finance** is limited as we are usually not interested in derivation of a complex hyperbolic tangent in a portfolio optimisation problem. Complex numbers are part of Fourier transform and can be used for **option pricing** (e.g. <http://www3.imperial.ac.uk/pls/portallive/docs/1/40346.PDF>).

For the sake of completion, in `cmath`, we recognise:

Constants (float)

<code>pi</code>	$\pi = 3.141592653589793$
<code>e, exp(1)</code>	$e = 2.718281828459045$

Let $z=a+b*1j$ be a complex number and a, b real numbers, then:

Powers and Logarithms (complex)

<code>exp(z)</code>	e^z
<code>log(z)</code>	the natural logarithm of z (to base e)
<code>log(z,base)</code>	the natural logarithm of z (to base $base$)
<code>log10(z)</code>	the base-10 logarithm; more accurate than <code>log(z,10)</code>
<code>sqrt(z)</code>	the square root of z

Trigonometric Functions (complex)

<code>sin(z)</code>	the sine of z
<code>cos(z)</code>	the cosine of z
<code>tan(z)</code>	the tangent of z
<code>asin(z)</code>	the arc sine of z
<code>acos(z)</code>	the arc cosine of z
<code>atan(z)</code>	the arc tangent of z

Hyperbolic Functions (complex)

<code>sinh(z)</code>	the hyperbolic sine of z
<code>cosh(z)</code>	the hyperbolic cosine of z
<code>tanh(z)</code>	the hyperbolic tangent of z
<code>asinh(z)</code>	the inverse hyperbolic sine of z
<code>acosh(z)</code>	the inverse hyperbolic cosine of z
<code>atanh(z)</code>	the inverse hyperbolic tangent of z

Other Functions

<code>isinf(z)</code>	True if $\text{Re}\{z\}$ and $\text{Im}\{z\}$ is $-\infty$ or $+\infty$
<code>isnan(z)</code>	True if $\text{Re}\{z\}$ and $\text{Im}\{z\}$ is not a number
<code>phase(z)</code>	the phase of z to be in $[-\pi; \pi]$; an equivalent of <code>math.atan2(z.imag, z.real)</code>

An operation of raising a complex number to the power of a real number we can obtain by using a standard operator of `**` or default function of `pow`, e.g.

```
>>> z
>>> z**2
(-85.51728900000002-39.8286j)
>>> pow(complex(-2.1, 9.483), 5.13)
(-112590.56145182674+28581.502958481673j)
```

`complex(a,b)`

Alternatively, you can create a new complex variable using `complex(a,b)` function for any $a+b*1j$ number.

The latter function does not need to be imported from `math`. The same methods can be used to find the result of raising a complex number to the power of a complex number, for example:

$$i^i = e^{i \ln i} = e^{i \ln(\cos \frac{\pi}{2} + i \sin \frac{\pi}{2})} = e^{i \ln e^{\frac{i\pi}{2}}} = e^{-\frac{\pi}{2}}$$

```
>>> from cmath import exp, pi
>>> 1j**1j
(0.20787957635076193+0j)
>>> exp(-pi/2.)
(0.20787957635076193+0j)
```

Is it the only solution? I'll leave it with you.

References

Silverman, R. A., 2013, *Introductory Complex Analysis*, Dover Publications, Inc., New York

Spiegel M. R., Lipschutz S., Schiller J. J., Spellman D., 2009, *Complex Variables, 2nd Ed.*, Schaum's Outlines

Further Reading

cmath—Mathematical functions for complex numbers
<https://docs.python.org/3.5/library/cmath.html>

2.3. Lists and Chain Reactions

Python introduces four fundamental **data structures**: lists, tuples, sets, and dictionaries. When encountered for the very first time, the concept may cause mixed feelings on data aggregation, manipulation, and their use. As a beginner in Python you just need to take a leap of faith that such clever design, in fact, pays huge dividends in practice.

For many reasons and by many people Python's **lists** are considered as one of the language's greatest assets. You can think of lists as of data collectors or containers. The fundamental structure is a sequence, e.g. of numbers, strings, or a mix of numbers and strings. Data are then organised in a given order and each element of such data sequence has its assigned position, thus an *index*. By making a reference by index we gain an immediate access to data or sub-data structures in a given list.

Not only we can build single lists containing data of different types but also use them concurrently to generate new lists or to gather results from (many) numerical calculations in one place. I dare to call this process—the *chain reactions*—a bit of nuclear power of Python at your private disposal. ☺

The Python's lists may seem a bit uncomfortable in the beginning. This are two reasons: (a) the indexing starts from zero and not from one; (b) slicing the list requires an odd use of indexing. The pain vanishes with practice and those two obstacles become your natural instinct.

Similarly as applied in C++ or Java languages, the indexing starting at 0th-position is highly unnatural. We usually do not start the counting process of fruits in a box from zero, right? The only example that could justify counting something from zero is: money in your wallet. First, you have nothing:

```
>>> wallet = [] # an empty list in Python; square brackets
>>> wallet
[]
>>> type(wallet)
<class 'list'>
```

or

```
>>> wallet = [None]; wallet
[None]
```

where None is of <class 'NoneType'>. Next, you earn \$1.50 by selling, say, an apple to your first customer and you jot this transaction down (add it to your Python's list) for the record:

```
>>> wallet = [None, 1.50]
>>> wallet[0]           # there is no physical value assigned
>>> wallet[1]           # index=1 :: 2nd position in the list
1.5
```

If you continue your merchandising experience you may end up at the end of the day with a track record of your income, e.g.: `wallet = [None, 1.50, 4.50, 3.00, 9.00, 1.50, 15.00, 10.50, 6.00]`.

Awesome! So, how many apples have you sold in each transaction and in total? What is your gross income? You can find it very easily. Analyse the following program:

Code 2.16 Trivial mathematics with Python's lists.

```
wallet = [None, 1.50, 4.50, 3.00, 9.00, 1.50, 15.00, 10.50, 6.00]

income = 0;
quantity = 0
apples = []
for dollars in wallet:
    if(isinstance(dollars, float)): # check: is it a float or not?
        q = dollars/1.50           # a quantity per transaction
        quantity += q              # a sum over apples sold
        income += dollars
        apples.append(int(q))      # create a new list!

print("%g apples sold for $1.50 each\ngross income: $%1.2f"
      % (quantity,income))
print("quantity of apples sold per transaction:\n %s" % apples)
```

what generates the following output:

```
34 apples sold for $1.50 each
gross income: $51.00
quantity of apples sold per transaction:
[1, 3, 2, 6, 1, 10, 7, 4]
```

Given a list of **wallet** we employ a **for-in** loop to perform the required calculations. What the loop does is it iterates the list—element by element. In Python we have freedom of naming a variable which refers to an element of the list. Therefore, in our case, the variable of **dollars** refers to the list's value `None` (in its first iteration), next to `1.50`, and so on. You can understand it better by printing **bucks** in your wallet as follows:

```
for bucks in wallet:
    print(bucks, end=" ")

None 1.5 4.5 3.0 9.0 1.5 15.0 10.5 6.0
```

All values of our list are of the float type except the first one. If we wish to perform a simple calculation of gross income or to find an individual number of items sold (and save them in a separate list; a list of apples sold), we need to be sure we deal with the list's elements of the float type only. In [Code 2.16](#) we make use of `isinstance` function again. We have already seen it in action in [Code 2.2](#). If the boolean value of `isinstance(dollars,float)` is `True` then we proceed with statistics.

You can create a new list by hand anytime. However, as in the case of **apples** in our program, we add **only one value** (here: a float object) coming from a new transaction by the `.append(obj)` function that changes the list *in-place*, i.e. it modifies the old list directly, e.g.

You can find a number of transactions (**number of list's elements**) with a help of `len` function:

```
len(apples)
```

or

```
len(wallet)-1
```



```
>>> a = [] # create/initiate an empty list
>>> x = 9
>>> a.append(x); a
[9]
>>> a.append(x*2); a
[9, 18]
```

The function of `.append` belongs to a group of operations referred to in Python as **list methods**. We will discuss them all very soon.

2.3.1. Indexing

The excitement related to Python's lists is enormous if you are a savvy data analyst. With progress of your studies you will embrace their usefulness. They can be utilised as a raw input for **NumPy** functions (Chapter 3) or applied anytime you need to organise your data without applying the third-party Python modules.

However, first, we need to make you fluent in the basics. Therefore, let's have a look at **indexing** — the way that data living in the lists are referred to. We've already spoken about it. The first element in a list has *0th* index,

```
>>> a = [3, 4, 9, 2, -3, 9, -14, 1, 0, 20]

>>> a[0]
3
>>> a[1]
4
>>> a[2]
9
```

and the last one is indexed at $(\text{len}(\text{list})-1)$, i.e.

```
>>> n = len(a) # number of elements in a list
>>> n
10
>>> a[n-1]
20
>>> a[n]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Here, the **IndexError** is raised when your (running) index-variable exceeds the accessible set of list's elements. Again, pay attention that although the list of **a** is 10-element long, just by making a reference `a[10]` you commit a *faux pas*. ☹

It is also handy to denote that the last element of a list can be accessed by:

```
>>> a[-1]
20
```

and, looking from the end towards the beginning of a list:

```
>>> a[-2]
0
>>> a[-3]
1
```

```
>>> a[-n]
3
```

That is exactly what I meant telling you about the *odd use of indexing*. However, if now you memorise all those rules now nothing else will be difficult in Python for you any more. All right, let's move on.

2.3.2. Constructing the Range

A natural consequence of list indexing is a construction of a range of numbers or list's elements. We achieve that with a help of two Python's functions: `range` and `xrange`. Let's consider printing of a list's members first:

`range`
`xrange`

```
a = [3, 4, 9, 2, -3, 9, -14, 1, 0, 20]
for i in range(len(a)): # or xrange(len(a)) in Py 2.7+ only
    print(i,a[i])

(0, 3)
(1, 4)
(2, 9)
(3, 2)
(4, -3)
(5, 9)
(6, -14)
(7, 1)
(8, 0)
(9, 20)
```

This code prints all elements indexed, in fact, from 0 to 9, as we have discussed in the previous section.

In Python 3.4+ `xrange` has been removed and now `range` behaves in the same way as `xrange` used to in Python 2.7.x, i.e:

```
$ python3
Python 3.5.0b4 (v3.5.0b4:c0d641054635,
Jul 25 2015, 16:26:13)
[GCC 4.2.1] on darwin
Type "help", "copyright", "credits" or
"license" for more information.

>>> xrange(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'xrange' is not defined

>>> r = range(10)
>>> type(r)
<class 'range'>
```

Italic style has been used to display the results. Run in Python 2.7.10 to see the list's elements as given or use `list` function in Python 3.5; see below:

```
>>> x = range(1, 11)
>>> x
range(1, 11) # Python 3.5
>>> list(x)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`xrange(len(a))` and `xrange(0, len(a))` have the same effect similarly to `range(len(a))` and `range(0, len(a))`. So, what's the difference? Well,

```
>>> xr = xrange(0,10); xr # only in Python 2.7.10
xrange(10)
>>> type(xr)
<type 'xrange'>

>>> r = range(0, 10); r
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> type(r)
<class 'range'>
```

where the latter returns a list and the former is an internal Python's iterator. **Tip:** make a habit to use `xrange` in Python 2.7.x and `range` in 3.4.x (see the side note). Both functions have been optimised for speed.

How about skipping some elements while printing (or accessing) a list? It is sufficient if you remember that in order to obtain **even** elements, *2nd, 4th, 6th, ...* from the list you use, e.g.

```
>>> range(1, 10, 2) # an equivalent to range(1, 11, 2)
[1, 3, 5, 7, 9]
```

i.e.

```
a = [3, 4, 9, 2, -3, 9, -14, 1, 0, 20]
for i in range(1, len(a), 2):
    print(a[i], end=" ")

4 2 9 1 20
```

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
>>> range(0, 11, 2)
[0, 2, 4, 6, 8, 10]
```

whereas odd elements: *1st, 3rd, 5th*, etc. we get:

```
>>> range(0, 10, 2) # not(!) the same as range(0, 11, 2)
[0, 2, 4, 6, 8]
```

for instance:

Also notice that:

```
>>> x = range(10)
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::2]
[0, 2, 4, 6, 8]
>>> range(0, 10, 2) == x[::2]
True
```

```
a = [3, 4, 9, 2, -3, 9, -14, 1, 0, 20]
for i in range(0, len(a), 2):
    print(a[i], end=" ")

3 9 -3 -14 0
```

where the latter example is an equivalent to writing a code:

```
for i in range(len(a)):
    if(i % 2 == 0):
        print(a[i], end=" ")

3 9 -3 -14 0
```

i.e., we check if the index modulo 2 is an even number then print odd elements of the list. But, this is much more confusing.

The **general rules** for the use of `range` and `xrange` functions are:

<code>range(n)</code>	0, 1, ..., n-1
<code>range(start, stop)</code>	start, ..., stop-1
<code>range(start, stop, step)</code>	start, start+step, start+2*step, ..., stop-1

2.3.3. Reversed Order

You can reverse the order with a `reversed` function:

```
a = [3, 4, 9, 2, -3, 9, -14, 1, 0, 20]
for i in reversed(range(10)):
    print(a[i], end=" ")

20 0 1 -14 9 -3 2 9 4 3
```

but keep in mind the most important difference in the application of `reversed` and `.reverse()` functions, namely:

```
>>> reversed(range(10))
<listreverseiterator object at 0x1005e5850>

>>> x = range(10); x # works in Python 2.7.10 and 3.5
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x.reverse() # only in Python 2.7.10
>>> x
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

The function of `.reverse()` changes the list *in-place* as you can see, however it only works in Python 2.7.10. To avoid the modification

of `x` and still to be able to generate a new list with reversed order of its elements, use:

```
>>> x = range(10)
>>> x[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Saying that, every second element as counted from the end of the list (including, unfortunately, the last one) is:

```
>>> x = range(1, 11); x
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> x[::-2]
[10, 8, 6, 4, 2]
```

and

```
>>> x[::-3]
[10, 7, 4, 1]
```

In fact, we can get *exactly* every second element as counted from the end by typing rather bizarre constructions of:

```
>>> x = range(1, 11); x
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> x[7::-2]          # remember: x[7] equals 8, not 7!
[8, 6, 4, 2]

>>> x[::-2][1::]      # an equivalent to x[::-2][1:]
[8, 6, 4, 2]
```

By now your mind should be getting the rhythm of how does indexing works. Study it over and over again and one day, maybe it will be tomorrow, you will be fluent in Python indexing.

The devil is never as bad as they paint it. For example, the very last command makes a lot of sense: first, we generate a list by `x[::-2]` returning `[10, 8, 6, 4, 2]` and next we directly apply indexing (more precisely: slicing) starting at *1st* index position (not *0th*).

A reversed order has another cool application: a **countdown**. We can use `range` function to display a number of "seconds" to the closure of a trading session on Wall Street:

```
for seconds in range(-10, 1):
    if(seconds < 0): print(-seconds, end=" ")
    else: print("Session closed!")

10 9 8 7 6 5 4 3 2 1 Session closed!
```

In this case, the same rules apply: 1 is not a member of the list generated by `range(-10, 1)` with 0 being the last one.

2.3.4. Have a Slice of List

Having said that, grasping the concept of **slicing** the list should take you a fraction of a second. Let's summarise what we already know on that subject just within a few lines of code that say absolute everything:

```
>>> x = range(11,21)
>>> x
[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

>>> x[0:0]          # 1st element without the 1st element
[]

>>> x[0:1]          # 1st element; equivalent to x[0]
[11]

>>> x[0:2]
[11, 12]

>>> x[5:7]
[16, 17, 18]
```

From 6th list's element till the end of the list:

```
>>> x[5:]
>>> x[5::]
[16, 17, 18, 19, 20]
```

Over all elements of the list—so-called *show off*—due to its complete impracticality but possibility to be executed in Python without any errors:

```
>>> x
>>> x[:]
>>> x[::]
[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

With some steps, from the beginning and from the end:

```
>>> x[5::2]          # x[5:2] will not work at all; an empty list
[16, 18, 20]
>>> x[5::3]
[16, 19]
>>> x[5::4]
[16, 20]
>>> x[4::5]
[15, 20]

>>> x[::-1]
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11]
>>> x[::-2]
[20, 18, 16, 14, 12]
>>> x[5::-2]
[16, 14, 12]
>>> x[5::-1]          # everything from x[i] down to x[0]
[16, 15, 14, 13, 12, 11]
>>> x[5::-5]          # works because of 5th and 0th index
[16, 11]
```

Playing with slicing can bring a lot of joy and confusion, all in one. Try to figure out what is going on in the following mind puzzle. Run it in Python 2.7.10:

```
x = range(11, 21)
```

```

print("x\t= %s" % x)
for i in range(1, 6):
    print("x[%d:%d] = %s" % (5, -i, x[5:-i]))

x      = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
x[5:-1] = [16, 17, 18, 19]
x[5:-2] = [16, 17, 18]
x[5:-3] = [16, 17]
x[5:-4] = [16]
x[5:-5] = []

```

Got it? It's simple. Start always at *6th* position in the list but finish at *i*-th position before the last list's element. Compare `x[5:-2]` and `x[5::-2]`. Close in notation but completely different in result!

2.3.5. Nesting and Messing with List's Elements

Do you still remember a definition of the Python's list? It is a good quality "container". You can put other lists in it, e.g. of your favourite stock tickers, current prices, or other data in a form of Python's strings, tuples, sets, or even dictionaries. Therefore, a list may contain a mixture of numbers and strings:

```

msg = ["Risk", "is", 4, "brave", "people!"]    # a message
for word in msg:
    if(word == 4): print("for", end=" ")
    else: print(word, end=" ")

```

Risk is for brave people!

Here we print a sentence built from string-type objects in list **msg** and if a number 4 is found, we substitute it with a word "for".

A simple manipulation may go beyond a single list! What follows is a more complicated version of the abovementioned code. Let's analyse step by step this very first chain reaction:

Code 2.17

```

num = [[4, "for"], [2, "to"]]          # numbers in slang
sc = ["!", "#", "@", "^", ".", ";", "+"] # special characters
msg = ["Come", 2, "me, ", "baby", "."]  # the message

for elem in msg:
    if(isinstance(elem, int)):
        # try to find 'elem' in 'num'
        is_in_num = [(elem in num[i]) for i in range(len(num))]
        if(sum(is_in_num) > 0):
            ind = is_in_num.index(True)
            print(num[ind][1], end=" ")
    else:
        if(elem in sc):
            print(elem*3, end=" ")
        else:
            print(elem, end=" ")

```

Come to me, baby ...

in
Python **also** allows you to check whether a given element (object) is a member of a list or not. It is so-called a **membership** check that returns a boolean value as a result, e.g.:

```

>>> a=[-4,"nice",7,"profit"]
>>> not "nice" in x
False
>>> "nice" and "profit" in x
True
>>> "nice" and "ass" in x
False
>>> ("lucky" and 7) not in x
True

```

The main loop **for..in** iterates over all elements in the list of **msg**. First we check with already a well-known to us function of **isinstance** whether a given element is of an integer type or not? If so, we try to locate its value (if present) in the list of **num** which stores some numbers and their slang-spoken equivalents. The list

of **num** is our first example of **nesting** objects (here: two sub-lists) in a list, such that indexing:

```
>>> num[1]
[2, "to"]
```

can be used anytime. In this point you can convince yourself that a membership test of the following form does not work:

```
>>> elem = 2
>>> elem in num
False
```

however,

```
>>> [2, "to"] in num
True
```

That is why we need to search for the number of 2 in **num** list in a more sophisticated manner. The construction used as the output for **is_in_num** variable is worth considering separately:

```
>>> [(elem in num[i]) for i in range(len(num))] # is_in_num
[False, True]
```

namely:

```
>>> (elem in num[0])
False
>>> elem in num[1]
True
```

What we do is in fact a creation of an inner list with outcomes dependent on variable **i** changing its value from 0 to **len(num)**, e.g.

This notation is often referred to as a **list comprehension**. Make an effort to experiment with it a lot to make yourself comfortable with this very *Pythonic* style. You will see it many, many times from now on. Everywhere.

```
>>> x = [-i for i in range(-5, -1)]
>>> x
[5, 4, 3, 2]
>>>
>>> y = [x[i]**2 for i in reversed(range(len(x)))]
>>> y
[4, 9, 16, 25]
```

Soon you will see how frequently this handy operation is for various mathematical calculations and data analysis. I love it for two more reasons: you talk to Python in English and you may complicate the latter list even more profoundly:

```
>>> x
[5, 4, 3, 2]
>>> y2 = [x[i]**2 for i in reversed(range(len(x))) if i % 2 != 0]
>>> y2
[4, 16]
```

or

```
>>> x
[5, 4, 3, 2]
>>> y3 = [x[i]**2 for i in reversed(range(len(x))) if x[i]
          % 2 != 0]; y3
[9, 25]
```

Beautiful, isn't it?

Coming back to the analysis of [Code 2.17](#), now you understand what a variable of `is_in_num` holds: a list of boolean values as a result of our check for `elem`'s membership. You can sum all list's elements since `False` equals 0 and `True` equals 1. If the sum is non-zero we have a certainty that the sought number of 2 is in! Its position (index) in `is_in_num` we find with a help of the Python list's function of `.index(elem)`.

Have a look at the following exemplary code:

```
.index()
>>> x = ["She gave me", 4, "kisses", 4, "free"]
>>> x.index("free")
4
>>> x.index("kisses")
2
```

however:

```
>>> x.index(4)
1
```

despite the fact that number 4 is present twice in the list: `.index` function returns an index corresponding to its first occurrence. Of course, you can find the second position of 4 by typing:

```
>>> -x[::-1].index(4) + (len(x)-1)
3
```

but it's too complex and still fails if 4 occurs more than two times. Solution? Based on what we have discussed so far we get:

```
>>> [i for i in range(len(x)) if x[i] == 4]
[1, 3]
```

Smoothly and painlessly.

Therefore, in [Code 2.17](#), a function `is_in_num.index(True)` will work solely for the first entry of 2 in `num`. The rest of [2.17](#) follows a requested logic of building a sentence in English. Note that if a special character of "." is in `sc` list then we print it three times applying a simple string multiplication (see more in [Section 2.8](#)).

2.3.6. Maths and statistics with Lists

Some of the abovementioned methods used in creation of Python's lists can be accomplished with the `enumerate` object as provided by `enumerate(list)` function. Analyse the following program being a modification of [Code 2.10](#).

Code 2.18

A bank pays on average 4% p.a. for keeping your money in its savings account. However, the interest rate fluctuates month-over-month. Write a code in Python that (i) simulates those monthly fluctuations; (ii) computes a compound return and your capital growth if you hold \$1,000 for a half of a year and compounding takes place monthly.

It is a great exercise where Python's lists can be utilised! First, let's create in a fancy way a temporary list storing 4% interest rate:

```
r = [i*0.01 for i in [4]*6]

[0.04, 0.04, 0.04, 0.04, 0.04, 0.04]
```

Next, it would be nice to get a list of numbers with different signs. Initially assuming that a negative number represents a decrease of the interest rate month-over-month and a positive number denotes its increase, we start from a simple list:

```
tmp = []
for i in range(1, 6+1):
    if(i % 2 == 0): tmp.append(-1)
    else: tmp.append(1)

[1, -1, 1, -1, 1, -1]
```

which is a short and elegant way to generate 1, -1, 1, ... sequence in Python. But we need pseudo-random fluctuations preserving the sign. With a help of modulo operation, we do it, e.g.:

```
fluct = []
for i in range(1, 6+1):
    if(i % 2 == 0): fluct.append(-1*(i % (i+2*i/10.)/1600.))
    else: fluct.append((i % (i+103.5*i/20.)/1600.))

[0.000625, -0.00125, 0.001875, -0.0025, 0.003125, -0.00375]
```

Modulo operation `%` is often used as a function to deliver **random numbers**. As we will see in Section 2.4 on Randomness Built-In, it still has some applications in modern era of programming (e.g. in Java).

Since this sequence is still periodic we wish to rearrange the order of those elements. We may employ a function of `shuffle(list)` from the `random` module in the following way:

```
from random import shuffle, seed
seed(2016)
shuffle(fluct)

[-0.00125, 0.000625, 0.003125, 0.001875, -0.0025, -0.00375]
```

where a quoted shuffled list is obtainable for `seed(2016)` function when executed. Therefore, the resultant 4% numbers destabilised month-over-month we derive as:

```
zip

R = [sum(e) for e in zip(r, fluct)]

[0.03875, 0.040625, 0.043125, 0.041875, 0.0375, 0.03625]
```

Let me take this opportunity to tell you a few words on the use of a new function of `zip`. Its name suggests that it zips parallel elements (an index-wise zipping) coming from different lists. Have a look at some examples below:

```
>>> x = range(1, 6); y = [X**2 for X in x]; x
[1, 2, 3, 4, 5]
>>> y
[1, 4, 9, 16, 25]
>>> z = [e for e in zip(x, y)]
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> z[2]
(3, 9)
>>> type(z[2])
<class 'tuple'>
```

and

list
Converts a set into a list element.

```
>>> z = [list(e) for e in zip(x, y)]
[[1, 1], [2, 4], [3, 9], [4, 16], [5, 25]]
>>> type(z[2])
<class 'list'>
```

In fact, we can zip more than two lists. Consider the following:

```
>>> w = [i**3 for i in range(1, 7)]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 4, 9, 16, 25]
>>> w
[1, 8, 27, 64, 125, 216]      # one element more than in x and y
>>>
>>> z = [list(e) for e in zip(x, y, w)]
>>> z
[[1, 1, 1], [2, 4, 8], [3, 9, 27], [4, 16, 64], [5, 25, 125]]
```

but if

```
>>> w = [i**3 for i in range(1, 4)]; w
[1, 8, 27]
>>>
>>> z = [list(e) for e in zip(x, y, w)]; z
[[1, 1, 1], [2, 4, 8], [3, 9, 27]]
```

the process of zipping takes the length of all three lists **x**, **y**, and **w** into account and trims the output accordingly.

Therefore, the line of:

```
R = [sum(e) for e in zip(r, fluct)]

[0.03875, 0.040625, 0.03625, 0.0375, 0.041875, 0.043125]
```

in our [Code 2.18](#) returns zipped elements of **e** being the tuples storing [(0.04, -0.00125), (0.04, 0.000625), ... respectively, and by acting upon each of them with the function of **sum**, we add both tuple's elements together (more on **sum**, next).

The remaining part of the [2.18](#) is trivial:

$$1 + \text{comp}R = \prod_{i=0}^{6-1} \left(1 + \frac{r_i}{12}\right)$$

The entire Python program being the solution to challenge [2.18](#) could be compiled as follows:

```
# a complete Code 2.18
from random import shuffle, seed

r = [i*0.01 for i in [4]*6]
fluct = []
for i in range(1, 6+1):
    if(i % 2 == 0): fluct.append(-1*(i % (i+2*i/10.)/1600.))
    else: fluct.append((i % (i+103.5*i/20.)/1600.))
```

enumerate

```

seed(2016)
shuffle(fluct)

R = [sum(e) for e in zip(r, fluct)]
tmp = [(1+r/12.) for r in R]

compR = 1
for r in enumerate(tmp):
    compR *= r[1]
compR -= 1

sav0 = float(format((1000.), ".2f"))
sav = float(format((1000.*(1+compR)), ".2f"))

print("compR = %.5f%%" % (100.*compR))
print("Capital growth from $%s to $%s\"
      % (format(sav0, ","), format(sav, ",")))

```

returning the final output:

```

compR = 2.00084%
Capital growth from $1,000.00 to $1,020.01

```

The use and functionality of a new `enumerate` function you can grasp by the careful analysis of the following code:

```

r = [-7, -5, -1, 9]
for k in enumerate(r):
    print(k)
    print(k[0])
    print(k[1])
    print()

(0, -7)
0
-7

(1, -5)
1
-5

(2, -1)
2
-1

(3, 9)
3
9

```

The running variable of `k` is a tuple with its first element to be the index of 0, next 1, 2, etc. If there is a need, both running index of *i* and list's *i*-th element, they can be processed concurrently. For a more complete picture, consider the following modification:

```

r = [[-7, 7], [-5, 5], [-1, 1], [9, -9]]
for k in enumerate(r):
    print(k)
    print(k[0])
    print(k[1])
    print(k[1][0])
    print(k[1][1])
    print()

(0, [-7, 7])
0
[-7, 7]
-7

```

```

7
(1, [-5, 5])
1
[-5, 5]
-5
5

(2, [-1, 1])
2
[-1, 1]
-1
1

(3, [9, -9])
3
[9, -9]
9
-9

```

No doubt, it is now much easier to understand the logic standing behind the applicability of the `enumerate` function.

Ready for more **nesting**, **indexing**, and **slicing**? I bet you are! So, here we go again:

Code 2.19

For the following list of `s` containing some information about two stocks given by tickers, display two most recent prices of their shares at the closure of the trading session and make sure that the latest price is being displayed as the first one.

```

s = [{"AAPL", (409.34, 410.23, 410.98, 399.45)}, \
     ["IBM", (125.53, 124.95, 125.01, 125.99)]]

for stock in s:
    print("%s" % stock[0])
    prices = stock[-1][-2:] # more nesting, indexing, slicing!
    for p in reversed(prices):
        print("  $%.2f" % p)

AAPL
  $399.45
  $410.98
IBM
  $125.99
  $125.01

```

In this code I used indexing and slicing of `stock` (a list variable) just to show you how effectively we can refer to any **nested** element in the list using Python. In this case, a list of `s` is composed of two inner lists, each storing a string (a ticker) and tuple (close prices of a stock for last four days).

Therefore the price of IBM four days ago pulled out from our "database" would be:

```

>>> s[1][1][0] # or s[-1][1][0] since IBM is the last
125.53         # element of the list 's'

```

and the average price of AAPL over past three days:

```

sum(list/tuple)      >>> sum(s[0][1][-3:])/3
406.88666666666667

```

Please mark in your notes that the `sum()` function works well for lists and tuples though, as discussed in Section 2.1, to ensure there would be no issues with precision of floats, it is advised applying a function of `fsum()` from the `math` module when working with Python 2.7.10 interpreter. For the latter, the benefit is two-fold, namely:

```
fsum(list/tuple)      >>> from math import fsum
>>> fsum([1, 2, 3])/3  # better when used in Python 2.7.10
2.0
```

returns float-type number even if a list or tuple contains integers and we divide the sum by integer.

statistics

Python 3.5 comes with a handy module of `statistics` that equips us with an alternative and quick way to compute of basic mathematical statistics functions, i.e.: the **mean**, sample and population **variance**, **standard deviation**, **median**, and **mode**. Those methods work fluently with Python's lists. Let's consider a couple of the most useful examples.

For any plain **list** of numbers:

```
>>> import statistics as st
>>> x = range(1, 5)          # a list or Python 3.5's iterator
1, 2, 3, 4
>>> sum(x)
10

>>> st.mean(x)
2.5
>>> st.variance(x)
1.6666666666666667
>>> st.stdev(x)
1.2909944487358056

.mean()
.variance()
.stdev()
```

the last two results have been computed with one degree of freedom, i.e. for the variance defined as a **sample variance**:

$$\text{var}(x) = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2$$

As we will see in Chapter 3, the same is accessible in `numpy` by:

```
>>> import numpy as np
>>> np.var(x, ddof=1)
1.6666666666666667
>>> np.std(x, ddof=1)
1.2909944487358056
```

```
.pvariance(), .pstdev()
.median(), .mode()
```

In order to account for the **population** measure of spread (zero d.o.f.), use `st.pvariance` and `st.pstdev` instead. For measures of central location use `st.median(x)` or `st.mode(x)`. Visit <https://docs.python.org/3.5/library/statistics.html> for more detail.

2.3.7. More Chain Reactions

Python's **list comprehensions** are so powerful for the needs of numerical computations that they require more examples in order to fully appreciate what they have to offer. Since examples tell the whole story, let's see how complicated Pythonic structures can be created from an innocent, simple list.

A classical gateway would be:

```
>>> [s for s in [3.14]]
[3.14]
```

Generation of n elements of the same kind we may achieve by:

```
>>> n = 5
>>> [s for s in [3.14]*n]
[3.14, 3.14, 3.14, 3.14, 3.14]
```

where doubling the result:

```
>>> n = 5
>>> [s*2 for s in [3.14]*n]
[6.28, 6.28, 6.28, 6.28, 6.28]
```

However if we aim at taking 3.14 first, and generating iteratively the i -th element as $3.14*i$ for $i = 1, 2, \dots$ then:

```
>>> n = 5
>>> [s*i for s in [3.14] for i in range(1, n+1)]
[3.14, 6.28, 9.42, 12.56, 15.70]
```

or

```
>>> n = 5
>>> [3.14*i for i in range(1, n+1)]
[3.14, 6.28, 9.42, 12.56, 15.70]
```

in short. As we have seen before, if the odd (or even) values of i are sought after, then apply additional condition, e.g.:

```
>>> n = 5
>>> [3.15*i for i in range(1, n+1) if (i % 2 != 0)]
[3.14, 9.42, 15.70]
```

You can generate a streak of sublists inside a list taking some trial values from the specific list:

```
>>> x = [[i, 2**i] for i in [4, 5, 6, 7, 8]]
>>> x
[[4, 16], [5, 32], [6, 64], [7, 128], [8, 256]]
>>> x[2]
[6, 64]
```

A conditional modification may filter the results, e.g.:

```
>>> from math import log
>>> [[i, 2**i] for i in [4, 5, 6, 7, 8] if
      (log(i, 2)-log(i, 2)//1 == 0)]

[[4, 16], [8, 256]]
```

where we forced to display those results for which the fractional part of expression $\log(i, 2)$ is equal zero (opposite of the `math`'s function of `trunc`).

A sequence of sublists we may get by:

```
>>> x = [list(range(1, n)) for n in range(2,7)]
>>> x
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4], [1, 2, 3, 4, 5]]
```

where **flattening** it takes a fancy construction employing the `for` function, e.g.:

```
>>> f = [z for y in x for z in y]
>>> f
[1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5]
```

or

```
>>> f = sum(x, [])
>>> f
[1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5]
```

Having that, the **unique values** of `f` list you may find by:

```
>>> unique = list(set(f))
>>> unique
[1, 2, 3, 4, 5]
```

If for some reasons the order of the original (by first occurrence) must be preserved then apply:

```
>>> x = [1, 1, 5, 2, 2, 4, 3, 3, 4, 5, 1, 2]
>>> unique2 = sorted(list(set(x)), key=lambda z: x.index(z))
>>> unique2
[1, 5, 2, 4, 3]
```

Our chain reactions may be more complicated than that. Have a look at the following case:

```
>>> x = [list([n, list(range(1, n+1))]) for n in range(2, 5)]
>>> x
[[2, [1, 2]], [3, [1, 2, 3]], [4, [1, 2, 3, 4]]]
```

Here, each `x`'s element is a sublist composed of a number and (subsub)list. Flattening of that list is more complicated than you think. I will leave it with you as an exercise.

What is appealing is the use of `_` (underscore) token for chain reactions. Analyse the following:

```
>>> x = [3, 4, 5, 6, 7]
>>> len(x)
5
>>> y = [0.01 for _ in x]
[0.01, 0.01, 0.01, 0.01, 0.01]
```

By doing so, we generate a new list, `y`, that contains 0.01 numbers in quantity corresponding to the number of elements of list `x`. A very useful trick. Memorise it!

In this case, you can check that "adding" **x** and **y** lists does not work as an algebraic (element-wise) addition:

```
>>> z = x + y
>>> z
[3, 4, 5, 6, 7, 0.01, 0.01, 0.01, 0.01, 0.01]
```

but a plain list concatenation. Disappointed a bit? No worries! The element-wise addition employing a list comprehension process you may achieve by:

```
>>> x = [3, 4, 5, 6, 7]
>>> y = [0.01 for _ in x]
[0.01, 0.01, 0.01, 0.01, 0.01]

>>> z = [x+y for x, y in zip(x, y)]
[3.01, 4.01, 5.01, 6.01, 7.01]
```

where we used the **zip** function discussed earlier.

Significantly high level of complexity involving list comprehension is illustrated by the following example. Imagine you have three stocks (AAPL, IBM, JNJ) and you would like to generate a list of pseudo-prices for *n* consecutive days for all three assets (without using **pandas** library). Here is the first step. You may wish to define a custom function **randomprice** that returns a random number, say, between \$80 and \$120. Given a list of stocks, **tickers**, you may combine all the information into one list by writing:

Code 2.20

```
def randomprice():
    import random
    return random.uniform(80, 121) # float

tickers = ["AAPL", "IBM", "JNJ"]
n = len(tickers) # 3

lst = [[i, stock, price] for i in range(1, n+1) \
        for stock in tickers \
        for price in [randomprice()]]

print(lst)

[[1, 'AAPL', 109.50214139219915],
 [1, 'IBM', 97.65676759901652],
 [1, 'JNJ', 109.09379544608493],
 [2, 'AAPL', 84.35342747568366],
 [2, 'IBM', 115.60531620970514],
 [2, 'JNJ', 80.21930868253156],
 [3, 'AAPL', 89.71910886539594],
 [3, 'IBM', 106.60642865139782],
 [3, 'JNJ', 106.29613079231312]]
```

We used an inner product property of the Python lists. Please note that our first variable, **i**, can take only three values: 1, 2, and 3. **stock** variable goes over all elements in the **tickers** list. As it comes to **price** — it looks into 1-element sublist where we put a call to the external function of **randomprice** that returns a single number (a random float). For the latter, every time the call is made, a new random number is returned. Therefore, by this example, you can see how much flexibility Python has to offer—in one line of code.

You may get a completely different outcome if you generate **lst** as:

```
lst = [[i, stock, price] for i in range(1, n+1) \
        for stock in tickers \
        for price in [randomprice()*n]]
```

what would cause a triple loop over all possible elements.

As a supplement to the abovementioned code, it is a slightly modified version (for a single stock only) that might look like this:

Code 2.21

`datetime`

is a handy module to convert or combine time and dates into more useful forms. We will cover it in greater detail within Appendix. For now, a few examples:

```
>>> d = datetime.now()
>>> d
datetime.datetime(2015, 7, 7,
                  12, 42, 5, 384105)
>>> d.year
2015
>>> d + timedelta(seconds=-11)
datetime.datetime(2015, 7, 7,
                  12, 41, 54, 384105)
```

```
import random
from datetime import datetime, timedelta

def randomprice():
    return random.uniform(115, 121) # float

def randomvolume():
    return random.randrange(100000, 1000000) # integer

lst2 = [(datetime.now()+timedelta(days=i)).strftime("%d-%m-%Y"),
        stock, price, volume] \
        for i in range(10) \
        for stock in ['AAPL'] \
        for price in [randomprice()] \
        for volume in [randomvolume()]]

print(lst2)

[['08-05-2015', 'AAPL', 119.36081566260756, 771321],
 ['09-05-2015', 'AAPL', 118.89791980293758, 142487],
 ['10-05-2015', 'AAPL', 115.78468917663126, 771096],
 ['11-05-2015', 'AAPL', 119.37994620603668, 936208],
 ['12-05-2015', 'AAPL', 118.92154718243538, 540723],
 ['13-05-2015', 'AAPL', 119.59934410132043, 242330],
 ['14-05-2015', 'AAPL', 119.29542558713318, 725599],
 ['15-05-2015', 'AAPL', 117.59180568515953, 465050],
 ['16-05-2015', 'AAPL', 115.18821876802848, 771109],
 ['17-05-2015', 'AAPL', 118.07603557924027, 408705]]
```

Above, we added more functionality from the Standard Library in the form of a (formatted) string storing a date. Without additional documentation on the `datetime` module you may decipher that we are using a running index **i** from 0 to 9 as a forward-looking time delay: the first entry is as for "now", i.e. for a current date (e.g., "08-05-2015") and the following nine dates are shifted by **i** i.e. by 1, 2, ..., 9 days. The function `.strftime("%d-%m-%Y")` helps us to convert the `datetime` object into a readable date (a string variable).

Having that, the quickest way to extract a list of prices or volumes is via an alternative application of the `zip` function—the **argument unpacking**:

`zip(*lst)`
Argument unpacking for lists

```
date, _, prices, vol = zip(*lst2) # use _ to skip a sublist
print(list(prices[-3:]))

[117.59180568515953, 115.18821876802848, 118.07603557924027]
```

where we displayed last three stock prices from the unpacked list of **prices**. In addition, if you look for the day when the volume was largest, try to add:

```
i = vol.index(max(vol)) # finding index; 3
print("Largest volume on day: %s" % date[i])

Largest volume on day: 11-05-2015
```

In short—advanced chain reactions in action! ☺

2.3.8. Lists and Symbolical Computations with `sympy` Module

Within the past 20 years, products like Wolfram's Mathematica or MathCAD established a solid position when it came to symbolic computations. It was my pleasure to share the very first moments of MathCAD Plus 6.0 (1995) software when I came across some heavy statistical problems that only a computer could help me in simplifying complex mathematical formulae into digestable form, easy to evaluate, plot, and seek for results. The major problem was (and still is) around hefty licence fees you need to pay in order to enjoy your work without limitations.

Python's response to symbolic computations delivering pinpoint solutions (free of charges) is its `sympy` module. It grew to the position of a serious threat to Mathematica and has been winning the hearts and minds of many mathematicians, scientists, and researchers over the past few years.

Symbolic computations helps financial **quants** too. Continuous efforts in searching for stable solutions in model validation, exotic derivative pricing and backtesting, risk hedging and applications development is often based on results derived by machines.

You can find the `sympy` module's full documentation at <http://docs.sympy.org> website. It is too extensive to describe here its complete spectrum of possibilities here. The typical applications cover **calculus**, **algebra**, and dedicated **solvers**. I do strongly encourage you to explore it further if you're seeking for analytical solutions related to your current work, projects, or problems.

From our current point of view, let's see how one can easily combine Python's lists with `sympy`'s selected goodies.

Code 2.22 Derivation of n -th derivative of $\log(1+x)$ function: (a) based on n -th term's formula; (b) with a help of `sympy` module.

It is easy to find iteratively that for:

$$f(x) = \ln(1 + x)$$

its first four derivatives are:

$$\begin{aligned}
 f'(x) &= \frac{1}{1+x} \\
 f''(x) &= -\frac{1}{(1+x)^2} \\
 f'''(x) &= (-1)(-2)\frac{1}{(1+x)^3} \\
 f^{iv}(x) &= (-1)(-2)(-3)\frac{1}{(1+x)^4} = (-1)^3\frac{3!}{(1+x)^4} \\
 &\quad \dots \\
 f^n(x) &= (-1)^{n-1}\frac{(n-1)!}{(1+x)^n} \quad \exists n \geq 1
 \end{aligned}$$

Given the n -th term, first, let's generate the Python's list that stores the values of derivatives for $x = 3$ at $n = 10$.

```

from math import factorial

def df(x, n):
    return (-1)**(n-1) * factorial(n-1) / (1+x)**n

x2 = 3 # x = 3
n = 10
res = []
for i in range(1, n+1):
    res.append(df(x2, i))
print("%10.3f" % res[-1])

```

We define a custom function, `df`, that accepts two input parameters x and n and returns the value of derivative for n -th term. In a loop we iterate over i to be between 1 and 10 and append a list of **res** with computed value of the i -th derivative and print its last item out by making a reference to the last list's element by `res[-1]`. When run, the output is:

```

0.250
-0.062
0.031
-0.023
0.023
-0.029
0.044
-0.077
0.154
-0.346

```

Excellent. Now, let's employ symbolical computations of n derivatives of $\log(1+x)$ with a help of the `sympy` module in the following way:

```

from sympy import symbols, diff

sympy.symbols()
x = symbols("x")
f = "log(1+x)" # string

res2 = []
for i in range(1, n+1):
    f = diff(f, x) # find i-th derivative of f
    print("%2g-th derivative is %20s" % (i, f))
    v = f.subs(x, x2) # derivative value at x = 3

sympy.diff()
.subs()

```

```

res2.append(v)
print("\t\tits value at x = 3 is %10.3f\n" % res2[-1])

print(res == res2)

```

We make use of two functions, namely, `symbols` and `diff`. The former informs Python that a new variable `x` is from now on "symbolical". The latter allows for a symbolical differentiation of any expression containing such defined `x`. Therefore, our fundamental function, $\log(1+x)$, is given as a string variable in `f`.

What follows is the equivalent of the loop we created a moment ago. In the loop, first, we compute a symbolical expression of the i -th derivative of function and with the assistance of the `.subs` function we find its value at point $x = 3$. The results we save in a new list, `res2`, and in the end with check have we derived the identical derivatives' values in both approaches:

```

1-th derivative is      1/(x + 1)
its value at x = 3 is      0.250

2-th derivative is      -1/(x + 1)**2
its value at x = 3 is      -0.062

3-th derivative is      2/(x + 1)**3
its value at x = 3 is      0.031

4-th derivative is      -6/(x + 1)**4
its value at x = 3 is      -0.023

5-th derivative is      24/(x + 1)**5
its value at x = 3 is      0.023

6-th derivative is      -120/(x + 1)**6
its value at x = 3 is      -0.029

7-th derivative is      720/(x + 1)**7
its value at x = 3 is      0.044

8-th derivative is      -5040/(x + 1)**8
its value at x = 3 is      -0.077

9-th derivative is      40320/(x + 1)**9
its value at x = 3 is      0.154

10-th derivative is     -362880/(x + 1)**10
its value at x = 3 is      -0.346

```

```
True
```

Too easy. Please note that `sympy` delivers the results in a form of a string variable when `f` is inspected. The final test is a simple list comparison, element-wise. As we will learn in the next Chapter, the latter can be obtained if we apply:

```

import numpy as np
print(np.all(res == res2))

True

```

instead of `print(res == res2)` what enforces a boolean check, element-wise, vouchsafing that all elements of both lists are matching and are the same (in terms of floating-point precision).

Code 2.23

Using sympy verify that for $n > 2$:

$$\lim_{n \rightarrow \infty} \frac{1}{n^{-1/\sqrt{n}}} = 1$$

and find for what n one reaches 99.0% of the limit accuracy making use of Python's list. Add "*" to the list after the corresponding value.

In this example we will try to supplement the `.append` with the `.insert` function for the lists. The latter allows us to specify a position (index) at which a new object will be added to the list. By adding "*" to the list we may wish to make a mark inside the list itself. Such construction may be used for data separation into segments within further data processing or analysis.

Let's have a look at the complete code first:

```

from sympy import Symbol, limit
from sympy import init_printing
from math import trunc

sympy.init_printing()    init_printing() # a lovely sympy's printing included

n = Symbol("n")
expr = n**(-1/(n-1)) # define an expression

sympy.limit              print("lim_{n --> 9} %s = %.5f" % (expr, limit(expr, n, 9)))
                          print("lim_{n --> +oo} %s = %.5f\n" %
                                (expr, limit(expr, n, "+oo")))
                          print("%8s %12s %17s" % ("n", "exact", "approx"))

values = []
N = 1000

for i in range(3, N+1):
    x = float(expr.subs(n, i))
    values.append(x)

    if(i < 10) or (i == N):
        print("%8d %10.10f %17s" % (i, x, expr.subs(n, i)))
    elif(i == N-1):
        print("    ...")

# find 1st item corresponding to 99% of the limit accuracy
tmp = [trunc(x*100)/100 for x in values]
i = tmp.index(0.99) # find index; 642

print()
print(values[i-1:i+3])

# add "*" token to the list
.insert(index, object)  values.insert(i+1, "*")
                          print(values[i-1:i+4])

```

We employ the `limit` function from the `sympy` module for deriving limits. As you will find, the syntax is intuitive. In the next step, inside the loop, we derive the exact value for the limit at each $n \rightarrow i$ and append it to the global list with results, `values`. Solely for printing purposes, we allow a comparison of the exact value with the best approximation of it found by `sympy`. In order to inspect its

form, the function of `init_printing()` has be initialised in the beginning.

99.0% of the limit we find by making use of list comprehension. An expression `trunc(x*100)/100` creates a temporary list with exact values in **values** list, limited to three decimal places only. Though slow for large N , this step allows us quickly to localise the very first element equal to 0.99. We achieve that with the help of the `.index` function (as discussed earlier in this Chapter).

Adding "*" to the list at any requested position (index) is done by `.insert`. Painlessly. The output of [Code 2.23](#) delivers:

```
lim_{n --> 9} n**(-1/(n - 1)) = 0.75984
lim_{n --> +oo} n**(-1/(n - 1)) = 1.00000

n      exact      approx
3 0.5773502692    sqrt(3)/3
4 0.6299605249    2**(1/3)/2
5 0.6687403050    5**(3/4)/5
6 0.6988271188    6**(4/5)/6
7 0.7230200264    7**(5/6)/7
8 0.7429971446    2**(4/7)/2
9 0.7598356857    3**(3/4)/3
...
1000 0.9931091814  10**(332/333)/10

[0.98999, 0.99000, 0.99001, 0.99003]
[0.98999, 0.99000, '*', 0.99001, 0.99003]
```

where a funny "+oo" sign stands for $+\infty$ in [sympy](#). 99% accuracy of the limit at $n \rightarrow \infty$ is achieved for 642th term.

2.3.9. List Functions and Methods

In general, Python 3.5 offers the user with a finite number of built-in functions and, so-called, list methods. The table below summarises the key players in the game.

Let `x = [1, 3, 4]`, else otherwise specified

Function	Description
<code>list(y)</code>	coverts <code>y</code> (tuple) into a list <pre>>>> list((1, 3, 4)) [1, 3, 4]</pre>
<code>min(x)</code>	returns an item with minimal value <pre>>>> min(x) 1</pre>
<code>max(x)</code>	returns an item with maximal value <pre>>>> max(x) 4</pre>
<code>len(x)</code>	number of elements in the list <pre>>>> len(x) 3</pre>
<code>reverse(x)</code>	returns <code>x</code> with reversed order of items; raw iterator <pre>>>> list(reversed(x)) [4, 3, 1]</pre>

Methods	Description
<code>x.append(obj)</code>	<p>appends an object at the end of x; in-place</p> <pre>>>> x.append(5.6); x [1, 3, 4, 5.6] >>> x.append([6, 7]); x [1, 3, 4, 5.6, [6, 7]]</pre>
<code>x.extend(y)</code>	<p>extends the list by adding y; in-place</p> <pre>>>> x.extend([4.5]); x [1, 3, 4, 4.5] >>> x.extend([9, 10]) [1, 3, 4, 4.5, 9, 10]</pre>
<code>x.count(y)</code>	<p>returns the number of times y appeared in x</p> <pre>>>> x.count(4) 1 >>> [1, [2, 3], [2, 3]].count(2) 0 >>> [[2, 3], [2, 3]].count([2, 3]) 2</pre>
<code>x.index(y[, i1[, i2]])</code>	<p>the index of the first occurrence of y</p> <pre>>>> i = [1, 2, 3, 5, 3].index(3); i 2 >>> [1, 2, 3, 5, 3].index(3, i+1) 4</pre>
<code>x.insert(i, y)</code>	<p>inserts y at i-th index position in-place</p> <pre>>>> x.insert(1, "IBM"); x [1, IBM, 3, 4] >>> x.insert(-1, "AAPL") [1, IBM, 3, AAPL, 4]</pre>
<code>x.pop(i)</code>	<p>returns item at i-th position, then it removes it from x in-place</p> <pre>>>> x.pop(1) 3 >>> x [1, 4]</pre>
<code>x.remove(y)</code>	<p>removes the 1st occurrence of y in-place</p> <pre>>>> x = [6, "a", 7, "a", 9] >>> x.remove("a"); x [6, 7, a, 9]</pre>
<code>x.sort()</code>	<p>a tricky sort function in-place</p> <pre>>>> ["b", 7, "a", "c", 9].sort() Traceback (most recent call last): File "<stdin>", line 1, in <module> TypeError: unorderable types: int() < str() >>> x = [5, 7, 1, 6] >>> x.sort(); x [1, 5, 6, 7] >>> x = ["c", "ac", "ab", "D"] >>> x.sort(); x [D, ab, ac, c]</pre>
<code>x.copy()</code>	<p>does a copy of x; breaks the links</p> <pre>>>> y = x.copy(); y[0] = 2; x [1, 3, 4]</pre>

Further Reading

More on Lists, <https://docs.python.org/3.5/tutorial/datastructures.html#more-on-lists>

Statistics module, <https://docs.python.org/3.5/library/statistics.html>

Sympy module, <http://docs.sympy.org>

2.4. Randomness Built-In

2.4.1. From Chaos to Randomness Amongst the Order

You do not need Python to create chaos. Everything just happens fortuitously. At random. In a haphazard order. — I thought while studying the science standing behind. Behind what? Chaos, randomness, and order.

She is
delightfully
chaotic.
A beautiful
mess.

Steve Maraboli

From the day when we are born and grow up we intuitively feel the difference between chaos and order as two opposite states of being. As kids we learn how to create mess in a room while it was tidy. We, men, laugh when **women** behave in a chaotic way but amongst the havoc in their heads they act in a surprisingly organised fashion. At school they tell us that in the beginning there was a Bing Bang where our Universe took its origin from. Funny enough when we discover how different religions deliver contradictory explanations on the same topic. We observe the movement of celestial bodies, a structure of the DNA double helix, a landing on the Moon with a finite precision. So many things seem to be arranged or deterministic at any point in time. Regretfully, we cannot predict the exact numbers of seconds our heart will pump life through our veins.

A thin line between what we consider to be a truth and false sets all reference points we need for functioning. It is easier that way. We usually think of the **order** as of something that can be determined, organised, aligned, predicted, known from its initial state, derived, more than just estimated or foreseen. **Chaos** seems to be a tiny and turbulent perturbation introduced to the state of order. You cannot place an ideally polished spherical ball on a tip of a pyramid-like object at your desk. It will fall down. While the ball's initial position can be determined in a three-dimensional cartesian space, its final location is unknown. The moment you release the ball small vibrations of your fingers make an impact on its fate. Every single time you repeat that experiment the final location of the ball is non-deterministic. So where is a randomness in it? The best answer is if we think of **randomness** as of the characteristic of a process whose outcome is unpredictable. We may say that the ball always rolls down in a random direction therefore its behaviour is chaotic.

Comets orbiting the Sun are good examples of bodies in order. Their trajectories can be calculated and positions precisely found. But if any of them passes in a close vicinity of Jupiter, its gravity may cause a major disruption. The gravitational perturbation is too complex to be described by a closed-form analytical solution thus it introduces an element of chaos into the life of a comet. If different spatial configurations of the Jupiter's four biggest moons are considered, we expect different outcomes as it comes to the comet's position in time. By changing the initial moons' locations on their orbits we may generate a random set of the comet's possible (new) trajectories. Such process some people call as *using*

chaos in order to generate randomness. Here, the chaos is generated through a local strength of the gravitational force field. Now, by studying random (a large collection of possible) paths for that comet, we may determine the most likely trajectory as it passes Jupiter in its closest point. Therefore, randomness has its direct application in helping us to find the best approximation of a deterministic problem. We know that the comet will change its course but how much? We can predict it with the use of randomness (also referred to as Monte Carlo simulations).

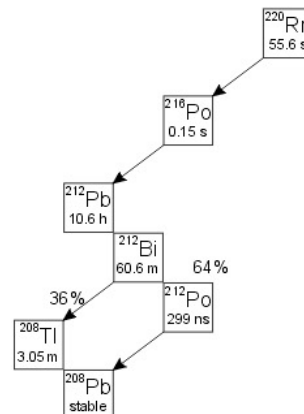
What does it mean that something is unpredictable or random? I am of the opinion that it simply cannot be determined with a finite precision. There is no single equation that could be used to definitely derive the result in the next (time) step.

In **quantitative finance** we make use of randomness for a number of reasons. **Option pricing** is one example. The process contains the randomness built-in. In order to price the option correctly we need to run a huge number of simulations. The outcomes will differ and based on the final distribution, the most likely result can be selected. The process is specified by the model. The randomness used—it's not. A gentle distinction between true and pseudo-randomness ought to be introduced for the clarity of their usefulness and pitfalls.

2.4.2. True Randomness

Do we need a chaotic process to describe a random process? Not necessarily. Again, the observation of nature delivers us some remarkable hints. Consider a thunderstorm. There is no way we are able to say when exactly the next **lightning** will take place (not even with a help of the Poisson process). If recorded, neither two-dimensional projection of the lightning on the sky plane nor the intensity of the thunder's noise can be predicted. Both are great examples of **true randomness** around us.

The physicists by observing the **decays of unstable isotopes** understood that they were unable to say when exactly the decay



could take place either! They only managed to describe in a quantitatively way the probability of a given transition.

The above diagram presents a decay chain of Radon-220 with the half-lives τ 's as given in (nano)seconds, minutes, and hours. A striking feature of that decay series of ^{220}Rn is its branching after the ^{212}Bi isotope. There is 64% of chances it will decay to unstable ^{212}Po before reaching stable ^{208}Pb and only 36% it will "choose" an alternative way via decay to ^{208}Tl . Both possible *routes* and the exact moment of ^{212}Bi decay are **truly** random!

The probability of ^{212}Bi decay in the time interval of length t given r could be described roughly by:

$$p(t|r) = \left(1 - 2^{-t/\tau}\right) \rightarrow \begin{cases} \text{decay to } ^{212}\text{Po} & \text{if } r \in (0; 0.64] \\ \text{decay to } ^{208}\text{Tl} & \text{if } r \in (0.64; 1] \end{cases}$$

where τ of 60.6 minute is the half-life of ^{212}Bi and r is a random variable drawn from a **uniform distribution** of real numbers between 0 and 1. A term in round brackets represents the probability of decay itself. This is the first step towards understanding how random processes can be discretised and used in **Monte-Carlo simulations**. If we drew r one million times counting how many times its value was between 0 and 0.64,

Code 2.24

```
from random import random

for k in range(5):
    n = 0
    for i in range(int(1e6)):
        r = random() # drawn from a uniform distribution
        if(r > 0 and r <= 0.64): n += 1
    print(100*n/1.0e6) # percent
```

we would get in five trials, e.g.

```
63.9041
63.9982
63.9002
64.0736
63.9802
```

Still the biggest unknown is: when in time t the decay will occur!? The probability of ^{212}Bi decay in the time interval of 1 sec is 0.00019. If we divide 1 sec into a time-series of nanoseconds,

$$t_1, t_2, \dots, t_N$$

we will end up with $N=1,000,000,000$ or 10^9 discrete values of the time moments. From our point of view that would be an approximated description of the set of all possibilities when the decay might happen. They **seem to be equally probable**. The question remains: which t_i , which one, Paweł? Well, we will never guess. That is the domain of quantum physics!

However, what we can do is we can **simulate** the decay process by drawing a random variable that represents the probability of decay.

If it is smaller than 0.00019 we assume that the decay took place. That's it. Simple as that! Analyse the following code:

Code 2.25

```
from random import random

tau = 60.6*60          # half-time of 212Bi (sec)
t = 1                  # time interval (sec)
p = 1 - pow(2, -t/tau)

for nsim in range(5): # number of simulations
    n1 = 0
    n2 = 0
    for trial in range(int(1e6)):
        if(random() < p): # we have a decay!
            r = random()
            if(r > 0 and r <= 0.64):
                n1 += 1 # a decay to 212Po
            else:
                n2 += 1 # a decay to 208Tl
    print(n1, n2)
```

We run five simulations of ²¹²Bi decay and count a number of times the decay to ²¹²Po and ²⁰⁸Tl occurred. For one million of trials in each simulation we get a fractions of events, for example:

```
114 64
121 57
108 64
127 59
139 70
```

This is, of course, justified by a very small probability of the decay process itself. As expected we observe a greater number of ²¹²Bi to ²¹²Po transitions.

The simulation of ²¹²Bi decay to stable ²⁰⁸Pb decay as shown in the diagram earlier we obtain by:

Code 2.26

```
from random import random

for nsim in range(100): # number of simulations
    n1 = 0
    n2 = 0
    for trial in range(int(1e6)):
        p = 1 - pow(2, -1/(60.6*60))
        if(random() < p):
            r = random()
            if(r > 0 and r <= 0.64): # 212Bi to 212Po
                p = 1 - pow(2, -1/299.0e-9)
                if(random() < p): # 212Po to stable 208Pb
                    n1 += 1
            else: # a decay of 212Bi to 208Tl
                p = 1-pow(2, -1/(3.05*60))
                if(random() < p): # 208Tl to stable 208Pb
                    n2 += 1
    print(n1, n2)
```

what, surprisingly, reveals

```
(128, 1)
(121, 0)
(127, 0)
(117, 2)
(105, 0)
...
```

that per 100 simulations we observe exactly two decays (per 1 million simulations) to ^{208}Pb via ^{208}Tl less than ten times. Without computers this sort of result would be nearly impossible to be estimated.

The abovementioned case studies and Python codes, in fact, constitute a bridge between true and pseudo-randomness. We were trying to describe a truly random process observed in nature making use of a large set of the discrete random numbers generated by computer. By now, we have only assumed that those numbers had been "magically" random and we have trusted the process of their selection. The devil resides in the Standard Library's module of `random` and the engine responsible for desired outcomes can be neglected. Is it so?

In case of `random` we deal with a certain pseudo-random number generator (PRNG; an algorithm) that appears to be efficient, fast, and highly reliable. We may deduce that it passed lots of tests before it has been approved for a use within the Python language. Most probably the algorithm is complex and it took years of research to design and test it. How unique it is? We will see, soon.

A link between **true random number generators** (TRNGs) and PRNGs has been explored widely. For instance, if we assume the source of true randomness as a radioactive decay or the noise of a semiconductor diode, then what we may consider by true random numbers would be the output of TRNGs, i.e. the results of physical experiments which are considered to be random. However, true random numbers are relatively difficult to be collected. The process requires plenty of external devices, specific environmental conditions to be met, storage, and proper distribution channels. Additionally, such a setup may be biased and returns quasi-true random numbers due to technical issues or data processing.

Certain hardware solutions appeared. The most noticeable one was *Lavarand* — designed by Silicon Graphics—a **hardware random number generator** (HRNG) that produced a stream of "seed" numbers for internal PRNG based on the photo-documentation and analysis of patterns of the flowing material inside the lava lamps. Since the seeds were considered to be truly random—that made Lavarand nearly perfect TRNG. Nearly.

The technological solutions in the domain of HRNG stepped fast forward after 2000. Nowadays, one considers devices utilising physical phenomena with or without the **quantum-random properties**, e.g. nuclear decay, photons traversing materials, shot noise, thermal noise, atmospheric noise — just to name few. The cost of construction may vary and is mainly dictated by a technology used and the customers' demand.

The general purpose of HRNG is to deliver a device both portable and of the **cryptographic security**. The latter plays a huge role in

information transmission, e.g. on the Web (bank transfers, ATMs, credit card payments, etc.).

Disclaimer

The author of this book has no business connection nor interest in advertising the products of ID Quantique SA, Switzerland. The information included is solely for the educational purposes.

One example of a modern HRNG is *Quantis* of ID Quantique SA, Genève, Switzerland — a hardware random number generator which uses the fundamentally random nature of **quantum optics** as a source of true randomness (<http://www.idquantique.com/random-number-generation/quantis-random-number-generator>). The company offers its HRNG in three versions: as a USB device, PCI Express (PCIe) board, and PCI board with a random stream between 4Mbits/sec and 16Mbits/sec. Its application range spans from lotteries and gaming, to cryptography, IT security applications, quantum cryptography, password and PIN number generation, random seed generation, mobile prepaid systems, numerical simulations, and statistical research.

Within a price range between €990 and €2990 we gain the "quantum" true random numbers in our computer. HRNG is not PRNG. The former generates a series of bits. They may be *uniform random* if they have expectation 0.5 and are independent. The term *uniform random* can be thought of as a sequence of n bits interpreted as a binary number that will be *uniform* on the integers 0 to $2^n - 1$. Therefore, the output of HRNG should be oriented towards seeing (testing) how closely the output from the generator is uniform random.

The same approach applies to PRNGs which use a formula for generating numbers. Notably, they are all available to us—free of charge. Python's `random` engine falls into that category too. ©

Let's inspect some acres of the devil's playground in order to understand the most crucial aspects of the **randomness built-in**.

2.4.3. Uniform Distribution and K-S Test

There is a lot of power standing behind the concept of the uniform distribution of (random) numbers in (a, b) interval, or in $(0, 1)$ for simplicity. As we have convinced ourselves in Section 2.4.2, every number picked at random and to be between 0 and 1 seems to be equally probable.

The **uniform distribution** is defined by the function:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

of the uniform density between a and b , with the expected value and variance equal to:

$$E(X) = \frac{b+a}{2}, \quad Var(X) = \frac{(b-a)^2}{12},$$

respectively. For $(0, 1)$ interval the variance is simply $1/12$ and suggests a fairly simple design of a **test for randomness**: we draw a large number of "uniform random" numbers from $(0, 1)$ interval and compute their variance, $Var(X)$. It should be near a theoretical value, i.e.:

$$\left| Var(X) - \frac{1}{12} \right| < \epsilon$$

However, the distance (or precision) is parametrised and does not constitute, in fact, any solid proof that we might deal with a sample of the uniformly distributed random numbers.

A much more elegant way to test any sample of **random variables** (**rvs**) is the application of the **Kolmogorov-Smirnov test** (or K-S test) for goodness of fit. Within this test, we compare the cumulative distribution function (**c.d.f.**) of sample data with, in our case, the uniform cumulative distribution function, i.e. $X(x)$ against $U(x)$. Under the null hypothesis these two distributions are identical, $X(x) = U(x)$. The alternative hypothesis can be either two-sided, less or greater. K-S test aims at the verification of the maximal distance between both cumulative distributions, returning *p-value* as a result. For the case when:

$$p\text{-value} < \alpha$$

i.e. *p-value* is less than assumed significance level, we reject the null hypothesis at that level. Let's analyse the following case.

Say, we have a set of 65 rvs to be in $(0, 1)$ interval and stored as a Python list **x**. The source of data is unknown. Making use of `scipy.stats` submodule (more on `scipy` in *Volume II* of *Python for Quants*) we employ K-S test directly to **x**. In addition, we generate a new list of rvs with a help of the `random` library and compare the cumulative distributions of both data sets:

Code 2.27

```
from random import random
import statistics as st
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# an unknown data set
x = [0.85906464, 0.45391992, 0.30094518, 0.35947101, 0.65084560,
     0.59774770, 0.85093582, 0.67892358, 0.57949969, 0.51680545,
     0.53016296, 0.36267706, 0.51915569, 0.34386377, 0.56175338,
     0.63607077, 0.35634497, 0.52965109, 0.46146248, 0.27364399,
     0.51165262, 0.38406413, 0.63410533, 0.86702544, 0.45153084,
     0.79633974, 0.80685862, 0.44867997, 0.68959574, 0.86788886,
     0.24311945, 0.35210901, 0.57610074, 0.49695454, 0.16283205,
     0.55022556, 0.45003607, 0.45194890, 0.30506118, 0.44786555,
     0.49293125, 0.77768506, 0.37584660, 0.56049562, 0.76204812,
     0.74261763, 0.63438029, 0.47814883, 0.35056373, 0.57848429,
     0.49396983, 0.92532668, 0.38941321, 0.16282464, 0.59157518,
     0.17022039, 0.77477934, 0.81258861, 0.38457275, 0.50546050,
     0.62353637, 0.62423579, 0.37363242, 0.41314893, 0.42630024]
x.sort()

# a list of uniformly distributed rvs
u = [random() for _ in x]
```

`scipy.stats.kstest()`

Try to deduce why in Code 2.27 we sort all three lists (**x**, **u**, and **t**) with random numbers.

```
u.sort()

# Perform K-S test (deriving p-values)
_, px = stats.kstest(x, 'uniform') # X(X) vs U(X)
_, pu = stats.kstest(u, 'uniform') # u(X) vs U(X)

print(px, pu); print()

print(st.mean(x), abs(st.variance(x)-1/12))
print(st.mean(u), abs(st.variance(u)-1/12))

# a large set of uniform rvs
t = [random() for _ in range(1000)]; t.sort()

# Plot cumulative distribution functions
plt.figure(1)
plt.step(x, np.cumsum(x)/np.sum(x), label="c.d.f. for x")
plt.step(u, np.cumsum(u)/np.sum(u), 'r', label="c.d.f. for u")
plt.legend(loc=2)
plt.show()
```

The `scipy.stats`'s function of `kstest` works as a black-box returning two arguments: K-S statistics (suppressed by underscore token in the code) and *p-value*. First, we test the cumulative distribution function of data sample **x** against the uniform cumulative distribution function, deriving the corresponding *p-value* of **px**. Next, we perform the same test for a sample **u** containing random variables, now, ensured to be drawn from the uniform distribution thanks to the `random` function from the `random` module.

By printing results, we obtain:

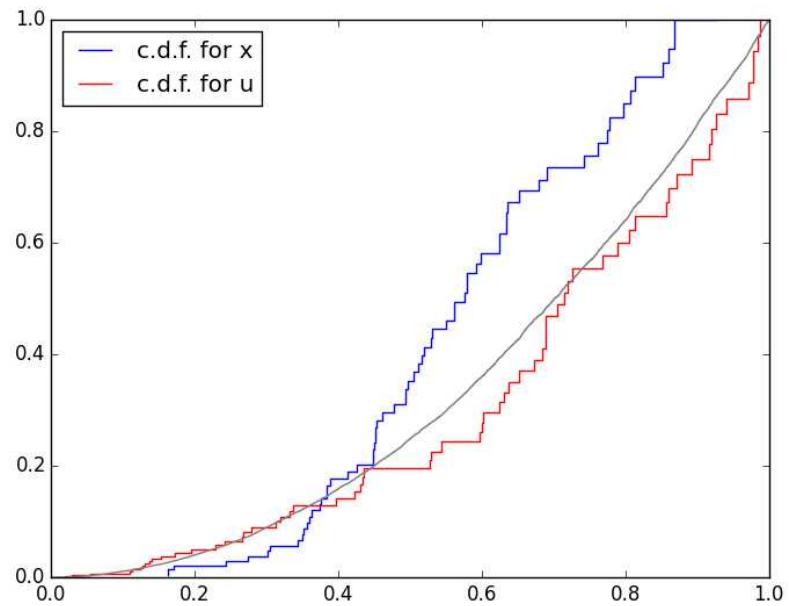
```
0.00113077233788 0.729760346867

0.5283342367692307 0.04904732615158596
0.49189508566570306 0.0040929025816235365
```

The first line displays *p-values*, **px** and **pu**. Immediately, we notice that for our unknown sample, **px** is less than, say, *0.05* (95% confidence level) allowing us to reject the null hypothesis. For the case of **u** set, as expected, the K-S test indicates at the agreement of the tested cumulative distribution functions at the *0.05* level. The absolute difference between the computed sample variances and the theoretical $1/12$ is least for **u** set too.

The visualisation of our results we present as a comparison of both cumulative distribution functions for **x** and **u** data set, employing some helping functions from the `numpy` module and the plotting library of `matplotlib`. For now, we skip the discussion of those functions however their appearance has been aimed at the introduction of new features for the completeness of our analysis.

From the plot (see next page) one can notice that a cumulative distribution function for 1000 uniform rvs (black line) is a good representation of the "model" c.d.f. assumed by the `kstest` function as a reference level. The maximal distance for **x**'s c.d.f is greater than for **u**'s c.d.f., as expected.



Okay, let's raise the bar.

So far, still, we haven't said anything about the engine generating uniformly distributed rvs. All we know is that the function of `random.random()` returns a floating-point number between 0 and 1. The question is: can we use K-S test in order to verify "how much" those uniform random numbers are random? And the answer is: sure! However, this time, we need to increase our confidence level from 95% to 99.99% to see what is going to happen.

Have a look:

Code 2.28

```
from random import random
import scipy.stats as stats
import numpy as np

num_tests = 100000
num_rejects = 0
alpha = 0.0001

for i in range(num_tests):
    data = [random() for _ in range(1000)]
    _, pval = stats.kstest(data, 'uniform')
    if pval < alpha:
        num_rejects += 1
ratio = float(num_rejects) / num_tests
print('{} / {} = {:.5f} rejects at rejection level {}'.format(
    num_rejects, num_tests, ratio, alpha))
```

You may gain a significant speed-up if you replace:

```
[random() for _ in range(1000)]
```

with

```
np.random.random(1000)
```

In this code, we run 100,000 simulations. In each of them, we generate a list of 1000 uniform rvs (**data** list). Based on our previous knowledge, we reject the null hypothesis at 0.0001 significance level if the computed *p-value* is less than that the threshold. If so, we count the total number of rejects. In the end, we use an old-school style of `print` function (which description has been omitted within this book; you are encouraged to explore

it elsewhere) to check a number of unsuccessful tests in `num_tests` simulations.

Surprisingly, we note that, for example:

```
13/100000 = 0.00013 rejects at rejection level 0.0001
```

i.e. 0.013% of the uniform random samples occurred not to pass K-S test at 0.0001 significance level. Now, by increasing the confidence level to 99.999% and re-running [Code 2.28](#), we get:

```
0/100000 = 0.00000 rejects at rejection level 1e-05
```

That allows us to reach the statistical confidence around the function of `random.random()` responsible for the sophisticated randomisation of numbers we asked for. However, do not be excited so much! We know it well that in statistics all is about the confidence level and you may find a number of statistical tests that would reject the null hypothesis at the same level.

Wishfully, we want to believe we remain "certain enough" to claim highly significant randomness. Before we make a remark on much more advanced tests for randomness, let's begin from the most simplistic algorithm for a pseudo-random numbers delivery to our doorsteps—the Linear Congruential Generator.

2.4.4. Basic Pseudo-Random Number Generator

The idea standing behind PRNGs is simple. We demand the highest possible degree of randomness, the irreproducibility after any given period (of time, of digits, etc.), fast computation times, uniqueness of the algorithm, and difficulty in prediction of the next random number. What appears to be random at first glance may, in fact, be periodic!

Linear Congruential Generators is the family of PRNGs defined by their fundamental iterative formula:

$$R_{i+1} = (aR_i + c) \bmod m$$

where for $i = 0$ the R_0 is called the "seed", a , c , and m are constant, and the family is often denoted by $LCG(a, c, m)$ of mixed type ($c > 0$) or $MLCG(a, m)$ of multiplicative ($c = 0$) type. A LCG returns a sequence of pseudo-random integers R_1, R_2, \dots between 0 and $m - 1$. Each R_i is, in next step and if required, scaled into the interval of $[0, 1)$. If the multiplier a is a primitive root modulo m and m is prime, the period of this generator is $m - 1$. In Python we obtain the corresponding solution in the following way:

Code 2.29

```
R = 2016      # the "seed"
a = 45319     # the "multiplier"
c = 171       # the "increment"
m = 2**5      # the "modulus"

N = 2*m - 1
```

```
# LCG(a,c,m)
rnd = []
for i in range(N):
    R = (a*R + c) % m
    rnd.append(R)

print(rnd)
```

which generates a list of the integer random numbers:

```
[11, 24, 19, 16, 27, 8, 3, 0, 11, 24, 19, 16, 27, 8, 3, 0, 11,
24, 19, 16, 27, 8, 3, 0, 11, 24, 19, 16, 27, 8, 3, 0, 11, 24, 19,
16, 27, 8, 3, 0, 11, 24, 19, 16, 27, 8, 3, 0, 11, 24, 19, 16, 27,
8, 3, 0, 11, 24, 19, 16, 27, 8, 3]
```

The repeatability of the sequence can be quickly visually detected.

Since the algorithm is plain and iterative, one can generate a sequence of pseudo-random numbers with "undetected" pattern of periodicity. How? Simply by making m large enough. That's not all. A special combination of a and c is also important. For example in the Java's `java.util.Random` function: $a = 25214903917$, $c = 11$, and $m = 281474976710656$.

Though fast, LCGs are highly unreliable for many computational tasks and simulations. It cannot be used for the parallel computing purposes where different threads may call and use the same seeds while running concurrently. Jones (2000) discusses why the importance of using "your own" PRNG is so crucial. He also provides with some hints on the selection of a good PRNG and C/C++ codes of the modern, state-of-the-art generators.

Detecting Pseudo-Randomness with `re` and `collections` Modules

A simple formula which defines LCG—eventually can be hacked. Below we will write two simple Python programs that perform that task. Both will employ some tricks with a help of the `re` (regular expressions) module from the Python's Standard Library. Namely, we will aim at conversion of any sequence of random numbers generated by LCG into a single long string expression and detection of the similar, repeatable patterns.

We begin as previously:

Code 2.30

LCG Test for Non-Periodic Randomness

```
import re

R = 2016      # the "seed"
a = 45319    # the "multiplier"
c = 171      # the "increment"
m = 2**5     # the "modulus"
N = 2*m - 1

# LCG(a,c,m)
rnd = []
for i in range(N):
    R = (a*R + c) % m
    rnd.append(R)
```

```
print("rnd:\n%s\n" % rnd)

# cover rnd list into single string expression
seq = "".join(str(s) for s in rnd)
print("rnd sequence:\n%s" % seq)
```

where a list of random numbers **rnd** is transformed into a single string, **seq**, making use of `"".join(par)` function. Though it may look odd, the function takes its input as defined by a *par* parameter. In our case, it is a list comprehension that converts every item from **rnd** list (a random number) into an individual string by `str(item)` and next, `.join(par)` function joins (glues) all those strings together with a separator specified between `""` (here: no space nor any token assumed). If you run that fragment of code, you will obtain the following output:

```
rnd:
[11, 24, 19, 16, 27, 8, 3, 0, 11, 24, 19, 16, 27, 8, 3, 0, 11,
 24, 19, 16, 27, 8, 3, 0, 11, 24, 19, 16, 27, 8, 3, 0, 11, 24, 19,
 16, 27, 8, 3, 0, 11, 24, 19, 16, 27, 8, 3, 0, 11, 24, 19, 16, 27,
 8, 3, 0, 11, 24, 19, 16, 27, 8, 3]

rnd sequence:
1124191627830112419162783011241916278301124191627830112419162783
011241916278301124191627830112419162783
```

Continuing 2.30,

```
k = int(len(seq)/2.)

T = []
npatt = 0
ntimes = 0

while(k >= 2):
    patt = seq[0:k+1]
    n = 0
    pos = []

    for match in re.finditer(patt, seq):
        # print(match)
        s = match.start()
        e = match.end()
        if(s != 0 and e != 0):
            n += 1
            pos.extend([s, e])

    .extend
```

A list method. See Section 2.3.9.

by **k** we define a half of the length of string **seq** what in our case gives $k=51$ at $\text{len}(\text{seq})=103$. Since the half of the sequence is the longest possible repeatable pattern in **seq**, we extract it first and store the result within **patt** variable.

The loop `for` is the goodie which we find in the Standard Library. It looks for pattern (**patt**) within a whole sequence of **seq**. If you uncomment the `print(match)` function in the code, for the first iteration you will see the Hell's Kitchen at work:

```
<_sre.SRE_Match object; span=(0, 52), match='1124191627830112419162783011241916278301124191627'>
<_sre.SRE_Match object; span=(0, 51), match='1124191627830112419162783011241916278301124191627'>
<_sre.SRE_Match object; span=(52, 103), match='1124191627830112419162783011241916278301124191627'>
<_sre.SRE_Match object; span=(0, 50), match='1124191627830112419162783011241916278301124191627'>
<_sre.SRE_Match object; span=(52, 102), match='1124191627830112419162783011241916278301124191627'>
<_sre.SRE_Match object; span=(0, 49), match='1124191627830112419162783011241916278301124191627'>
<_sre.SRE_Match object; span=(52, 101), match='1124191627830112419162783011241916278301124191627'>
<_sre.SRE_Match object; span=(0, 48), match='112419162783011241916278301124191627830112419162'>
```

[illegible]

For every entry, **seq**'s index corresponding to **match**'s first and last character is extracted (**s** and **e** variables) and both added to the list with positions, **pos**. A number of the same patterns **patt** in **seq** is denoted by running **n** variable.

Now, writing further 2.30,

```

if(n > 2) and (float(patt)/N > 10):
    print("\npattern: '%s'" % patt)
    print(" detected %d times" % n)
    for i in range(0, len(pos)-1, 2):
        print("\tat %d:%d" % (pos[i], pos[i+1]))

    dist = []
    dist = [(pos[i+1]-pos[i]) for i in range(1, len(pos)-1)
            if i % 2 != 0]

    print(" distances between patterns: %s" % dist)
    test = dist.count(dist[0]) == len(dist)
    print(" equally separated: %s" % test)
    T.append(test)
    if(test):
        npatt += 1
        ntimes += n

k -= 1

```

if at least three the same patterns occur, we display information on the pattern (the kind, number of occurrences, *start:end* index position) and compute the distance, **dist**, between those patterns in **seq**. It is important for the further test. Namely, we want to make sure that all $n > 2$ patterns are equally placed in **seq** among each other. The variable of **dist** is a Python's list and **test** employs

the list method of `x.count(y)` counting a number of `y` in `x` (see the previous Chapter).

After all, we reduce `k` by 1, i.e. we look for a new pattern shorter by one character. The output of this section is:

```
pattern: '1124191627830112419162783'
detected 3 times
  at 26:51
  at 52:77
  at 78:103
distances between patterns: [1, 1]
equally separated: True

pattern: '112419162783011241916278'
detected 3 times
  at 26:50
  at 52:76
  at 78:102
distances between patterns: [2, 2]
equally separated: True

pattern: '11241916278301124191627'
detected 3 times
  at 26:49
  at 52:75
  at 78:101
distances between patterns: [3, 3]
equally separated: True

...

pattern: '11241'
detected 7 times
  at 13:18
  at 26:31
  at 39:44
  at 52:57
  at 65:70
  at 78:83
  at 91:96
distances between patterns: [8, 8, 8, 8, 8, 8]
equally separated: True

pattern: '1124'
detected 7 times
  at 13:17
  at 26:30
  at 39:43
  at 52:56
  at 65:69
  at 78:82
  at 91:95
distances between patterns: [9, 9, 9, 9, 9, 9]
equally separated: True
```

We add the final test for detection of the naive periodic patterns within any sequence of numbers considered as a string:

```
test1 = (len(T) == sum(T))
test2 = (sum(T) > 0)

print("\n%s" % ("-"*55))
print(" LCG Test for Non-Periodic Randomness:"),
if(test1 and test2):
    print(" FAILED")
    print(" %d distinct pattern(s) detected, in total, %d times"
          % (npatt, ntimes))
```

```
elif(test1 and not test2):
    print(" PASSED\n Increase N from %d to %d and re-run" %
          (N, N*10))
print("%s" % ("-"*55))
```

displaying:

```
-----
LCG Test for Non-Periodic Randomness:
FAILED
22 distinct pattern(s) detected, in total, 105 times
-----
```

Within [Code 2.30](#) we have shown a new feature of Python at work: operations on strings used to break LCG and detect the periodicity in its random numbers. Interestingly, if we omit, in the very beginning, the list of **rnd** to be generated by LCG algorithm and, instead of it, we replace that fragment with:

```
from random import random

N = 103
rnd = [random() for _ in range(N)]
```

the output from our test will be:

```
-----
LCG Test for Non-Periodic Randomness:
PASSED
Increase N from 103 to 1030 and re-run
-----
```

Somehow, any sequence of pseudo-random numbers generated by **random.random** function (regardless of *N*) passes our simple test.

Now, one may think of a very similar test employing the functionality of the **re** library as shown above, however, this time slightly modified:

Code 2.31

LCG Test for Non-Periodic Randomness based on examination of the frequency distribution of the random numbers in **seq**.

```
from re import finditer
from random import randrange, random
from collections import Counter
import statistics as st

R = 2016      # the "seed"
a = 45319    # the "multiplier"
c = 171      # the "increment"
m = 2**5     # the "modulus"
N = 2*m - 1

# LCG(a,c,m)
rnd = []
for i in range(N):
    R = (a*R + c) % m
    rnd.append(R)

rnd2 = [randrange(0, max(rnd)) for _ in range(len(rnd))]
# rnd = rnd2
```

```

seq = "".join(str(s) for s in rnd)

print("rnd:\n%s\n" % rnd)
print("rnd sequence:\n%s\n" % seq)

ra = []
for j in range(max(rnd)):
    patt = str(j)
    pos = []
    n = 0

    for match in finditer(patt, seq):
        s = match.start()
        e = match.end()
        if(s != 0 and e != 0):
            n += 1
            pos.extend([s, e])

    if(n > 1):
        dist = []
        dist = [(pos[i+1]-pos[i]) for i in range(1, len(pos)-1)
                if i % 2 != 0]
        dist.sort()
        print("\nPattern '%s' detected %d times, dist: %s" %
              (patt, n, dist))
        c = Counter(dist)
        v = c.values()
        # print(c)
        # print(list(v))
        # print(list(v).count(1)) # how many "1" in list "v"?

        ratio = 1.0 - list(v).count(1)/float(len(v))
        ra.append(ratio)

print("\n%s" % ("-"*45))
print(" LCG Test for Non-Periodic Randomness (2)"),
if(st.mean(ra) == 1.0):
    print(" FAILED")
else:
    print(" PASSED")
print("%s" % ("-"*45))

```

collections.Counter

delivering the following results:

```

rnd:
[11, 24, 19, 16, 27, 8, 3, 0, 11, 24, 19, 16, 27, 8, 3, 0, 11,
24, 19, 16, 27, 8, 3, 0, 11, 24, 19, 16, 27, 8, 3, 0, 11, 24, 19,
16, 27, 8, 3, 0, 11, 24, 19, 16, 27, 8, 3, 0, 11, 24, 19, 16, 27,
8, 3, 0, 11, 24, 19, 16, 27, 8, 3]

rnd sequence:
1124191627830112419162783011241916278301124191627830112419162783
011241916278301124191627830112419162783

Pattern '0' detected 7 times, dist: [12, 12, 12, 12, 12, 12]
Pattern '1' detected 31 times, dist: [0, 0, 0, 0, 0, 0, 0, 1, 1,
                                     1, 1, 1, 1, 1, 1, 2, 2, 2,
                                     2, 2, 2, 2, 2, 6, 6, 6, 6,
                                     6, 6, 6]
Pattern '2' detected 16 times, dist: [5, 5, 5, 5, 5, 5, 5, 5, 5, 6,
                                     6, 6, 6, 6, 6, 6]
Pattern '3' detected 8 times, dist: [12, 12, 12, 12, 12, 12, 12, 12]
Pattern '4' detected 8 times, dist: [12, 12, 12, 12, 12, 12, 12, 12]
Pattern '6' detected 8 times, dist: [12, 12, 12, 12, 12, 12, 12, 12]
Pattern '7' detected 8 times, dist: [12, 12, 12, 12, 12, 12, 12, 12]
Pattern '8' detected 8 times, dist: [12, 12, 12, 12, 12, 12, 12, 12]
Pattern '9' detected 8 times, dist: [12, 12, 12, 12, 12, 12, 12, 12]
Pattern '11' detected 7 times, dist: [11, 11, 11, 11, 11, 11, 11]
Pattern '12' detected 8 times, dist: [11, 11, 11, 11, 11, 11, 11, 11]

```



```

Pattern '16' detected 8 times, dist: [11, 11, 11, 11, 11, 11, 11]
Pattern '19' detected 8 times, dist: [11, 11, 11, 11, 11, 11, 11]
Pattern '24' detected 8 times, dist: [11, 11, 11, 11, 11, 11, 11]

```

```

-----
LCG Test for Non-Periodic Randomness (2)
FAILED
-----

```

As you may see in the code, we have not done too much till the definition of **dist** list inside the loop. However, the design of the loop has been thought differently. We scan the whole space of integer numbers from 0 to the maximal number found in the list of random numbers, **rnd**, generated by LCG. In this test, for each number to be assumed as a pattern (**patt**), we want to compute all distances as in 2.30, however, this time, additionally to examine the frequency of the pattern appearance.

By the introduction of that modification to the previous code, we gain an opportunity to use a new function of **Counter** from the **collections** module. If any number, say, 3, is present in the list:

```
>>> x = [1, 3, 4, 1, 5, 3, 7]
```

the result of the **Counter** function, applied onto **x** would be:

```

>>> c = Counter(x)
>>> c
Counter({1: 2, 3: 2, 4: 1, 5: 1, 7: 1})

```

i.e. we quickly find that 3 appears two times in **x**. Extracting the values and/or keys follows:

```

>>> list(c.keys())
[1, 3, 4, 5, 7]
>>> list(c.values())
[2, 2, 1, 1, 1]

```

as **Counter** returns data in the form of Python's dictionary (more on dictionaries in Chapter 2.5).

If we detect the pattern **patt**, say, of 11 in **seq** seven times to be equidistant, the calculated **ratio** must be equal 1. This is not the case if we uncomment in 2.31:

```

rnd2 = [randrange(0, max(rnd)) for _ in range(len(rnd))]
rnd = rnd2

```

and re-run the program. First, by doing so, we overwrite LCG list of **rnd** with the uniform integer random numbers falling between 0 and maximal value found in **rnd**. It is a very handy function coming from the **random** module you may wish to use a number of times in your future programs. Secondly, the output delivers, e.g.

```

rnd:
[6, 7, 17, 24, 14, 9, 23, 17, 14, 1, 20, 10, 6, 3, 6, 0, 6, 11,
18, 24, 13, 25, 25, 24, 15, 21, 9, 4, 19, 18, 26, 26, 20, 5, 1,
23, 20, 8, 2, 2, 21, 1, 0, 13, 6, 3, 22, 16, 14, 7, 14, 2, 12,
16, 8, 21, 14, 14, 2, 16, 25, 17, 6]

```

random.randrange

returns a uniform integer rv in $[a, b)$ interval. For example:

```

>>> [randrange(0, 11) for _ in
     range(10)]
[5, 3, 3, 2, 5, 10, 7, 7, 9, 4]

```

```

rnd sequence:
67172414923171412010636061118241325252415219419182626205123208
222110136322161471421216821141421625176

Pattern '0' detected 6 times, dist: [1, 3, 5, 6, 30]
Counter({1: 1, 30: 1, 3: 1, 5: 1, 6: 1})
[1, 1, 1, 1, 1]
5

Pattern '1' detected 28 times, dist: [0, 0, 0, 0, 1, 1, 1, 1, 1,
1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 5, 6, 7, 8, 8]
Counter({1: 7, 2: 6, 0: 4, 3: 4, 8: 2, 4: 1, 5: 1, 6: 1, 7: 1})
[4, 7, 6, 4, 1, 1, 1, 1, 2]
4

Pattern '2' detected 23 times, dist: [0, 0, 0, 1, 1, 1, 1, 1, 1,
2, 2, 3, 3, 3, 3, 4, 5, 6, 7, 7, 7, 12]
Counter({1: 6, 3: 4, 0: 3, 7: 3, 2: 2, 4: 1, 5: 1, 6: 1, 12: 1})
[3, 6, 2, 4, 1, 1, 1, 3, 1]
4

...

Pattern '24' detected 3 times, dist: [6, 23]
Counter({6: 1, 23: 1})
[1, 1]
2

Pattern '25' detected 3 times, dist: [0, 59]
Counter({0: 1, 59: 1})
[1, 1]
2

-----
LCG Test for Non-Periodic Randomness (2)
PASSED
-----

```

Since we deal with the highest degree of randomisation delivered by the `random.randrange` function, we do not expect to detect the equidistant spaces between the same numbers (patterns) in `seq`. For example, pay attention to the pattern '24'. It is detected 3 times in `seq` but separated, first, by 23 and next by 6 digits:

```

67172414923171412010636061118241325252415219419182626205123208
222110136322161471421216821141421625176

```

A variable `v` in 2.31 stores information of how many different distances between the patterns occurred. If '24' were equally separated, `ratio` would be equal 1. This time, for all patterns, the mean value for all derived `ratios` is different than 1 what justifies the result of the test.

A similar test one can perform by employing `random.random()` function. Code 2.31 could be supplemented with a manual conversion, e.g.:

```

from math import trunc
rnd2 = [trunc(random()*10e5) for _ in range(len(rnd))]
rnd = rnd2

```

The possibilities of modifications of both codes 2.30 and 2.31 are nearly endless. Try yourself. Happy hacking! ☺

2.4.5. Mersenne Prime Numbers

As we have mentioned in the previous Section, the choice of the proper constants for the usefulness of the fundamental Linear Congruential Generator is peculiar. LCG may reach the full period of m , inter alia, if m itself is a co-prime. A much "stronger" prime numbers constitute so-called **Mersenne Primes** defined as:

$$M_n = 2^n - 1$$

where n is a prime number. It has been shown that such condition is weak and an additional test which would allow for a quick discovery of the Mersenne prime numbers is the **Lucas-Lehmer** test for primality, namely, let:

$$x_0 = 4 \quad \text{and} \quad x_i = x_{i-1}^2 - 2$$

then M_n is prime if and only if x_{n-2} is divisible by M_n . Thanks to that, the search for "large" primes becomes accessible for anyone with an access to a personal computer.

Up to date, the largest, 48th (found but unconfirmed) Mersenne Prime corresponds to $n = 57885161$. The number itself has over 17 million digits and has been found in 2013 within the Great Internet Mersenne Prime Search (GIMPS) Project.

It is not too difficult to understand why the "large" Mersenne Primes are so attractive for the needs of simple LCGs. The key is the period of a generator. The longer the better. If fact, if any complex but efficient algorithm generating a stream of pseudo-random numbers displays a very long period of repetition, in the limit of its usefulness, it remains safe for various applications.

As an exercise in Python, let's provide with a full algorithm for the search of Mersenne Primes (courtesy of Joe Kelley) supplemented with a new language feature: the **measurement of time**.

Code 2.32

Mersenne Prime Search

```
import time

def mersenne(n):
    return 2 ** n - 1

def mersenneisprime(n):
    if not(simpleisprime(n)):
        return False
    if(n == 1):
        return False
    if(n == 2):
        return True
    m = mersenne(n)
    x = 4 # Lucas-Lehmer test for primality
    for i in range(n-2):
        x = (x*x-2) % m
    return (x == 0)
```



After the 23rd Mersenne prime was found at the University of Illinois, the mathematics department was so proud that the chair of their department, Dr. Bateman, had their postage meter changed to stamp $2^{11213}-1$ is prime on each envelope.

`time.time()`

while True:

An infinite loop that never terminates itself.

```
def simpleisprime(n):
    if(n == 1):
        return False
    div = 2
    while(div*div <= n):
        if(n % div == 0):
            return False
        div = div+1
    return True

# main program
n = 0
t0 = time.time()

while True:
    if mersenneisprime(n):
        t = time.time() - t0
        print("M_ "+str(n)+" = "+str(mersenne(n)))
        print("\tlength:\t\t %g digits" % len(str(mersenne(n))))
        print("\tfound in:\t %5fs (%.5fh)\n" % (t, t/3600.))
    n += 1
```

In our program we define three custom functions. The last one, `simpleisprime`, checks whether a number is a simple prime number. The second one, `mersenneisprime`, makes use of the last one and additionally applies Lucas-Lehmer test assuming any given n based on which we define a Mersenne number utilising a simple `mersenne` function.

A real joy delivers the main program which we commence by defining a time-related variable of `t0`. Python's Standard Library of `time` facilitates the measurement of the **elapsed time between any two lines of code**. In our case, the command of:

```
t0 = time.time()
```

stores an inner integer number (expressed in seconds from some initial time marker). This is where we start counting time since the inception of our private "Mersenne Prime Search". Next, in the loop, we apply a quick test for the Mersenne primality, i.e.,

```
if mersenneisprime(n):
```

and if its boolean value is `True` then we know that a new M_n prime has been discovered and, immediately, we perform the another measurement of time in order to calculate the **time** consumed for its search; defined as a difference.

More on `time` module and time measurements see Appendix.

Simple in coding. More exciting in execution! Have a look:

```
M_2 = 3
length:          1 digits
found in:    0.00000s (0.00000h)

M_3 = 7
length:          1 digits
found in:    0.00005s (0.00000h)

M_5 = 31
length:          2 digits
```

```

found in:    0.00006s (0.00000h)

M_7 = 127
length:      3 digits
found in:    0.00008s (0.00000h)

M_13 = 8191
length:      4 digits
found in:    0.00010s (0.00000h)

M_17 = 131071
length:      6 digits
found in:    0.00011s (0.00000h)

M_19 = 524287
length:      6 digits
found in:    0.00013s (0.00000h)

M_31 = 2147483647
length:      10 digits
found in:    0.00016s (0.00000h)

M_61 = 2305843009213693951
length:      19 digits
found in:    0.00033s (0.00000h)

M_89 = 618970019642690137449562111
length:      27 digits
found in:    0.00055s (0.00000h)

M_107 = 162259276829213363391578010288127
length:      33 digits
found in:    0.00074s (0.00000h)

M_127 = 170141183460469231731687303715884105727
length:      39 digits
found in:    0.00092s (0.00000h)

M_521 = 68647976601 ... 115057151
length:      157 digits
found in:    0.02121s (0.00001h)

M_607 = 53113799281 ... 393219031728127
length:      183 digits
found in:    0.03308s (0.00001h)

M_1279 = 10407932194 ... 55703168729087
length:      386 digits
found in:    0.35498s (0.00010h)

M_2203 = 14759799152 ... 6697771007
length:      664 digits
found in:    2.33282s (0.00065h)

M_2281 = 446087557 ... 418132836351
length:      687 digits
found in:    2.68216s (0.00075h)

M_3217 = 2591170860 ... 60677362909315071
length:      969 digits
found in:    9.29050s (0.00258h)

M_4253 = 1907970075 ... 87815350484991
length:      1281 digits
found in:    26.13799s (0.00726h)

M_4423 = 285542542 ... 2608580607
length:      1332 digits
found in:    29.79591s (0.00828h)

M_9689 = 4782202788 ... 826225754111

```

```

length:          2917 digits
found in:    551.35114s (0.15315h)

M_9941 = 3460882824908 ... 883789463551
length:          2993 digits
found in:    611.11865s (0.16976h)

M_11213 = 28141120 ... 91476087696392191
length:          3376 digits
found in:    921.22742s (0.25590h)

M_19937 = 43154247 ... 39030968041471
length:          6002 digits
found in:    8018.67913s (2.22741h)

M_21701 = 448679166119 ... 8353511882751
length:          6533 digits
found in:    11050.95511s (3.06971h)

M_23209 = 4028741157789 ... 523779264511
length:          6987 digits
found in:    14460.4163s (4.01678h)

```

As we can see, running [Code 2.32](#) on MacBook Pro equipped with Intel i7 2.6GHz CPU and 16GB RAM—the "discovery" of the 26th number took a bit over 4 hours (1 CPU core).

You can verify the results at <http://www.mersenne.org/primes/> webpage that lists all confirmed numbers as well as those still waiting for confirmation. The 26th number has been found in February 1979. At this point, it is worth reflecting upon the power of computing at our disposal — today.

Try to run the code for a week. I wonder how many M_n you can find? E-mail me then. We will come back to the multi-core Python programming in *Volume II*.

2.4.6. Randomness of `random`. Mersenne Twister.

So far we understood that, in general, the output in a form of a random number came out from a dedicated computer PRNG function. To be perfectly honest, it is like a refined gentleman, of a sophisticated quality recognised by its efficiency when a great number of drafts is requested.

In Python 3.5 (2.7.10) the algorithm being responsible for delivery of pseudo-random numbers is known as **Mersenne Twister** (MT) developed by two Japanese researchers Makoto Matsumoto and Takuji Nishimurain in 1997 and has become "the standard" worldwide.

Its basic version uses a 32-bit word generation and is labeled as MT19937. It has been designed with a consideration on the flaws of the various past and currently existing PRNGs. The mathematics standing behind the algorithm itself is beyond the scope of this book. However, as one might suspect, the algorithm

must repeat itself and in case of our Japanese hero, its period is very long, defined by the 24th Mersenne Prime number, i.e.

$$M_{19937} = 2^{19937} - 1 = 431542479...968041471$$

We derived it in the previous Section.

A full story about the research on MT you may find at <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html> — the official website of MT. For the sake of curiosity, below, we will scan across one of the accessible Python implementation of the Mersenne Twister algorithm. Although the code is possible to be found on the Web (e.g. <http://code.activestate.com/recipes/578056-mersenne-twister/>) its functionality may not be perfect.

In this exercise we aim at the inspection of the complexity of the MT algorithm itself and at showing an example of implementation utilising some additional Python elements, for instance, **bitwise operators** (omitted in this *Volume*) or global variables.

Code 2.33 Mersenne Twister in Python.

```
from datetime import datetime

def initialize_generator(seed):
    global MT
    global bitmask_1
    MT[0] = seed
    for i in range(1,624):
        MT[i] = ((1812433253 * MT[i-1]) ^ ((MT[i-1] >> 30) + i))
        & bitmask_1

def extract_number():
    global index
    global MT
    if index == 0:
        generate_numbers()
    y = MT[index]
    y ^= y >> 11
    y ^= (y << 7) & 2636928640
    y ^= (y << 15) & 4022730752
    y ^= y >> 18

    index = (index + 1) % 624
    return y

def generate_numbers():
    global MT
    for i in range(624):
        y = (MT[i] & bitmask_2) + (MT[(i + 1) % 624] &
            bitmask_3)
        MT[i] = MT[(i + 397) % 624] ^ (y >> 1)
        if y % 2 != 0:
            MT[i] ^= 2567483615

# main program
MT = [0 for i in range(624)]
index = 0
bitmask_1 = (2 ** 32) - 1
bitmask_2 = 2 ** 31
```

```

bitmask_3 = (2 ** 31) - 1

now = datetime.now()
initialize_generator(now.microsecond)

# printing 10 random numbers
for i in range(10):
    rnd = extract_number() # an integer!
    print(rnd)

```

A possible output could be:

```

3489063849
884313957
3591573376
4172670803
3535921056
2614888581
229773391
3718065951
1387448252
4277039060

```

As for today, the Mersenne Twister established its firm position among numerous PRNGs as a fast and reliable random number source. It has passed a lot (though not all) of tests for randomness. Jones (2000) points at more reliable and newer algorithms characterised even by much longer periods. Explore them.

Every new random number generator needs to pass a rich set of tests, such as—the NIST Suite (<http://csrc.nist.gov/groups/ST/toolkit/rng>) composed of more than 15 cryptographic tests for randomness, in order to determine whether it is sufficient for the cryptographic use.

The Python version of NIST Suite has been recently provided by my friend and quant fellow Stuart Reid (2015). Earlier the same year, I described a novel approach making use of the Walsh–Hadamard Transform and found a strong evidence of randomness for Mersenne Twister algorithm as implemented in Python 2.7.10 and 3.4 (<http://www.quantatrisk.com/2015/04/07/walsh-hadamard-transform-python-tests-for-randomness-of-financial-return-series/>).

Now, one the easiest ways to obtain a float number (e.g. of a 12+ digit precision and to be in $(0; 1)$ interval) out of the provided output is to modify the last four lines of [Code 2.33](#) in the following way:

```

# printing 10 random numbers (float)
for i in range(10):
    rnd = '0.' + str(extract_number()) + str(extract_number())
    rnd = float(rnd)
    print(rnd)

```

leading to, e.g.:

```

0.38811813362753395
0.6571432443750916

```



```

0.3169495821382487
0.8952493993946118
0.9744576171808691
0.9519982367771294
0.2890105433198552
0.4355670321291
0.4654332166969249
0.7070961503032999

```

Quite handy flexibility of using both strings, integers, and floats—all together. Nothing new. Just a few seconds of knowledge mixed with our imagination.

As an exercise, try to apply K-S Test for the output obtained based on [Code 2.33](#). Can you confirm uniform distribution for those random numbers? Write your own program that merges [Code 2.33](#) as an input and [Code 2.28](#) as a testing framework. What are your findings? You should be surprised. Tell me why. ☺

Seed and Functions for Random Selection

When you plant a seed of an oak tree, eventually it will grow reaching an impressive size. The problem is that you cannot recreate exactly the same tree from the same seed. It does not apply to PRNGs like the built-in Mersenne Twister. If exactly the same stream of random numbers has to be generated, one can employ the `random.seed` function.

In [Code 2.33](#) the seed value is taken from the current value of time read out with a help of, mentioned earlier, `datetime` module. We assumed the **seed** is an integer number corresponding to a microsecond:

```

>>> from datetime import datetime
>>> now = datetime.now()
>>> now
datetime.datetime(2015, 3, 21, 20, 39, 34, 610531)
>>> now.microsecond
610531

```

Such method can be effective but limited by 999999, i.e. the total number of possible combinations. Analyse the following code:

Code 2.34

```

from datetime import datetime
import random as r

x = datetime.now()
x = x.microsecond

random.seed(x)
print("seed = %g" % x)
print("rv   = %.10f" % r.random())

# shuffle the list
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
random.shuffle(lst, r.random)
print(lst)

# random sample from the list

```

random.sample`print(r.sample(lst, 3))`

where one of possible outputs is:

```
seed = 319303
rv    = 0.5248070368
['c', 'e', 'b', 'f', 'd', 'g', 'a']
['a', 'g', 'e']
```

First of all, we specified the seed based on the current value of microsecond. If the function is called as `seed()` or completely omitted, the current system time is used in generating the seed (by default). Therefore, the use of `seed(seed)` makes sense if we want to repeat all "random" calculations with the same seed.

Secondly, for a list `lst` we employed the `shuffle` function which returns a new order of the items in a random manner. The function works in-place. On the other hand, the `sample(lst, 3)` function picks randomly three items from the list. The latter can be easily used in order to build a simple LOTTO simulator:

```
import random

x = range(1, 50)
lucky6 = list(random.sample(x, 6))
lucky6.sort()
print(lucky6)
```

where we pick 6 lucky numbers among the numbers between 1 and 49, e.g:

```
[10, 14, 21, 29, 41, 45]
```

The total number of combinations is 13983816. The same result we can get with the application of the `choice` function. It returns a random element from the non-random sequence:

random.choice

```
import random

x = range(1, 50)
newlucky6 = []

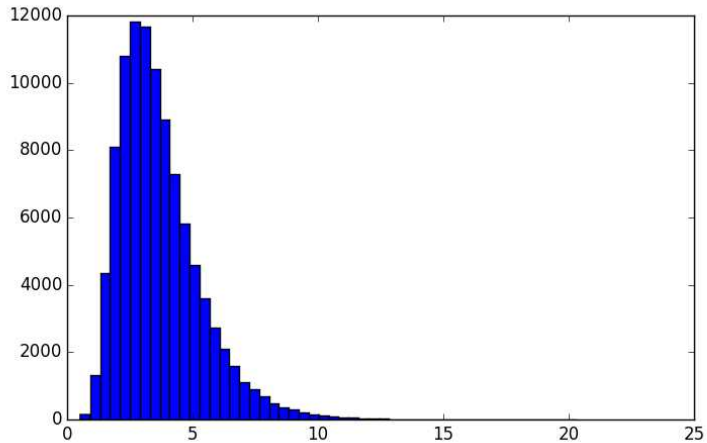
for i in range(1, 7):
    num = random.choice(x)
    while num in newlucky6:
        num = random.choice(x)
    else:
        newlucky6.append(num)

newlucky6.sort()
print(newlucky6)
```

returning, e.g.:

```
[7, 11, 12, 25, 26, 46]
```

Here, we ensure that the numbers picked by the `choice` function are not the same.



Random Variables from Non-Random Distributions

The `random` module comes with some basic ready-to-use functions allowing us to draw random numbers associated with a specific distribution. Say, you need to generate a sample of rvs based on the underlying **lognormal distribution** described by two parameters. Employing the `lognormvariate` function your wish is possible:

Code 2.35

```
import random
from matplotlib import pyplot as plt

mu = 1.21
sigma = 0.43
r = [random.lognormvariate(mu, sigma) for i in range(100000)]

plt.figure(figsize=(8, 5))
plt.hist(r, bins=50)
plt.show()
```

`random.lognormvariate`

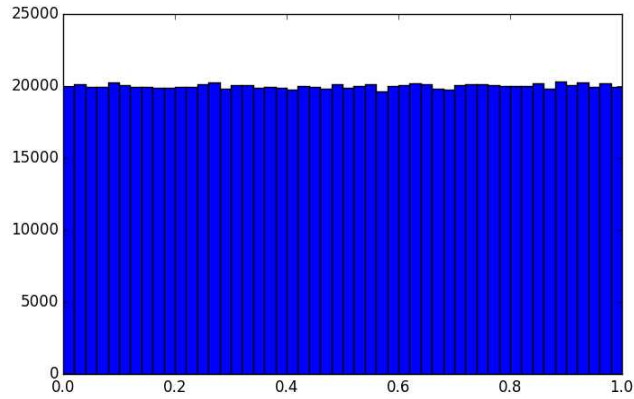
where the empirical histogram of the lognormal rvs we plot for the visual verification (see above).

In summary, the `random` module offers the following functions:

`random. Functions`

- `randrange(stop)`
- `randrange(start, stop, [step])`
- `randint(a, b)`
- `random()`
- `uniform(a,b)`
- `triangular(low, high, mode)`
- `betavariate(alpha, beta)`
- `expovariate(lambda)`
- `gamavariate(alpha, beta)`
- `gauss(mu, sigma)`
- `lognormvariate(mu, sigma)`
- `paretovariate(alpha)`
- `weibullvariate(alpha, beta)`

See *References* for further exploration of the topic. More on random numbers you will learn in Section 3.4.



2.4.7. urandom

Lastly, a short comment on a viable alternative to the Mersenne Twister implemented within the `random` module. A computer may be used as source of randomness. Think for a moment that anything from your keystrokes to the vibration of the cooling fan may be considered as a source of **entropy**. The operating system has a continuously running method of generating random numbers from the kernel space. The generator keeps the estimate of a number of bits of **noise** in the **entropy pool**. In Linux and Mac OS X, that information is stored at the location of `/dev/urandom`. There exists an abundant documentation across the Web arguing on `/dev/urandom` as an attractive source of pseudo-random numbers of the cryptographic quality. I strongly encourage you to explore this field for your own curiosity. Trust me. It's fascinating!

Code 2.36

In Python 3.5, we can use `/dev/urandom` in the following way:

```
from matplotlib import pyplot as plt
import array
import os

# Generates n random floats in the range [0, 1) using
# os.urandom() as source of randomness
def urandom_random(n):
    data = os.urandom(n * 8)
    arr = array.array("Q", data)
    return [float(ulong)/(2**64+1) for ulong in arr.tolist()]

# Generates n random ints in the range [a, b] using os.urandom()
# as source of randomness
def urandom_randint(n, a, b):
    random = urandom_random(n)
    return [int(r * (b-a+1) + a) for r in random]

r = urandom_random(1000000)
rint = urandom_randint(1000000, 5, 19)

plt.figure(figsize=(8, 5))
plt.hist(r, bins=50) # or (rint,
plt.show()
```

what returns a lovely uniform distribution for floats (see above). More on `/dev/urandom` at you will find at: <https://en.wikipedia.org/wiki/dev/random> webpage.

References

Jones, D., 2000, *Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications*, src: <http://www0.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf>

Matsumoto, M., Nishimura, T., *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 1998

Reid, S., 2015, *Random walks down Wall Street, Stochastic Processes in Python*, src: <http://www.turingfinance.com/random-walks-down-wall-street-stochastic-processes-in-python/>

Further Reading

Janke W., 2002, *Pseudo Random Numbers: Generation and Quality Checks*, Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms, Lecture Notes, J. Grotendorst, D. Marx, A. Muramatsu (Eds.), John von Neumann Institute for Computing, Julich, NIC Series, Vol. 10, ISBN 3-00-009057-6, pp. 447-458, src: https://www.physik.uni-leipzig.de/~janke/Paper/nic10_447_2002.pdf

Hardy, S., 2004, *Pseudorandom Number Generation, Entropy Harvesting, and Provable Security in Linux*, src: <http://www.blackhat.com/presentations/bh-europe-04/bh-eu-04-hardy/bh-eu-04-hardy.pdf>

Malone M., 2015, *TIFU by using Math.random()*, src: <https://medium.com/@betable/tifu-by-using-math-random-f1c308c4fd9d#.mt0nz380p>

Rock, A., 2005, *Pseudorandom Number Generators for Cryptographic Applications*, src: <https://www.rocq.inria.fr/secret/Andrea.Roeck/pdfs/dipl.pdf>

2.5. Beyond the Lists

Python offers us with three more built-in data structures: **tuples**, **sets**, and **dictionaries**. Together with lists, we become equipped with an appealingly powerful arsenal for data manipulation. This Section addresses the most practical aspects of the remaining three musketeers.

2.5.1. Protected by Tuples

Tuples are some kind of isolated and immutable creatures, in many instances and behaviour resembling Python's lists. However, they are different. Playing with tuples is like exploring new living organisms that can be immediately distinguished. A trait of the **immutability** for tuples means that once created they cannot be altered *so easily* when it comes to their size and content.

The most elementary form of a tuple would be:

```
>>> t = 1,
>>> t
(1,)
>>> type(t)
<class 'tuple'>
```

The main characteristic of a tuple are **round brackets**, if present, and at least one **comma** after the first element.

possible to be defined in a few alternative ways, e.g.:

```
>>> x = 1
>>> tuple([x]) # assume an input as a direct list
(1,)
```

tuple

Note that the following conversion won't work too:

```
>>> x = 1
>>> tuple(list(x))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

while

```
>>> tuple(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

returns an error.

A bit more sexy and mature look of the tuple would be:

```
>>> t = (3.14, -12, 'call me', [-9, -8, -7], 3-7.1j)
>>> t
(3.14, -12, 'call me', [-9, -8, -7], (3-7.1j))
```

where we stored a collection of numbers, a string, list, and even one complex number. Again, the main difference between the lists and tuples is in their brackets' shape, `[]` vs. `()`, respectively.

Data Processing and Maths of Tuples

A primary reason for using tuples is not only **data grouping** but **protecting** the content of a tuple from outside. It is a kind of an inverse concept of the astronomical black-hole: you can retrieve

something from a tuple but it is impossible to amend what has been already put and placed there. Have a look:

```
>>> t = (3.14, -12, 'call me', [-9, -8, -7], 3-7.1j)

>>> t[0]
3.14
>>> t[1]
-12
>>> t[-2]
[-9, -8, -7]
```

but

```
>>> t[0] = 2.71
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Above, we have seen that an **extraction** of an indexed item for a tuple works the same way as for the lists. **Slicing**? The same story:

```
>>> t[-3:-1]
('call me', [-9, -8, -7])
>>> c = t[-1]
>>> c
((3-7.1j),)
>>> type(c)
<class 'tuple'>
```

and the outputs come out as tuples. Pay attention to the form of **c** variable in this example. The additional round brackets around the complex number may be misleading and it does not mean we deal with a tuple inside the tuple. You may check it quickly that:

```
>>> c[0]
(3-7.1j)
>>> type(c[0])
<class 'complex'>

>>> c[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

Tuples have **built-in tuple functions**, i.e.:

```
>>> t = tuple(list(range(6))); t
(0, 1, 2, 3, 4, 5)
>>> min(t)
0
>>> max(t)
5
>>> len(t) # a number of elements inside the tuple
6
```

With a help of **math**, **statistics** or **numpy** module, you can perform the basic operations on tuples storing numbers, for instance:

```
>>> import math
>>> import statistics as st
>>> import numpy as np

>>> math.fsum(t)
15.0
>>> np.sum(t)
15
```



```
>>> st.mean(t)
2.5
```

however:

```
>>> -t
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for unary -: 'tuple'

>>> x = -1*t
>>> x
()
>>> type(x)
<class 'tuple'>
>>> x[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

where the last operation returns an **empty tuple** without any elements inside. Honestly, a bit bizarre organism that you can, in fact, sometimes meet in the Python's ecosystem.

Methods and Membership

The **arithmetics of tuples** is counterintuitive, pointing at no element-wise operations as we know from algebra:

```
>>> t
(0, 1, 2, 3, 4, 5)
>>> t + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int") to tuple
```

and

```
>>> t + 10,
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int") to tuple
```

but

```
>>> t + (10,)
(0, 1, 2, 3, 4, 5, 10)
>>> t + ('Volvo', 4, 'Life')
(0, 1, 2, 3, 4, 5, 'Volvo', 4, 'Life')
>>> ('Counting',":") + t
('Counting', ':', 0, 1, 2, 3, 4, 5)
```

extends tuple by appending another tuple. Unfortunately, tuples do not have `.extend()`, `.append()`, `.remove()`, nor `.pop()` methods as we have witnessed for the lists. However, the following operations work:

```
>>> t
(0, 1, 2, 3, 4, 5)
>>> t * 2
(0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5)
>>> tuple([t[i] for i in range(2, 6)])
(2, 3, 4, 5)
```

A verification of the **membership** may be tricky, therefore try to memorise what returns an invalid output:

```
>>> t
(0, 1, 2, 3, 4, 5)
>>> (2, 3) in t
False
>>> (2,) in t
False
>>> 2 in t
True
```

Tuple Unpacking

Probably the most useful application of the tuples in Python is so-called **tuple unpacking**. Say, we want to store in a tuple the geographical coordinates of the Changi Airport in Singapore. Next, we wish to unpack them into two separate variables: the first one storing the N (north) and the second one storing E (east) coordinate expressed in degrees, respectively. We achieve it by:

Swapping with tuples:

```
>>> e, n = n, e
>>> e
1.364667
>>> n
103.991563
```

```
>>> changi = 1.364667, 103.991563
>>> n, e = changi
>>> n
1.364667
>>> e
103.991563
```

If for some reason, we are solely interested in N value then the use of an underscore character will suppress E while unpacking:

```
>>> del e # removes e variable, if defined earlier
>>> n, _ = changi
>>> n
1.364667
```

however

```
>>> n = changi
>>> n
(1.364667, 103.991563)
```

will return **n** as a tuple.

The same method is very popular for returning results computed within a custom function, say:

```
def changi2city(a, b, unitprice):
    from math import sqrt
    x, y = a
    x1, y1 = b
    distance = sqrt((x-x1)**2 + (y-y1)**2)
    fuel_consumption = distance * unitprice
    return distance, fuel_consumption # return a tuple
```

```
changi = 1.364667, 103.991563
singapore = 1.292073, 103.861430
```

```
dist, fc = changi2city(changi, singapore, 1.2)
print(dist, fc)
```

```
0.1490116992890162 0.17881403914681943
```

Named Tuples

Lastly, Python's Standard Library offers a brilliant feature that one can utilise as a fancy part of your code—the **named tuples**. They become available if the `namedtuple` function is being imported from the `collections` module.

Named tuples allow us to assign physical names to the tuple's elements and use them as a tuple object with specified labels.

In the following example, we will employ this procedure for describing the **market** (tangent) **portfolio** defined by risk, return, and risk-free asset values, and next, by selecting a portfolio placed randomly in a risk-return plane, we will check its validity assuming an approximated shape of the efficient frontier:

Code 2.37

`collections.namedtuple`

```
from collections import namedtuple

frontier = namedtuple('Frontier', ['risk', 'ret', 'riskfree'])
market = frontier(0.236, 0.21, 0.04)

print(frontier)
print(market.risk)
print(market.ret)

# random portfolio (risk, return)
portfolio = 0.13, 0.4

if(portfolio[0] < market.risk):
    if(portfolio[1] > market.ret):
        print('Invalid portfolio')

<class '__main__.Frontier'>
0.236
0.21
Invalid portfolio
```

Here, by creating a new variable of **market** (a market portfolio placed on the effective frontier line), we assign an object defined by the named tuple of **frontier**. Thanks to that, `market.risk` and `market.ret` can be easily understood while further operations.

2.5.2. Uniqueness of Sets

The title of this Section reveals a widely spread and the most useful application of **sets**—from a group of elements return the gems. Sets are officially declared as the **unordered collections of unique elements** with a limited number of operations one can perform upon them. They differ from lists and tuples in a number of ways, with a basic syntax:

```
{ }, set

>>> s = {1} # alternatively: s = set([1])
>>> s
{1}
>>> type(s)
<class 'set'>
```

where a new function of `set` has been used to convert the list.

The first striking surprise is the behaviour:

```
>>> s = {3, -1j, 6, 'python', 4, 'quants'}
>>> s
{3, 4, 6, 'python', (-0-1j), 'quants'}
```

i.e. after providing the input, the order is rearranged. The second surprise comes when we try to extract any element from a set:

```
>>> s[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>> s[1:3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

revealing that neither **indexing** nor **slicing** for sets do not work.

Therefore, what is all that ado about sets? Well, as it has been said in the beginning, we are able to convert a collection of repeatable elements into a **unique set**, for instance:

```
>>> rvs = [random.randrange(20) for i in range(15)]
>>> rvs
[19, 8, 15, 9, 18, 8, 11, 9, 13, 14, 1, 17, 2, 11, 9]

>>> s = set(rvs)
>>> s
{1, 2, 8, 9, 11, 13, 14, 15, 17, 18, 19}
```

where all duplicates have been removed.

Having two different sets, we can perform certain operations upon them. Analyse the following examples:

```
>>> u1 = [random.randrange(20) for i in range(15)]
>>> u2 = [random.randrange(20) for i in range(15)]
>>> s1 = set(u1)
>>> s2 = set(u2)
>>> s1; s2
{0, 3, 5, 6, 7, 9, 11, 13, 14, 16, 17}
{2, 3, 4, 7, 9, 11, 13, 14, 18}

>>> s1.intersection(s2) # common elements
{3, 7, 9, 11, 13, 14}

>>> s1.union(s2) # all elements of s1 and s2 together
{0, 2, 3, 4, 5, 6, 7, 9, 11, 13, 14, 16, 17, 18}

>>> s1.difference(s2) # s1 - s2, or missing elements of
{0, 16, 5, 6, 17} # s1 in s2

>>> s2.difference(s1) # s2 - s1, or missing elements of
{2, 18, 4} # s2 in s1
```

Updating the set cannot be done by a simple addition, i.e.:

```
>>> {1, 2, 3} + {'a', '7'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

however, with an aid of a special function, namely,

```
>>> s1
{0, 3, 5, 6, 7, 9, 11, 13, 14, 16, 17}
>>> s1.update({99, 100})
{0, 3, 99, 5, 6, 7, 100, 9, 11, 13, 14, 16, 17}
```

and the miracles can happen. What could be pretty useful is:

```
>>> s1
{0, 3, 5, 6, 7, 9, 11, 13, 14, 16, 17}
>>> s3 = {11, 13}
>>> s1.intersection_update(s3)
{11, 13}
```

You may apply this method in order to update **s1** set leaving it solely with those elements which have been common between **s1** and a new set of **s3**.

For more information on other operations available for Python's sets see, e.g.: <https://docs.python.org/2/library/sets.html>.

2.5.3. Dictionaries, i.e. Call Your Broker

The forth attractive form of storing and arranging the data are the Python's **dictionaries**. The elements of the dictionaries enter the barn being labeled by *key* : *value*. Similarly to sets, the dictionaries are surround by curly brackets, and their *keys* are **immutable**.

Let's introduce the fundamentals of dictionaries using an example of brokers' details gathered together in one place (phonebook) where every one of them is described by name and phone number. We start with:

```
>>> brokers = ["Lucky Luc", "Fat Joe", "Filthy Richard"]
>>> phones = [555432032, 555221891, 555988913]

{ }, dict()
>>> d = dict(zip(brokers, phones))
>>> d
{'Fat Joe': 555221891, 'Filthy Richard': 555988913,
 'Lucky Luc': 555432032}
```

where, first, we zip both lists and next we create a dictionary with a help of the **dict** function. The *key* is given by the broker's name while the corresponding *value* by his phone number. Adding a new broker's detail we achieve thanks to:

```
>>> new = {"Johnny Cash" : 555443677}
>>> new
{'Johnny Cash': 555443677}
>>> d.update(new)
>>> d
{'Fat Joe': 555221891, 'Johnny Cash': 555443677,
 'Filthy Richard': 555988913, 'Lucky Luc': 555432032}
```

therefore finding a phone number of *Fat Joe* in your phonebook is as easy as:

```
>>> d["Fat Joe"]
555221891
```

Listing the names of your brokers or, alternatively, only all saved phone numbers in **d** requires:

```
>>> list(d.keys())
['Fat Joe', 'Johnny Cash', 'Filthy Richard', 'Lucky Luc']
>>> list(d.values())
[[555221891, 555443677, 555988913, 555432032]
```

Looking the broker's name up follows:

```
>>> "Fat Joe" in d
True
```

however

```
>>> 555443677 in d
False
>>> 555443677 in d.values()
True
```

Updating the key's value is fairly straightforward, namely:

```
>>> new_phone_number = 555666777
>>> d["Fat Joe"] = new_phone_number
>>> d["Fat Joe"]
555666777
```

The application of Python's dictionaries in **quantitative finance** or **algorithmic trading** is rather limited. However, you may find them practical if you fetch some data online and they are returned in a JSON file format as in the following case of the Google Finance query for the latest update on AAPL stock:

<http://finance.google.com/finance/info?client=ig&q=AAPL:NASDAQ>

The body of the JSON file can be saved as a dictionary:

```
>>> stock = { "id": "22144", "t": "AAPL", "e": "NASDAQ", "l": "117.75", "l_fix": "117.75", "l_cur": "117.75", "s": "1", "lts": "4:00PM EST", "lt": "Nov 23, 4:00PM EST", "lt_dts": "2015-11-23T16:00:01Z", "c": "-1.55", "c_fix": "-1.55", "cp": "-1.30", "cp_fix": "-1.30", "ccol": "chr", "pcls_fix": "119.3", "el": "117.18", "el_fix": "117.18", "el_cur": "117.16", "elt": "Nov 24, 7:31AM EST", "ec": "-0.59", "ec_fix": "-0.59", "ecp": "-0.50", "ecp_fix": "-0.50", "eccol": "chr", "div": "0.52", "yld": "1.77" }
```

which corresponds to the information provided online, i.e.:

Apple Inc. (NASDAQ:AAPL)

117.75 -1.55 (-1.30%)	Range	117.34 - 119.73	Div/yield	0.52/1.77
Pre-market: 117.18 -0.57 (-0.48%)	52 week	92.00 - 134.54	EPS	9.20
Nov 24, 7:31AM EST	Open	119.27	Shares	5.58B
NASDAQ real-time data - Disclaimer	Vol / Avg.	15,256.00/44.37M	Beta	0.86
Currency in USD	Mkt cap	665.08B	Inst. own	59%
	P/E	12.80		

Having that, the last close-price of Apple Inc. and the most current price of the stock in *pre-market* can be extracted by:

```
>>> stock["1"]  
'117.75'  
>>> stock["e1"]  
'117.18'
```

respectively. More on hacking Google Finance for algo traders see: <http://www.quantatrisk.com/2015/05/07/hacking-google-finance-in-pre-market-trading-python/>.

If you find the Python's dictionaries particularly appealing in your research or for the projects, please visit <https://docs.python.org/3.5/library/stdtypes.html#dict> to learn more.

2.6. Functions

One of the coolest thing people love about Python is its flexibility in defining and creating **functions**. Those of you who are more familiarised with other languages like Fortran or C++ remember how much pain it causes to make sure that a function is of a specific type and it returns a right result. There exists a certain trade-off between the final speed of the code execution and the time dedicated to doubly verify that all has been written correctly.

The simplest Python syntax that defines a *custom* function, i.e. the function that you write on your own (or your girlfriend sitting on your laps). I introduced the term *custom* in this book in order to mark a distinction among functions available to us in the standard version of Python or provided by the third-party modules, and the ones you compose from a scratch.

A void function would take a form of:

```
def, return      def donothing():
                  return

                  donothing()
```

that does absolutely nothing while called. We define a function by typing a keyword `def` first, followed by the function's name (best to use lowercase with/out underscore token) and with double round brackets empty inside (if no parameters/arguments are specified or required to be passed on).

A good style of writing a code in Python suggests to keep double blank lines between the function(s) and/or the rest of the main program. This rule can be neglected but it makes all future Python codes easily readable and understandable worldwide. Therefore, learn now how to keep your style at the highest level.

Every function which does not return any output is not asked be ended with a `return` keyword, e.g.

```
def donothing():
    x = 1

    donothing()
```

or

```
def justprint():
    print(" "*20)
    print("Name and Date")

justprint()
```

Name and Date

2.6.1. Functions with a Single Argument

The mathematics imprinted upon us a firm and long-lasting framework of things to be calculated as a function with a single argument, say $y = f(x)$. Thus, when it comes to programming, we expect the same to repeat and appear in the same form. Somehow.

In Python, we use:

```
def domath(x):
    return x**2

print(domath(4))

16
```

We have a freedom of calling and/or passing the argument on:

```
def domath2(x):
    y = x**2      # inner computations
    return y

z = 5 - 1
print(domath2(z))

16
```

and so on. How about that one:

```
def domath3(x=5):
    return x**2

print(domath3(4))
```

Do you think that the function will return 25 or rather 16? You may guess the answer if I say to you that:

```
def domath3(x=5):
    return x**2

print(domath3())

25
```

This method is safe. If you forget to pass any value on to the function, by default, it will assume then that x should be 5.

The complexity of operations inside the function can be of a different magnitude and the result can be found or not. In the following example, we will try to derive a value of a simple mathematical expression where x will be in range between 0 and 4, and division by zero will need to be considered as a potential roadblock preventing the function from ending with success:

Code 2.38

```
def simplemath(x):
    import math
    try:
        c = 1/x + x*(math.exp(1/x))
        return True, c
    except:
        return False, None
```

```

for x in range(5):
    ok, y = simplemath(x)
    if(ok):
        print("f(%g) = %.5f" % (x, y))
    else:
        print("f(%g) = skipped because of x equal 0" % x)

f(0) = skipped because of x equal 0
f(1) = 3.71828
f(2) = 3.79744
f(3) = 4.52017
f(4) = 5.38610

```

Knowing about the problem with an argument equal 0, we protect our calculations with a help of the `try-except` block. For $x > 0$ the function returns a tuple with a boolean variable (*True*) and the derived value of the expression. In other case, *False* and no result is returned. And this is a fantastic way if you think about further data/results processing. Why? Well, if `ok` is *True* (as returned by the function) then we print the results. If not, an alternative output can be considered. For more on good practices in that domain see Slatkin (2015).

2.6.2. Multivariable Functions

There is no much hassle if we want to make our functions working for two or more variables. Analyse first:

```

def f(x, y):
    import cmath
    return cmath.sqrt(-x) * y

print(f(1, 2))
print(f(-1, 2))
print(f(0, 0))
print(f(0, 3))

2j
(2+0j)
0j
0j

```

Again, it may be called as follows and still to deliver the same results:

```

def f(x=10, y=10):
    import cmath
    return cmath.sqrt(-x) * y

print(f(1, 2))
print(f(-1, 2))
print(f(0, 0))
print(f(0, 3))
print(f())

2j
(2+0j)
0j
0j
31.622776601683796j

```

where the calling of the function without any input arguments is possible thanks to the fact that we assigned some default values in the definition of the function itself. Similarly, the

```
def f(x, y=10):
    return x + y

print(f(1, 6))
print(f(1))
print(f(1, y=1))
print(f(y=1, x=4))

7
11
2
5
```

combinations are possible, i.e. the order may be different as long as the assignment is done correctly.

There is a way to call the function with an arbitrary number of arguments. However, that requires an inclusion of the `*` character before the name of a variable, e.g:

```
def func(x=5, *y):
    import math
    print(x)
    print(y)
    out = tuple([x]) + y
    return out, math.fsum(out) # return tuple and a sum

t, s = func(1.1, -3.5, 4.0, 6.9, 3.14)
print(t)
print(s)

t, s = func(1.1)
print(t)
print(s)

1.1                                # the first call
(-3.5, 4.0, 6.9, 3.14)
(1.1, -3.5, 4.0, 6.9, 3.14)
11.64

1.1                                # the second call
()
(1.1,)
1.1
```

Note how much the results differ when additional arguments (`y`) are added while the function's calling.

That's not all. The use of `**` character before the name of an additional argument variable helps us to turn our function into a sophisticated engine at work. Analyse possible outputs for:

```
def func(x=5, *y, **z):
    import math
    print(x)
    print(y)
    print(z)

func(1.1, -3.5, 4.0, 6.9, 3.14, a=3, b=4)
func(1.1, a=3, b=4, c=5)
```

```
1.1
(-3.5, 4.0, 6.9, 3.14)
{'b': 4, 'a': 3}

1.1
()
{'b': 4, 'a': 3, 'c': 5}
```

You may notice that if within the list of arguments, we specify two additional parameters, say `a=3`, `b=4`, then they are stored in a form of a dictionary. A practical aspect of `**` could be:

```
def func(x=5, *y, **z):
    import math
    par = [item[1] for item in z.items()]
    return x + math.fsum(y)*par[0] + par[1]

# a sum should be equal 1 + (2+4)*10 + 20 = 1 + 60 + 20 = 81
y = func(1, 2, 4, a=10, b=20)
print(y)

81
```

Having that said, you gain an extra choice. From now, only your imagination may limit you.

Reference and Further Studies

Beazley, D., Jones, B. K., 2003, *Python Cookbook*, 1st Ed., O'Reilly Media

Slatkin, B., 2015, *Effective Python: 59 Specific Ways to Write Better Python*, Addison-Wesley Professional

Slatkin, B., PyCon 2015, *How to Be More Effective with Functions*, YouTube: <https://www.youtube.com/watch?v=WjJUPxKB164>

van Rossum, G., 2001, *PEP 8—Style Guide for Python Code*, <https://www.python.org/dev/peps/pep-0008/>

3. Fundamentals of NumPy for Quants

3.1. In the Matrix of NumPy

At some stage of your daily work with the applied mathematics, statistics, and data analysis you notice that the majority of operations are, in fact—the **matrix operations**. From a plain vector addition, through square matrix multiplication, to more abstract 5D dataset slicing—all seems to be based on the **matrix algebra**. And the applications are endless: the time-series analysis, data comparison, extraction, filtering, image processing, etc.

Python facilitates programming of any matrix-based operations thanks to **NumPy** (Numerical Python)—a dedicated library for sophisticated mathematical computations. From the very beginning, NumPy has captured the eyes of scientists and researchers and nowadays it establishes one of the most well-known extensions for Python. Its current *NumPy Reference* manual (release 1.10.1; Oct 18, 2015) accessible at <http://docs.scipy.org/doc/> contains over 1500 pages written by the NumPy Community.

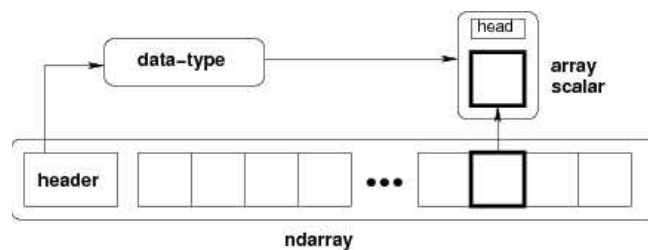
NumPy constitutes a gateway to **pandas**, a second in line the most powerful library designed for data analysis every quantitative analyst, algorithmic trader, or risk investigator should learn. Combined with **SciPy** (a library of efficient numerical routines) and **matplotlib** for plotting, today, NumPy equips you with “more than enough” skill-set to face demanding computational problems and its visualisation.

NumPy becomes alive by typing:

```
import numpy as np
```

which is a recognisable way of referring to the NumPy’s functions worldwide these days. In this book we will be working with NumPy’s release **1.10.1** accessible via Anaconda Python 3.5 distribution or so (see to Section 1.3 for additional information on the installation).

NumPy makes use of the concept of *N*-dimensional array (**ndarray**). According to its definition, the *ndarray* is a (usually fixed-size) multidimensional container of items of the same type and size. The number of dimensions and items in an array is defined by its shape, which is a tuple of *N* positive integers that specify the sizes of each dimension. The type of items in the array is specified by a separate data-type object (**dtype**), one of which is associated with each *ndarray*. As with other container objects in Python, the contents of an *ndarray* can be accessed and modified by indexing or slicing the array (using, for example, *N* integers), and via the methods and attributes of the *ndarray*. Different *ndarrays* can share the same data, so that changes made in one *ndarray* may be visible in another. That is, an *ndarray* can be a “view” to another *ndarray*, and the data it is referring to is taken care of by the “base” *ndarray*. *ndarrays* can also be views to memory owned by Python strings or objects implementing the buffer or array interfaces. Graphically, the holistic picture of *ndarray* can be represented as follows:



Still confused? All right. Let’s make it much simpler now. Imagine, you wish to represent the following matrix of **x** of two rows and three columns, storing the integer numbers:

$$x = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

NumPy allows you for that in the following way:

```
np.array      >>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int32)
               >>> x
               array([1, 2, 3],
                  [4, 5, 6], dtype=int32)
```

Each row must be in square brackets separated by comma(s). The same applies to the outer matrix bracket. **np.array** function creates an *ndarray* object and an additional parameter **np.int32** specifies 32-bit integer type of the object. You can verify that anytime by:

```
Alternatively:      >>> type(x)
>>> x.dtype        <class 'numpy.ndarray'>
dtype('int32')
```

In order to extract **a number of rows and columns** (shape) of your matrix use **.shape** function in one of a few possible ways:

```
.shape           >>> x.shape
                 (2, 3)
```



```

>>> n, m = x.shape # tuple unpacking
>>> n
2
>>> m
3

>>> x.shape[0] # indexing
2
>>> x.shape[1]
3

```

Note on matplotlib for NumPy

Within this Chapter (as we have also done it within the previous one), we will make a few references to **matplotlib**—a special library dedicated for **visualisation** of your numerical results. Since it will be a subject of the *Python for Quants Supplement Series* book of *matplotlib for Quants* (2016), today, I strongly encourage you to study the fundamental plotting functions of matplotlib, concurrently. The best Web sources that I can recommend include:

- The matplotlib Official Page (<http://matplotlib.org>);
- Matplotlib tutorial by Nicolas Rougier (<http://www.labri.fr/perso/nrougier/teaching/matplotlib/>);
- Sebastian Raschka's Gallery (<https://github.com/rasbt/matplotlib-gallery>)

Additionally, the module of **seaborn** for statistical data visualisation should find itself on your list of the graphical libraries for Python to explore (see <http://stanford.edu/~mwaskom/software/seaborn/>).

3.2. 1D Arrays

In the beginning there was an emptiness in NumPy,

```
>>> x = np.array([])
>>> x
array([], dtype=float64)
```

so NumPy developers added some numbers and arranged them in one line as a **row vector**:

```
>>> x = np.array([1,2,3,4])
>>> x
array([1, 2, 3, 4])
>>> x.size
4
>>> x.shape
(4,)
```

.size
It counts the number of all elements in the matrix.

.dtype
>>> x.dtype
dtype('int64')

And it was good. ☺

As a custom in algebra, the term of a *row vector* can be used interchangeably with *one row matrix (array)*, *1D matrix*, or its **transposed** version of a **column vector** that you can obtain by:

.T
>>> x = np.array([1,2,3,4])
>>> x.T
array([1, 2, 3, 4])

but

```
>>> x = np.array([[1,2,3], [5,6,7]])
>>> x.T
array([[1, 5],
       [2, 6],
       [3, 7]])
```

3.2.1. Types of NumPy Arrays

As we have learnt in Section 2.1, Python offers different types of numbers as represented, used, and stored by computer. In NumPy, all arrays of numbers are referred to by their *dtype* as follows:

```
>>> x=np.array([1,2,3,4], dtype=np.int32)
>>> x=np.array([1,2,3,4], dtype=np.int64)      # default
>>> x=np.array([1,2,3,4], dtype=np.float32)
>>> x=np.array([1,2,3,4], dtype=np.float64)    # default
>>> x=np.array([1,2,3,4], dtype=np.float128)
>>> x=np.array([1,2,3,4], dtype=np.complex64)
>>> x=np.array([1,2,3,4], dtype=np.complex128) # default
>>> x=np.array([1,2,3,4], dtype=np.complex256)
```

where `default` denotes a default type, i.e. you can skip its specification as an additional parameter if you aim to have your array's elements to be integer or float or complex in type, for example:

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int32)
>>> x
array([1, 2, 3, 4], dtype=int32)
```

while

```
>>> x = np.array([1, 2, 3, 4])
>>> x
array([1, 2, 3, 4])
>>> x.dtype
dtype('int64')
```

or

```
>>> x = np.array([1.0, 2.6, 3., 4.3]); x
array([ 1. ,  2.6,  3. ,  4.3])
>>> x.dtype
dtype('float64')
```

Conversion of Types

In the abovementioned definitions of **x** arrays, for both `.floatXXX` and complex `.complexXXX` types, we may obtain a **conversion** of the integer-type elements into requested matrix type by, e.g.:

```
>>> x = np.array([1, 2, 3, 4], dtype=np.float128)
>>> x
array([ 1.0,  2.0,  3.0,  4.0], dtype=float128)

>>> x = np.array([1, 2, 3, 4], dtype=np.complex256)
>>> x
array([ 1.0+0.0j,  2.0+0.0j,  3.0+0.0j,  4.0+0.0j],
      dtype=complex256)
```

what can be very useful in many instances of your work with data analysis.

Alternatively, we can convert array's type using a dedicated function:

```
>>> x = np.array([1., -2., 3., 4.])
>>> y = x.astype(np.float32)
>>> y
array([ 1., -2.,  3.,  4.], dtype=float32)
```

what, fortunately, does a transformed and independent copy of **x** (more on creating the copies of arrays, shortly).

Also,

```
>>> x = np.array([1., -2., 3., 4.])
>>> z = x.astype(np.complex64)
>>> z
array([ 1.+0.j, -2.+0.j,  3.+0.j,  4.+0.j], dtype=complex64)
```

Verifying 1D Shape

Checking the shape of matrix **x** may deliver an Index Error. Analyse the following outputs:

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape[0]
4

>>> x.shape[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

Therefore, in order to make sure we deal with a row vector, run a quick **check** utilising `.ndim` or `len` function:

```
.ndim
len

x = np.array([1, 2, 3, 4])
if(x.ndim == 1):           # alternatively if(len(x.shape)==1):
    print('A row vector')

A row vector
```

where `x.shape` returns a tuple (4,) and its number of elements is one.

More on Type Assignment

It is possible to specify an array type only **once** and do it as a string variable. Therefore, every new matrix can have an assigned type, for example:

```
np.dtype

>>> dt = np.dtype('complex256') # or dt=np.dtype(np.complex256)
>>> x = np.array([1,2,3,4], dtype=dt) # or x=np.array([1,2,3,4], dt)
>>> x
array([ 1.0+0.0j,  2.0+0.0j,  3.0+0.0j,  4.0+0.0j],
      dtype=complex256)
```

or alternatively:

```
>>> dt = np.dtype(np.complex256)
>>> x = np.array([1,2,3,4], dt)
>>> x
array([ 1.0+0.0j,  2.0+0.0j,  3.0+0.0j,  4.0+0.0j],
      dtype=complex256)
```

3.2.2. Indexing and Slicing

An access to 1D array's elements is pretty much the same as for the standard Python's list. Analyse the following examples. You should notice a striking resemblance:

```
>>> x = np.array([1.0, 2.6, 3., 4.3, 7.7])

>>> y = x[0]
>>> z = x[-1]
>>> print(y+z)           # add the first and last element of matrix x
8.7

>>> x[1:]                # elements starting from index=1 till the end
array([ 2.6, 3., 4.3, 7.7])

>>> x[-2:]               # last two elements of x
array([ 4.3, 7.7])
```

```
>>> x[0:4]      # elements indexed from 0 to 3 (without 4th)
array([ 1., 2.6, 3., 4.3])

>>> x[1:2]      # 2nd element of x
array([ 2.6])

>>> x[1:2] == x[1] # checking a logical condition of equality
array([ True], dtype=bool)
```

and so on.

Basic Use of Boolean Arrays

Interestingly, the last command opens up an access to a few useful ways of thinking in Python with NumPy. Firstly, we notice a new type of an array to be **boolean** with *True* or *False* elements. We can use them, as follows:

```
>>> x = np.array([1.0, 3. , 2.6, 4.3, 7.7, 5.6])
>>> y = np.array([1. , 2.6, 3., 4. , 7. , 5.6])
>>> cmp = (x == y) # test for equality
>>> cmp
array([ True, False, False, False, False, True], dtype=bool)
```

what returns an array with the results of an index-wise **comparison**.

If we are interested in verifying whether all elements are the same, then, instead of writing:

```
.sum()      if(cmp.sum() == len(x)): # sums all elements in cmp; True=1, False=0
            print('The same!')
            else:
                print('Different')
```

we can use a better method, namely:

Code 3.1

```
import numpy as np

x = np.array([1.0, 3. , 2.6, 4.3, 7.7, 5.6])
y = np.array([1. , 2.6, 3., 4. , 7. , 5.6])
cmp = (x == y) # test for equality
print(cmp)

i = 0
.all()      if(cmp.all()):
            print("All elements are the same")
.any()      elif(cmp.any()):
            print("Some elements are the same, i.e.:")
            # Test1:
            # index-wise
            for e in cmp:
                if(e): # tests if e==True (a useful trick!)
                    print("%.1f" % x[i])
                    i += 1
            # Test2:
            # element-wise
            print('---')
            for e in x:
np.where    j = np.where(e == y) # a tuple containing arrays with
                                     indexes

            print(j)
            if(j[0].size != 0):
                print(y[j])
            else:
                print("All elements are different")
```

what results in the following output:

```
[ True False False False False  True]
Some elements are the same, i.e.:
1.0
5.6
---
(array([0]),)
[ 1.]
(array([2]),)
[ 3.]
(array([1]),)
[ 2.6]
(array([], dtype=int64),)
(array([], dtype=int64),)
(array([5]),)
[ 5.6]
```

and brings us to a few new great NumPy features. Let's analyse them step by step now.

The *Test1* delivered, as expected, values of 1.0 and 5.6 based on `cmp` test of equality (index-by-index). However, as noticed, both arrays, **x** and **y**, contain some identical elements. Therefore, a new *Test2* scans all elements in **x** and checks the corresponding indexes in **y** where the values of its elements are the same. We employ `where` function that returns a tuple with its elements returned as arrays listing indexes in **y**. Since we work with 1D matrixes, each tuple is of 1 or 0 (empty) in its number of elements and we verify it using `.size` function applied to the tuple's first element.

It may sound complicated but with a bit of practice, it will become your second nature as it comes to that sort of simple technical operations on NumPy arrays.

Code 3.1 fails if `cmp` returns all *Falses* even if some elements had been the same. As homework, try to modify it to take this issue into account.

3.2.3. Fundamentals of NaNs and Zeros

It is often a case in the time-series or image processing that our matrixes have some gaps or missing numbers. By convention, we mark them as **NaNs** (Not a Number).

In case of 1D arrays, we can create any row vector with nans using a ready-to-use function of `np.nan`, as in the following example:

```
np.nan >>> x = np.array([1, 2, np.nan, 3, 4, np.nan, 5])
>>> x
array([ 1.,  2., nan,  3.,  4., nan,  5.])
```

If we want to filter out all non-nan values, we type:

```
np.isnan >>> y = x[~np.isnan(x)]
>>> y
```

```
array([ 1.,  2.,  3.,  4.,  5.])
```

Finding the sum of vector **x** follows:

```
>>> s = x[~np.isnan(x)].sum()
>>> s
15.0
```

Now let's add some **zeros** and see what can we do more with such extended vector:

```
>>> x = np.array([0, 1, 2, 0, np.nan, 3, 0, 4, np.nan, 5])
>>> x
array([ 0.,  1.,  2.,  0., nan,  3.,  0.,  4., nan,  5.])
```

A function `.nonzero()` acting on any matrix returns an array of indexes corresponding to non-zero values. Analyse:

```
>>> x = np.array([0, 1, 2, 0, np.nan, 3, 0, 4, np.nan, 5])
>>> x
array([ 0.,  1.,  2.,  0., nan,  3.,  0.,  4., nan,  5.])
>>> i = x.nonzero()
>>> x[i]
array([ 1.,  2., nan,  3.,  4., nan,  5.])
```

Good. Having that, a new version of **x** with zeros and nans removed, we may obtain in two steps:

```
>>> x = np.array([0, 1, 2, 0, np.nan, 3, 0, 4, np.nan, 5])
>>> x = x[x.nonzero()]
>>> x = x[~np.isnan(x)]
>>> x
array([ 1.,  2.,  3.,  4.,  5.])
```

3.2.4. Independent Copy of NumPy Array

For those of you who are familiar with programming, it is nearly intuitive that when you create a matrix-variable, say **a**, and you wish to create its (modified) copy, say $b = a + 1$, **b** in fact will be independent of **a**. This is not a case within **older versions** of NumPy due to the "physical" pointing at the same object in the memory. Analyse the following case study:

```
>>> a = np.array([1,2,3,4,5])
>>> b = a + 1
>>> a; b
array([1, 2, 3, 4, 5])
array([2, 3, 4, 5, 6])
```

but now, if:

```
>>> b[0] = 7
>>> a; b
array([7, 2, 3, 4, 5])
array([7, 3, 4, 5, 6])
```

we affect both 0-th elements in **a** and **b** arrays. In order to "break the link" between them, you should create a new copy of a matrix using a `.copy` function:

.copy()

```
>>> a = np.array([1,2,3,4,5])
>>> b = a.copy()
>>> b = a + 1
>>> b[0] = 7
>>> a; b
array([1, 2, 3, 4, 5])
array([7, 3, 4, 5, 6])
```

Fortunately, in Python 3.5 with NumPy 1.10.1+ that problem ceases to exist:

```
>>> import numpy as np
>>> np.__version__
'1.10.1'
>>> a = np.array([1,2,3,4,5])
>>> b = a + 1
>>> a; b
array([1, 2, 3, 4, 5])
array([2, 3, 4, 5, 6])
>>> b[0] = 7
>>> a; b
array([1, 2, 3, 4, 5])
array([7, 3, 4, 5, 6])
```

however, keep that pitfall in mind and check for potential errors within your future projects. Just in case. ☺

3.2.5. 1D Array Flattening and Clipping

For any already existing row vector you can substitute its elements with a desired value. Have a look:

.fill

```
>>> a = np.array([1,2,3,4,5])
>>> a.fill(0);
>>> a
array([0, 0, 0, 0, 0])
```

or

.flat

```
>>> a = np.array([1,2,3,4,5])
>>> a.flat = -1
>>> a
array([-1, -1, -1, -1, -1])
```

It is so-called **flattening**. On the other side, **clipping** in its simplistic form looks like:

np.where

Returns an array with the indexes corresponding to a specified condition (see Section 3.3.3 and 3.8)

```
>>> x = np.array([1., -2., 3., 4., -5.])
>>> i = np.where(x < 0)
>>> x.flat[i] = 0
>>> x
array([ 1.,  0.,  3.,  4.,  0.])
```

Let's consider an example. Working daily with **financial time-series**, sometimes we wish to separate, e.g. a daily return-series into two sub-series storing negative and positive returns, respectively. To do that, in NumPy we can perform the following logic by employing a **.clip** function.

Say, the vector **r** holds daily returns of a stock. Then:

```
>>> r = np.array([0.09,-0.03,-0.04,0.07,0.00,-0.02])

.clip
>>> rneg = r.clip(-1, 0)
>>> rneg
array([ 0. , -0.03, -0.04,  0. ,  0. , -0.02])
>>> rneg = rneg[rneg.nonzero()]
>>> rneg
array([-0.03, -0.04, -0.02])
```

Here, we end up with **rneg** array storing all negative daily returns.

The `.clip(-1, 0)` function should be understood as: clip all values less than -1 to -1 and greater than 0 to 0. It makes sense in our case as we set a lower boundary of -1 (-100.00% daily loss) on one side and 0.00% on the other side. Since zero is usually considered as a “positive” return therefore the application of the `.nonzero` function removes zeros from the **rneg** array.

The situation becomes a bit steeper in case of positive returns. We cannot simply type `rneg=r.clip(0, 1)`. Why? It will replace all negative returns with zeros. Also, if **r** contains daily returns equal 0.00, extra zeros from clipping would introduce an undesired input. We solve this problem by replacing “true” 0.00% stock returns with an abstract number of, say, 9 i.e. 900% daily gain, and proceed further as follows:

```
>>> r2 = r.copy(); r2
array([ 0.09, -0.03, -0.04,  0.07,  0. , -0.02])
>>> i = np.where(r2==0.); r2[i] = 9 # alternatively r2[r2==0.] = 9
>>> rpos = r2.clip(0, 9)
>>> rpos
array([ 0.09,  0. ,  0. ,  0.07,  9. ,  0. ])
>>> rpos = rpos[rpos.nonzero()]
>>> rpos
array([ 0.09,  0.07,  9. ])
>>> rpos[rpos == 9.] = 0.
>>> rpos
array([ 0.09,  0.07,  0. ])
```

If you think for a while, you will discover that in fact all the effort can be shortened down to two essential lines of code providing us with the same results:

```
>>> r = np.array([0.09,-0.03,-0.04,0.07,0.00,-0.02])
>>> rneg = r[r < 0] # masking
>>> rpos = r[r >= 0] # masking
>>> rneg; rpos
array([-0.03, -0.04, -0.02])
array([ 0.09,  0.07,  0. ])
```

however by doing so you’d miss a lesson on the `.clip` function ☺. More on **masking** for arrays in Section 3.8.

As you can see, Python offers more than one method to solve the same problem. Gaining a flexibility in knowing majority of them will make you a good programmer over time.

By separating two return-series we gain a possibility of conducting an additional research on, for instance, the distribution of extreme losses

or extreme gains for a specific stock in a given time period the data come from. In the abovementioned example our return-series is too short for a complete demonstration, however in general, if we want to extract from each series two most extreme losses and two highest gains, then:

.sort()
By default this function sorts all elements of 1D array in an ascending order and alters the matrix itself (in-place).

```
>>> rneg.sort()
>>> rpos.sort()
>>> el = rneg[0:2]
>>> el
array([-0.04, -0.03])
>>> hg = rpos[-2:]
>>> hg
array([ 0.07,  0.09])
```

If repeated for, say, 500 stocks (daily return time-series) traded within **S&P 500 index**, the same method would lead us to an insight on an empirical distribution of extreme values both negative and positive that could be fitted with a Gumbel distribution and tested against GEV theory.

3.2.6. 1D Special Arrays

NumPy delivers an easy solution in a form of **special arrays** filled with: zeros, ones, or being “empty”. Suddenly, you stop worrying about creating an array of specified dimensions and flattening it. Therefore, in our arsenal we have:

np.zeros
np.empty
np.ones

```
>>> x = np.zeros(5); x
array([ 0.,  0.,  0.,  0.,  0.])
>>> y = np.empty(5); y
array([ 0.,  0.,  0.,  0.,  0.])
>>> z = np.ones(5); z
array([ 1.,  1.,  1.,  1.,  1.])
```

where in fact

```
>>> np.zeros(5) == np.empty(5)
array([ True,  True,  True,  True,  True], dtype=bool)
```

The alternative way to derive the same results would be with an aid of the **.repeat** function acting on a 1-element array:

```
>>> x = np.array([0])
>>> z = np.array([1.])

>>> x = x.repeat(5); x
array([0, 0, 0, 0, 0]) # dtype('int64')

>>> z = z.repeat(5); z
array([ 1.,  1.,  1.,  1.,  1.]) # dtype('float64')
```

or with a help of **.full** function:

FutureWarning: in the future, `full((1, 5), 0)` will return an array of dtype('int64')

```
>>> x1 = np.full((1, 5), 0)
>>> x1
array([[ 0.,  0.,  0.,  0.,  0.]])

>>> x2 = np.full((1, 5), 1, dtype=np.int64)
```

```
>>> x2
array([[1, 1, 1, 1, 1]])
```

where the shapes of the arrays, **x1** and **x2**, have been provided within the inner round brackets: 1 row and 5 columns.

An additional special array containing numbers from 0 to $N-1$ we create using the **arange** function. Analyse the following cases:

np.arange

```
>>> a = np.arange(11)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

>>> a = np.arange(10) + 1
>>> a
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

>>> a = np.arange(0, 11, 2)
>>> a
array([ 0,  2,  4,  6,  8, 10])

>>> a = np.arange(0, 10, 2) + 1
>>> a
array([1, 3, 5, 7, 9])
```

and also

```
>>> b = np.arange(5, dtype=np.float32)
>>> b
array([ 0.,  1.,  2.,  3.,  4.], dtype=float32)
```

Array-List-Array

The **conversion** of 1D array into Python's list one can achieve by the application of the **.tolist()** function:

```
>>> r = np.array([0.09,-0.03,-0.04,0.07,0.00,-0.02])
>>> l = r.tolist()
>>> l
[0.09, -0.03, -0.04, 0.07, 0.0, -0.02]
```

On the other hand, to go from a flat Python list to NumPy 1D array employ the **asarray** function:

```
>>> type(l)
<class 'list'>

>>> a = np.asarray(l)
>>> a
array([ 0.09, -0.03, -0.04,  0.07,  0.  , -0.02])

>>> a.dtype
dtype('float64')
```

3.2.7. Handling Infs

Quite often you may encounter a problem of the **infinite numbers** in your arrays. They appear denoted by **inf**. Below, we will see how

this may occur and what is the best way to get rid of infs from any array.

A typical error can be overlooked if you compute:

```
>>> a = np.array([1., 2., 3., 4., 5.])
>>> b = np.array([1., 2., 3., 0., 5.])
>>> c = a/b
__main__:1: RuntimeWarning: divide by zero encountered in
      true_divide
>>> c
[ 1.,  1.,  1.,  inf,  1.]
```

From this point we can allow our program (a) to inform us about existing infs in our vector or (b) remove them. For the latter, we write a function preceding the main code:

Code 3.2

```
import numpy as np

def removeinf(x):
    i = np.where(x != np.inf)
    return x[i]

a = np.array([1.,2.,3.,4.,5.])
b = np.array([1.,2.,3.,0.,5.])
c = a/b
print(c)

try:
    np.asarray_chkfinite(c)
except ValueError:
    c = removeinf(c)

print(c)
```

np.inf

np.asarray_chkfinite

what returns:

```
[ 1.  1.  1.  inf  1.]
RuntimeWarning: divide by zero encountered in divide
c=a/b
[ 1.  1.  1.  1.]
```

usually supplemented with a warning that Python encountered a division by zero. A special function of `asarray_chkfinite(c)`, coming from the `asarray` family, is expected to return `ValueError` if `inf` has/have been found in the array of `c`.

3.2.8. Linear and Logarithmic Slicing

Our mind operates in a linear scale. Not too many of us even realise that our eyes react to the amount of light in a logarithmic manner. Getting used to the log scale is a challenge itself. Portioning the numbers into equidistant fragments may be helpful once we work with the histograms or empirical distributions.

As we have seen above, the use of the `asarray` function returns the numbers based on *step*, if provided. In case when the boundaries are known but a desired number of **equidistant points** is given instead, we derive:

```
np.linspace    >>> x = np.linspace(0, 10, num=5)
               >>> x
               array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

however

```
>>> x = np.linspace(0, 10, num=5, endpoint=False)
>>> x
array([ 0.,  2.,  4.,  6.,  8.] )
```

what is an equivalent to:

```
>>> x = np.linspace(0, 8, num=5)
>>> x
array([ 0.,  2.,  4.,  6.,  8.] )
```

and the number of bins, num, has been set to 5.

In case of the **logarithmic scales**, we use a function of `logspace` with similar input parameters:

```
np.logspace    >>> x = np.logspace(1, 4, num=4, base=10)
               >>> x
               array([ 10.,  100., 1000., 10000.] )
```

where not only we control the number of bins but also the range for exponents the base is raised to.

3.2.9. Quasi-Cloning of Arrays

NumPy also allows for a convenient way to create a new array based on the other array's **properties**. It is sort of copying the object's initial setup but filling an array with the arbitrary numbers.

Analyse the following lines:

```
>>> x = np.array([.2,-4.5,3.43,9.8,-0.02,1.7], np.float128)
>>> x
array([ 0.2, -4.5,  3.43,  9.8, -0.02,  1.7], dtype=float128)

np.empty_like  >>> y = np.empty_like(x)
               >>> y
               array([ 0.0,  0.0,  0.0,  0.0,  0.0,  0.0], dtype=float128)

np.zeros_like  >>> y = np.zeros_like(x)
               >>> y
               array([ 0.0,  0.0,  0.0,  0.0,  0.0,  0.0], dtype=float128)

np.ones_like   >>> y = np.ones_like(x)
               >>> y
               array([ 1.0,  1.0,  1.0,  1.0,  1.0,  1.0], dtype=float128)
```

By employing this method, we make sure that the resultant array **keeps** the same type and has been **pre-allocated** in RAM.

Some GPU solutions utilising CUDA in Python make use of this approach (e.g. `numbapro` which offers developers the ability to target multicore and GPU architectures with Python code for both ufuncs and general-purpose code; see <http://docs.continuum.io/numbapro/index>).

3.3. 2D Arrays

3.3.1. Making 2D Arrays Alive

Within the previous two Sections we have seen the explicit method of creating 2D array in NumPy:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int32)
>>> x
array([[1, 2, 3],
       [4, 5, 6]], dtype=int32)
```

and

```
>>> y = x.T
>>> y
array([[1, 4],
       [2, 5],
       [3, 6]], dtype=int32)
```

Looks innocent? Just recall our knowledge on copying of arrays. You may find its application in case of 2D arrays very useful. How? Have a look. First we change a single value in the array by specifying the index (cell address):

```
>>> y[2, 1] = 10 # [row, column]
>>> y
array([[ 1,  4],
       [ 2,  5],
       [ 3, 10]], dtype=int32)
```

however:

```
>>> x
array([[ 1,  2,  3],
       [ 4,  5, 10]], dtype=int32)
```

has been changed too. That is why I strongly encourage you to keep the copies of working arrays intact with a help of the `.copy` function (see Section 3.2.4).

We have also learnt that 2D array could be created by **conversion** of 2D Python's **list** into NumPy object, e.g.:

```
>>> x = np.array([range(i-1, i + 4) for i in [1, 2, 3, 4]])
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
>>> x.shape
(4, 5)
```

and the extraction of any row or column we achieve by, e.g.:

```
>>> x[2, :]
array([2, 3, 4, 5, 6])
>>> x[:, 4]
array([4, 5, 6, 7])
```

In addition, the **special arrays** come into the light nearly intuitively.

Let's start with the `np.empty` function:

```
>>> np.empty((5, 2))
array([[ 4.94065646e-324,  9.88131292e-324],
       [ 1.48219694e-323,  1.97626258e-323],
       [ 2.47032823e-323,  2.96439388e-323],
       [ 3.45845952e-323,  3.95252517e-323],
       [ 4.44659081e-323,  4.94065646e-323]])
```

next,

```
>>> np.zeros((2, 2))
array([[ 0.,  0.],
       [ 0.,  0.]])
```

np.eye
Returns a 2-D array with ones on the diagonal and zeros elsewhere.

```
>>> np.eye(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

where a special function of:

np.identity

```
>>> np.identity(4) == np.eye(4)
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
```

is an equivalent of the previous one (element-wise).

If for some reason you need to add three square matrices 3x3, the first one filled fully with value of 3, the second one with 2, and the forth one with zeros, apply:

np.full
np.zeros

```
>>> np.full((3, 3), 3) + np.full((3, 3), 2) + np.zeros((3, 3))
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

The operation of addition is element-wise.

How would you get those two arrays:

```
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2]])

array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

It is accessible through the `fromfunction` function in connection with the inner inclusion of the `lambda` function in the following way:

np.fromfunction

```
>>> np.fromfunction(lambda i, j: j, (3, 3), dtype=int)
>>> np.fromfunction(lambda i, j: i, (3, 3), dtype=int)
```

In a need of creating an empty 2D array with the specified **diagonal elements**, try:


```
np.diagflat    >>> x = np.diagflat([[1, -2, 3, 4]]); x
               array([[ 1,  0,  0,  0],
                       [ 0, -2,  0,  0],
                       [ 0,  0,  3,  0],
                       [ 0,  0,  0,  4]])
```

where the operation of fetching diagonal elements from the 2D array requires:

```
np.diag        >>> d = np.diag(x)
               >>> d
               array([ 1, -2,  3,  4])
```

and the **trace** of **x** (a sum over diagonal elements) we get:

```
np.trace       >>> trace = np.trace(x)
               >>> trace
               6
```

Simple as that.

3.3.2. Dependent and Independent Sub-Arrays

Given any $N \times M$ array, one may look at it from different points of view. Say, the array is 4096 x 4096 however, we would like to work with a selected $K \times L$ fragment of it. In this case, there are two options. The first one is: extract a sub-array and work over it independently aside. The second option is: do the same, i.e. extract it but while working with the sub-array, make sure that all conducted operations are reflected in the original, larger array.

Let's consider the **Vandermonde matrix** for which the columns of the output matrix are powers of the input vector:

```
np.vander      >>> a = np.array([1, 2, 3, 4, 5])
               >>> n = 5
               >>> v = np.vander(a, n); v
               array([[ 1,  1,  1,  1,  1],
                      [16,  8,  4,  2,  1],
                      [81, 27,  9,  3,  1],
                      [256, 64, 16,  4,  1],
                      [625, 125, 25,  5,  1]])
```

The array is 5 x 5. Our goal is (1) to extract two fragments,

```
>>> x = v[0:2, 3:5]; x
array([[1, 1],
       [2, 1]])

>>> y = v[3:5, 0:2]; y
array([[256, 64],
       [625, 125]])
```

next (2) to transpose **y**:

```
>>> y = y.T; y
array([[256, 625],
       [ 64, 125]])
```

Now, those operations:

```
>>> v
array([[ 1,  1,  1,  1,  1],
       [16,  8,  4,  2,  1],
       [81, 27,  9,  3,  1],
       [256, 64, 16,  4,  1],
       [625, 125, 25,  5,  1]])
```

does not affect the original Vandermonde matrix. Therefore, both sub-array **x** with **y** are not linked to **v**. However, by executing what follows, e.g.:

```
>>> v[0:2, 3:5] = y
>>> v[3:5, 0:2] = x
```

surprisingly we get:

```
>>> v
array([[ 1,  1,  1, 256, 625],
       [16,  8,  4,  64, 125],
       [81, 27,  9,  3,  1],
       [256, 625, 16,  4,  1],
       [ 64, 125, 25,  5,  1]])
```

what is not what we expected. Somehow, the sub-array of **x** has lost its corrected values. The error may remain undetected for a longer while as it goes against our logical and mathematics intuition. That is why, it is so important to apply `.copy` function as often as possible. In our code, first add it while typing:

```
>>> x = v[0:2, 3:5].copy() # an independent copy
>>> x
array([[1, 1],
       [2, 1]])

>>> y = v[3:5, 0:2].copy() # an independent copy
>>> y
array([[256,  64],
       [625, 125]])
```

to ensure that in the end:

```
>>> v
array([[ 1,  1,  1, 256, 625],
       [16,  8,  4,  64, 125],
       [81, 27,  9,  3,  1],
       [ 1,  1, 16,  4,  1],
       [ 2,  1, 25,  5,  1]])
```

This tricky example reveals how cautious you should be when it comes to trivial operations on NumPy arrays. In the process of learning, use **interactive shell** of Python to verify some steps that you are taking. Otherwise, the errors can become a source of the undesired stress and anxiety.

3.3.3. Conditional Scanning

The NumPy's *N*-dimensional array can be scanned for specified logical sentence (conditions) with a help of the `np.where` function. It

offers us with two basic scenarios: (a) searching and indexing, (b) array filtering. Analyse the following:

```
>>> np.random.seed(1)
>>> x = np.random.randint(100, size=(3, 3)); x
array([[37, 12, 72],
       [ 9, 75,  5],
       [79, 64, 16]])
```

Here we employ a function returning a 3 x 3 array of random integers (more on randomness in NumPy in Section 3.4) which we will assume as an input.

We would like to find in **x** all values that are greater equal 5 and less than 20. The first method is:

```
np.where >>> r, c = np.where((x >= 5) & (x < 20))
>>> r # row index
array([0, 1, 1, 2])
>>> c # column index
array([1, 0, 2, 2])
```

Having that and:

```
>>> values = x[r, c] # values.sort() next, if required
array([12,  9,  5, 16])
```

we extract from **x** all possible value meeting our searching criteria. A list storing index coordinates can be created as a list of tuples:

```
>>> i = list(zip(r,c))
>>> i
[(0, 1), (1, 0), (1, 2), (2, 2)]
```

therefore

```
>>>[x[j] for j in i]
[12, 9, 5, 16]
```

However, if we wish to display **x** putting all array's values not meeting the requested condition equal 0, then:

In defining logical conditions we make use of **bitwise logic operators** (&, |, ^, ~).

More practical examples on searching and NumPy array screening you will find in **Section 3.8**.

```
>>> a = np.where((x >= 5) & (x < 20), x, 0)
array([[ 0, 12,  0],
       [ 9,  0,  5],
       [ 0,  0, 16]])
```

unless our goal is achieve an opposite effect, then:

```
>>> b = np.where(~((x >= 5) & (x < 20)), x, 0)
array([[37,  0, 72],
       [ 0, 75,  0],
       [79, 64,  0]])
```

Both method can be thought of as a quick array **filtering** or **clipping** as we have seen in the previous Section. It is important to note that **a** and **b** are independent of **x**:

```
>>> x[0, 0] = -100
>>> x
array([[ -100, 12, 72],
       [  9, 75,  5],
       [ 79, 64, 16]])
```

```
>>> b
array([[37,  0, 72],
       [ 0, 75,  0],
       [79, 64,  0]])
```

i.e. the initial value 37 in **b** is not effected by any change applied in **x**.

3.3.4. Basic Engineering of Array Manipulation

NumPy delivers a few useful functions for the array manipulation and creation through the **basic engineering**: reshaping, concatenating, and splitting. As we will witness in a moment, it enriches our experience in understanding how certain solutions have been coded in NumPy. Let's start with some working matrixes:

```
>>> np.random.seed(2)
>>> x = np.random.rand(3, 3); x
array([[ 0.4359949 ,  0.02592623,  0.54966248],
       [ 0.43532239,  0.4203678 ,  0.33033482],
       [ 0.20464863,  0.61927097,  0.29965467]])
>>> y = x.copy() + 1; y
array([[ 1.4359949 ,  1.02592623,  1.54966248],
       [ 1.43532239,  1.4203678 ,  1.33033482],
       [ 1.20464863,  1.61927097,  1.29965467]])
```

and

```
>>> z = y[2, :].copy(); z
array([ 1.20464863,  1.61927097,  1.29965467])
```

A fast method to transform a row vector, **z**, into $N \times 3$ matrix for, say, $N = 4$, is the application of the **np.vstack** function:

```
np.vstack >>> np.vstack([z, z, z, z])
array([[ 1.20464863,  1.61927097,  1.29965467],
       [ 1.20464863,  1.61927097,  1.29965467],
       [ 1.20464863,  1.61927097,  1.29965467],
       [ 1.20464863,  1.61927097,  1.29965467]])
```

or

```
Code 3.3 import numpy as np

np.random.seed(2)
x = np.random.rand(3, 3)
y = x.copy() + 1
z = y[2, :].copy()

for i in range(5):
    if i > 1:
        tmp = np.vstack([tmp, z])
    else:
        tmp = z.copy()

print(tmp)
```

what allows for a **vertical** duplication of **z**. Yes, the **horizontal** expansion is also possible. This time with a help of the **np.hstack** function acting on the column vector. The transposition of the original **z** you may obtain in two ways:

```
>>> z1 = z.T
array([[ 1.20464863],
       [ 1.61927097],
       [ 1.29965467]])
```

or

np.reshape
a row vector into a column
vector.

```
>>> z1 = np.reshape(z, (3, 1))
```

if you specify the dimensions as an input in the **np.reshape** function.
Therefore,

np.hstack

```
>>> np.hstack([z, z, z, z])
array([[ 1.20464863,  1.20464863,  1.20464863,  1.20464863],
       [ 1.61927097,  1.61927097,  1.61927097,  1.61927097],
       [ 1.29965467,  1.29965467,  1.29965467,  1.29965467]])
```

Reshaping, in its most popular application, is mainly used for the transformation of 1D arrays into 2D matrixes. The dimensions have to be selected in the way to allow for the transformation itself:

```
>>> x
array([[ 0.4359949 ,  0.02592623,  0.54966248],
       [ 0.43532239,  0.4203678 ,  0.33033482],
       [ 0.20464863,  0.61927097,  0.29965467]])

>>> xflat = np.reshape(x, x.size)
>>> xflat
array([ 0.4359949 ,  0.02592623,  0.54966248,  0.43532239,
        0.4203678 ,  0.33033482,  0.20464863,  0.61927097,
        0.29965467])
```

Here, **np.reshape** **flattens** 3x3 NumPy array of **x**. Something went wrong? No problem. **Rebuilding** the 3x3 array from a flat row vector takes a second:

```
>>> np.reshape(xflat, (3, 3))
array([[ 0.4359949 ,  0.02592623,  0.54966248],
       [ 0.43532239,  0.4203678 ,  0.33033482],
       [ 0.20464863,  0.61927097,  0.29965467]])
```

In general, for any row vector, say **v**, of the total number of elements **v.size** please check:

```
(np.sqrt(v.size) - np.sqrt(v.size) // 1) == 0)
```

in order to ensure that you can apply **np.reshape** without an error, e.g.:

```
>>> xflat
array([ 0.4359949 ,  0.02592623,  0.54966248,  0.43532239,
        0.4203678 ,  0.33033482,  0.20464863,  0.61927097,
        0.29965467])
>>> xs = xflat.size; xs
9
>>> xd = int(np.sqrt(xs)); xd
3
>>> xs = xflat.size; xs
9
>>> test = (np.sqrt(xs) - np.sqrt(xs) // 1) == 0; test
True
>>> if(test):
...     np.reshape(xflat, (xd, xd))
... 
```

```
array([[ 0.4359949 ,  0.02592623,  0.54966248],
       [ 0.43532239,  0.4203678 ,  0.33033482],
       [ 0.20464863,  0.61927097,  0.29965467]])
```

Splitting of a row vector, **a**,

```
>>> a = np.arange(12) + 1
>>> a
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

into sub-arrays can be achieved by:

```
np.split      >>> a1, a2 = np.split(a, [6])
>>> a1; a2
array([1, 2, 3, 4, 5, 6])
array([ 7,  8,  9, 10, 11, 12])
```

where we specify a split point. For a 2D array, we can perform vertical and horizontal splitting, depending on our goal:

```
>>> a = np.arange(24).reshape(4, 6)
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])

np.hsplit    >>> h1, h2, h3 = np.hsplit(a, [2, 4])
>>> h1
array([[ 0,  1],
       [ 6,  7],
       [12, 13],
       [18, 19]])
>>> h2
array([[ 2,  3],
       [ 8,  9],
       [14, 15],
       [20, 21]])
>>> h3
array([[ 4,  5],
       [10, 11],
       [16, 17],
       [22, 23]])
```

and

```
np.vsplit    >>> v1, v2 = np.vsplit(a, [2])
>>> v1
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
>>> v2
array([[12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

Lastly, a plain **concatenation** of row vectors is possible thanks to the **np.concatenate** function. Have a look:

```
np.concatenate >>> c = np.array([1, 2, 4, -3])
>>> np.concatenate([c, c])
array([ 1,  2,  4, -3,  1,  2,  4, -3])
```

or by the application of:

`np.tile`
constructs an array by
repeating `c` \times times.

```
>>> np.tile(c, 2)  
array([ 1,  2,  4, -3,  1,  2,  4, -3])
```

Also

```
>>> np.tile(h1, 3)  
array([[ 0,  1,  0,  1,  0,  1],  
       [ 6,  7,  6,  7,  6,  7],  
       [12, 13, 12, 13, 12, 13],  
       [18, 19, 18, 19, 18, 19]])
```

works great with 2D arrays, too.

3.4. Arrays of Randomness

One of the greatest, unseen at the first glance, benefits of NumPy's arrays is their speed of creation, transformation, and abundant numerical operations. With the backbone buried in C, NumPy equips the quants with the light-fast results. That has been applied, *inter alia*, within the `np.random` module implemented within NumPy.

In this Section, we will scan across most frequently used functions for randomisation and random selection. As you may guess—yes—the **Mersenne Twister** engine finds its implementation in NumPy, too.

3.4.1. Random Variables, Well Shook

Normal and Uniform

Working with the `np.random` module, the alias that we will be using through the entire book is:

```
>>> from numpy import random as npr
```

It is not a must to apply the same rule in your code. The choice is yours. However, if more programmers make a habit of the same kind, our future codes will become more readable.

A **random variable** drawn from the **Standard Normal distribution** we obtain by:

```
np.random.randn()
2.2922080128149576
```

The input parameters for the NumPy's random functions, in general, follow the same fashion:

```
# a row vector
>>> npr.randn(4)
array([-1.11792545,  0.53905832, -0.5961597 , -0.0191305])

# 2D array (5x2)
>>> npr.randn(5, 2)

# 3D array (3x2x4)
>>> npr.randn(3, 2, 2)
array([[[ -0.82913529,  0.08771022,  1.00036589, -0.38109252],
        [-0.37566942, -0.07447076,  0.43349633,  1.27837923]],

       [[ -0.63467931,  0.50839624,  0.21611601, -1.85861239],
        [-0.41931648, -0.1323289 , -0.03957024,  0.32600343]],

       [[ -2.04032305,  0.04625552, -0.67767558, -1.43943903],
        [ 0.52429643,  0.73527958, -0.65325027,  0.84245628]])])
```

however, always check the syntax to avoid mistakes.

A transition of the Gaussian distribution into the normalised Normal distribution is possible through a transformation or *rv*:

$$Z = \frac{X - \mu}{\sigma}$$

well known from the course on statistics. In Python we achieve it by:

Code 3.4

```
from numpy import random as npr
from matplotlib import pyplot as plt

mu = 3.5
sig = 3.0

np.random.normal X = npr.normal(mu, sig, size=10000)
Z = (X - mu)/sig
```

where we allowed to generate a 1D array with 10,000 random numbers drawn from the Normal distribution described by the mean of 3.5 and standard deviation 3.0, and to ensure that $Z \sim N(0, 1)$.

Uniform random variables populating one- and multi-dimensional NumPy arrays one can generate in one of the following ways. In general, and as usual, the distinction is made between integer and floating-point random numbers. The

```
np.random.rand >>> npr.rand(4)
array([ 0.84511312,  0.9885097 ,  0.04886809,  0.23211825])
```

command returns uniform rvs in $(0, 1)$ interval. For 2D arrays, apply:

```
>>> npr.rand(4, 3)
array([[ 0.64331143,  0.16146656,  0.87014589],
       [ 0.21740243,  0.74175504,  0.65302051],
       [ 0.79888551,  0.03124756,  0.22957403],
       [ 0.7046275 ,  0.08756251,  0.03058948]])
```

The function of `npr.randf` makes sure the rvs are in $(0, 1]$ interval:

```
np.random.randf >>> npr.randf(4)
array([ 0.35713493,  0.58978199,  0.05222273,  0.06566367])

np.random.random >>> npr.random(4)
array([ 0.04350119,  0.39515076,  0.66842705,  0.19802711])
```

the equivalent of the `np.random` function.

If solely the **integer** random numbers are requested, try to employ either `npr.randint` or `npr.random_integers`. The former ensures random integers from *low* (inclusive) to *high* (exclusive) while the latter random integers between *low* and *high*, inclusive:

```
np.random.randint >>> npr.randint(10, 20, 4)
array([19, 10, 13, 14])

np.random.random_integers >>> npr.random_integers(10, 20, 4)
array([16, 12, 20, 10])
```

and

```
>>> npr.randint(10, 20, size=(2, 2))
array([[17, 19],
       [13, 11]])
```

Randomness and Monte-Carlo Simulations

The random **choice** can have two forms. Namely, one can assume that our array's elements are equally probable to be selected (default), or there is some probability, **prob**, of picking corresponding to every item. Therefore, within a default setup we can obtain:

np.random.choice

```
>>> x = np.array([1, 2, 3, 4, 5, 6])
>>> prob = np.array([1/6, 1/6, 1/6, 1/6, 1/6])

>>> npr.choice(x, 3)
array([5, 2, 3])
```

equivalent to:

```
>>> npr.choice(x, 3, p=prob)
array([5, 2, 3])
```

However, if we set a new probability vector, then:

```
prob = np.array([1/6, 5/6, 0, 0, 0, 0])
for i in range(8): # 8 simulations
    print(npr.choice(x, 3, p=prob))

[2 1 2]
[2 2 1]
[2 2 2]
[1 2 1]
[2 2 2]
[2 2 2]
[2 2 2]
[1 2 2]
```

as expected. For the latter, if you want to be double sure, run:

Code 3.5

```
import numpy as np
from numpy import random as npr

x = np.array([1, 2, 3, 4, 5, 6])
prob = np.array([1/6, 5/6, 0, 0, 0, 0])

n1, n2, n3, n4, n5, n6 = 0, 0, 0, 0, 0, 0 # setting initial values
Nsim = 1000000

for i in range(Nsim):
    tmp = npr.choice(x, 3, p=prob)
    n1 += list(tmp).count(1)
    n2 += list(tmp).count(2)
    n3 += list(tmp).count(3)
    n4 += list(tmp).count(4)
    n5 += list(tmp).count(5)
    n6 += list(tmp).count(6)

print(1/6, 5/6, 0., 0., 0., 0.)
print(n1/Nsim/3, n2/Nsim/3, n3/Nsim/3,
      n4/Nsim/3, n5/Nsim/3, n6/Nsim/3)
```

which is a simple **Monte-Carlo simulation** with 1,000,000 trials run to verify that the assumed probabilities are in agreement with those we can get from the simulation. The output is:

```
0.16666666666666666 0.8333333333333334 0.0 0.0 0.0 0.0
0.166619 0.833381 0.0 0.0 0.0 0.0
```

Well... excellent! Let's have a drink. Bottoms up! ☺

Heading to Las Vegas? A good **shuffle** is a skill the machines can display a mastery in. Python's NumPy does it without breathing. We may shuffle, in a random way, a random (already!) array as follows:

```
>>> npr.seed(3)
>>> m = npr.randint(1, 10, size=(2, 2))
>>> x = m.copy()
>>> m
array([[9, 4],
       [9, 9]])

>>> npr.shuffle(x)
array([[9, 9],
       [9, 4]])
```

No big deal, right? Hold on.

Let's conduct a Monte-Carlo simulation (using the `np.random.shuffle` function) trying to estimate what is the probability of finding exactly the same shuffled 3x3 array. First, analyse the code:

Code 3.6

```
from numpy import random as npr

np.random.seed(3)
m = npr.randint(1, 10, size=(3, 3))
x = m.copy()
npr.shuffle(x) # works in-place
print(x)
print()

Nsim = 100000000 # a number of simulations
n = 0

for i in range(Nsim):
    tmp = npr.randint(1, 10, size=(3, 3))
    cmp = (tmp == x)
    if(cmp.all()):
        n += 1

print("n      = %g" % n)
print("Prob = %.9f" % (n/Nsim))
```

In the beginning, we use the `npr.seed(3)` function for making 3x3 array of **x** —frozen:

```
[[9 1 6]
 [9 4 9]
 [4 6 8]]
```

Next, we run 100,000,000 simulations. In the loop, the local variable **tmp** generates a temporary matrix 3x3 with the integer random

numbers to be between $[1, 10)$. The boolean array of **cmp** stores the results of a comparison between **tmp** and **x**. If all elements of **cmp** are of value *True* then we increase the total number of "lucky" counts. Finally, the results are displayed: both **n** and the estimated probability. And if you are a truly lucky guy, you may see the following output from this simulation:

```
n      = 1
Prob = 0.000000001
```

I wasn't! ☹ Can you find out "why"? Well, the probability of hitting:

```
[[9 1 6]
 [9 4 9]
 [4 6 8]]
```

matrix is:

$$\frac{1}{9^9} = 2.58 \times 10^{-9} = 0.00000000258$$

based on the definition of the `npr.randint` function.

Let's consider another two examples.

Personally, I love the game of *LOTTO*. As previously highlighted, it is all about picking 6 numbers to be between 1 and 49. Randomly. Let's use `npr.choice` function again in order to estimate the probability of hitting a lucky "6". We know that it is equal:

$$\text{Prob} = \frac{1}{C_{49}^6} = \left(\frac{49!}{6!(49-6)!} \right)^{-1} = \frac{1}{13983816}$$

i.e. 0.000000071511 with nearly 14 millions of all combinations. Surprisingly, even those odds do not stop some people from getting rich! ☺

Our mission can be accomplished if we run:

Code 3.7

```
from numpy import random as npr

Nsim = int(1e9)
n = 49
k = 6

data = []

for i in range(Nsim):
    tmp = npr.choice(n, k)
    tmp = set(tmp)
    if (tmp not in data) and (len(tmp) == k):
        data.append(tmp)

print("N      = %g" % len(data))
print("Prob = %.12f" % (1/len(data)))
```

After few hours, we find:

```
N      = 13983816
Prob = 0.000000071511
```

What is interesting about [Code 3.7](#) is the use of Python's **sets**. The function of `npr.choice` returns an array with k numbers drawn from a n -large sample. Unfortunately, those numbers may be the same in **tmp** therefore converting NumPy array into a set makes a lot of sense. Only when a number of elements of **tmp** is equal k , we allow to append it to **data** (a plain list), if **tmp** has not been in it so far.

A second useful thing you may derive for our *LOTTO* game is the probability of hitting your lucky six-number combination—one more time! I used to play with this system in late 90s what, in fact, brought me a lot of fun and hope to fulfil my dreams—sooner.

Say, you are like me, and you randomly pick 6 out of 49 numbers—your lucky "6". The strategy is to play two times a week and... to wait for your *earlier* retirement. With NumPy library in Python, the estimation of the waiting time is straightforward. Analyse:

Code 3.7a

```
import numpy as np

def lotto_numbers():
    ok = False
    while not ok:
        x = np.random.choice(49, 6, replace=False)
        x.sort()
        tmp = np.where(x == 0)
        (m, ) = tmp[0].shape
        if(m == 0):
            ok = True
    return x

fav = lotto_numbers() # choose your favourite 6 numbers
print(fav)

match = False
i = 0 # iterations
while not match:
    tmp = lotto_numbers()
    cmp = (tmp == fav)
    i += 1
    if cmp.all():
        match = True

print(tmp) # print matching number when found
print("Iterations: %g\nProbability = %.2e" % (i, 1./i) )
```

First, we create a custom function `lotto_numbers` with a 6-element array of random integers drawn from a set $[0, 49]$ and we sort them all. Because 0 is not the part of the game, we need to check for its presence and draw again, if detected. A temporary array, **tmp**, inside `lotto_numbers` stores an index where **x** has a zero-element. Since we use the `random.choice` function with a parameter `replace=False`, we can be sure that there is only one zero in **x**, if drawn. Therefore, **m** should be equal 0 or 1.

In the main program of [Code 3.7a](#) the loop is alive until we find the match between a (random) selection of our favourite 6 numbers, **fav**, and a new *LOTTO* draw. The array of **cmp** holds six boolean elements as the result of comparison. Therefore, making the arrays sorted is an essential part of the code. Finally, the function of **.all()** returns *True* or *False* if all elements are *True* (or not), respectively. And, as in life, the *Truth* is better than false, right? ☺

An exemplary outcome you can get is:

```
[ 1  2  4 11 13 18]
[ 1  2  4 11 13 18]
Iterations: 37763
Probability = 2.65e-05
```

meaning that playing with the system defined above, we would need to wait:

```
>>> 37763/(2*52)
363.1057692307692
```

years for your lucky combination to repeat itself! Note, that the outcome is just a single run of the simulation. As a homework, could you run 10,000 simulations (modifying [Code 3.7a](#) respectively) and plot the distribution of "years" derived as `int(i/(2*52))`. Also, what could be modified to make [3.7a](#) faster?

Aside a random choice powered by Mersenne Twister, **permutations** of *N*-array are accessible in NumPy by:

```
np.random.permutation      >>> np.random.permutation(6)
                             array([3, 5, 2, 4, 1, 0])
```

or

```
>>> x = np.identity(3)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> npr.permutation(x) # in-place
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  0.,  1.],
       [ 0.,  1.,  0.]])
```

Saying that, here is one more challenge: modify [Code 3.6](#) and find a number of all permutations for 3x3 identity array. Compare.

3.4.2. Randomness from Non-Random Distributions

As we have tasted at the beginning of this Section, NumPy provides us with some ready-to-use functions for generating *N*-dimensional array filled with random numbers. Our choice is confined solely to the distribution which we wish to use.

What follows, is a full list of the function names and the corresponding distributions. The detailed information on the proper call and syntax you may find if you look at: <http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.random.html>.

A typical way to call those functions is:

```
>>> import numpy as np
>>> np.random.gumbel(0.1, 2.5, size=(1, 4))
array([[ 1.57207262,  1.29800811, -2.9606535 , -1.32470797]])
```

or, as we have mentioned earlier:

```
>>> from numpy import random as npr
>>> npr.pareto(0.1, size=(3, 1))
array([[ 2.94280749e-01],
       [ 3.79999371e+00],
       [ 4.24316547e+02]])
```

Function	Distribution
beta	Beta distribution
binomial	binomial distribution
chisquare	chi-square distribution
dirichlet	Dirichlet distribution
exponential	exponential distribution
f	F distribution
gamma	Gamma distribution
geometric	geometric distribution
gumbel	Gumbel distribution
hypergeometric	Hypergeometric distribution
laplace	Laplace or double exponential distribution
logistic	logistic distribution
lognormal	log-normal distribution
logseries	logarithmic series distribution
multivariate_normal	multivariate normal distribution
negative_binomial	negative binomial distribution
noncentral_chisquare	noncentral chi-square distribution
noncentral_f	noncentral F distribution
normal	normal (Gaussian) distribution
pareto	Pareto II or Lomax distribution
poisson	Poisson distribution
power	power distribution
rayleigh	Rayleigh distribution
standard_cauchy	standard Cauchy distribution with mode = 0
standard_exponential	standard exponential distribution
standard_gamma	standard Gamma distribution
standard_normal	standard Normal distribution, $N(0,1)$
standard_t	standard Student's t distribution
triangular	triangular distribution
uniform	uniform distribution
vonmises	von Mises distribution
wald	Wald, or inverse Gaussian, distribution
weibull	Weibull distribution
zipf	Zipf distribution

The problem with those functions is in their limited application. If a cumulative distribution function or percent point function is sought after, you have to reach for SciPy's library of `scipy.stats`. The `scipy.stats` module will be addressed in greater detail within *Volume II*, however, now, for fulfilling your curiosity, let's consider a practical example of the MasterCard Inc. stock data and its statistical analysis—in the next Section.

3.5. Sample Statistics with `scipy.stats` Module

SciPy is an open source Python library used by scientists, analysts, and engineers. It contains a rich set of tools and useful functions for optimization, linear algebra, integration, interpolation, special functions, Fourier transform, signal and image processing, and not forgetting about ODE solvers. It also constitutes the established and respected gateway to the computational **statistics** in Python.

The module of `scipy.stats` equips us with over one hundred functions including continuous, multivariable and discrete distributions; 81 statistical functions; circular and contingency table function; and plot-tests. Depending on the complexity of our data and problem to be solved, `scipy.stats` is a right place where you can commence exploring a variety of options Python has to offer. Below, we will make use of a tiny fraction from the `scipy.stats`'s arsenal (see <http://docs.scipy.org/doc/scipy/reference/stats.html>).

Our goal is to provide the fundamental data description based on the **financial data** fetched from the Wall Street trading sessions. We will perform the analysis of the **daily returns** of MA (MasterCard Inc.) stock. All the steps can be summarised as follows:

1. Using Yahoo! Finance data provider retrieve a daily adjusted-close stock prices-series for a period of 5 years;
2. Transform price- into return-series and treat it as an input data sample;
3. Assuming that our dataset can be described by the Normal distribution, fit the data with the Normal probability density function (pdf) and estimate a daily mean return and standard deviation;
4. Transform data to the standardised Normal distribution and confirm in the process of fitting of pdf that both model's parameters are equal 0 and 1, respectively;
5. Find the cumulative distribution function;
6. Assuming the significance level of 0.05, find the **Value-at-Risk** for MA daily returns; reconfirm that for such VaR, the probability is 0.05; use the analytical model;
7. Write a function which derives VaR based on the empirical integration of MA's returns histogram; compare both VaRs.

In this case study, we will demonstrate the essential functions required to perform those basic statistical analyses—a good start to building your own experience with `scipy.stats` and Python (have a look at <http://docs.scipy.org/doc/scipy/reference/tutorial/stats.html> for an introductory tutorial on `scipy.stats`).

Ready? Let' do it...

`scipy.stats` delivers an independent method for generating arrays filled with random numbers drawn from a specific distribution. In case of the Normal distribution:

Code 3.8

```

from numpy import random as npr
from scipy.stats import norm

mu = 0.001
sig = 0.02

rvs1 = npr.normal(mu, sig, 100000) # NumPy
.rvs rvs2 = norm.rvs(mu, sig, 100000) # SciPy's stats

```

the random array of **rvs2** is generated with a help of the SciPy's **stats** module while **rvs** in the way we have discussed within the previous Section. Both methods are equivalent.

The difference you may notice is that SciPy contains a broader sample of the statistical distributions than NumPy.

3.5.1. Downloading Stock Data From Yahoo! Finance

For many years Yahoo! Finance has been a great source of financial data—free of charge. That includes stock daily price-series for almost all stocks traded within major stock exchanges. Here, we are interested in MA's adjusted-close price series.

The easiest way to fetch Yahoo! Finance data in Python is by employing a dedicated module of **pandas-datareader** which, very soon, will replace **pandas.io** (see <https://github.com/pydata/pandas-datareader>). If you use Anaconda Python 3.5 distribution, most probably you need to install it, if not present already:

```
$ conda install pandas-datareader
```

```

Fetching package metadata: ....
Solving package specifications: .....
Package plan for installation in environment //anaconda:

```

The following packages will be downloaded:

package	build	
pandas-datareader-0.2.0	py35_0	40 KB
Total:		213 KB

The following NEW packages will be INSTALLED:

```
pandas-datareader: 0.2.0-py35_0
```

```

Fetching packages ...
pandas-dataarea 100% |#####| Time: 0:00:00 63.42 kB/s
Extracting packages ...
[ COMPLETE ]|#####| 100%
Unlinking packages ...
[ COMPLETE ]|#####|100%
Linking packages ...
[ COMPLETE ]|#####|100%

```

In other case, install via :

```
$ pip3.5 install pandas-datareader
```

typed and executed in the Terminal. Now, we are ready!

Code 3.9 Sample Statistics for NYSE:MA daily stock returns.

```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
import pandas_datareader.data as web

# 1. DATA DOWNLOAD
#
# Fetching Yahoo! for MasterCard Inc. (MA) stock data
data = web.DataReader("MA", data_source='yahoo',
                      start='2010-05-13', end='2015-05-13')['Adj Close']
cp = np.array(data.values) # daily adj-close prices
ret = cp[1:]/cp[:-1] - 1 # compute daily returns
```

In this part, we download five years of MA price-series in the form of **pandas**' DataFrame (more on **pandas** for quantitative finance within *Volume II* of *Python for Quants*). Next, we solely extract values from **data** DataFrame and store them as a NumPy 1D array of **cp** (adjusted-close). At last, the price-series is converted into array of daily returns, **ret**.

3.5.2. Distribution Fitting. PDF. CDF.

Continuing Code 3.9,

```
# 2. APPLIED STATISTICS
#
# Fit ret with N(mu, sig) distribution and estimate mu and sig
.fit mu_fit, sig_fit = norm.fit(ret)
print("Daily")
print(" mu_fit, sig_fit = %.4f, %.4f" % (mu_fit, sig_fit))
print("Annualised")
print(" mu, stdev = %.4f, %.4f" % ((mu_fit+1)**364-1,
                                   sig_fit*(252**0.5)))

# Find PDF
dx = 0.001 # resolution
x = np.arange(-5, 5, dx)
.pdf pdf = norm.pdf(x, mu_fit, sig_fit)

np.sum print("Integral pdf(x; mu_fit, sig_fit) dx = %.2f" %
              (np.sum(pdf * dx)))
```

Every distribution class of **scipy.stats** has the same **methods**. The method of **norm.fit** takes as an argument the array with numbers and making use the Normal distribution, **scipy.stats.norm**, as the model—it fits the data with a corresponding **probability density function** (pdf). As the output, the best estimates of the mean and standard deviation are returned. We display their values and, in addition, compute the annualised *expected* return and volatility for MA stock:

```
Daily
mu_fit, sig_fit = 0.0013, 0.0175
Annualised
mu, stdev = 0.5854, 0.2779
```

Given the annualised return of 58.5% at 27.8% of risk, it makes MA quite attractive stock to invest (as of May 13, 2015), would you agree?

The probability density function of the Normal distribution can be derived as:

```
pdf = norm.pdf(x, mu_fit, sig_fit)
```

for the estimated parameters of the model. By \mathbf{x} we describe a set of all possible discrete values defining "x-axis" domain. Therefore, **pdf** function will be spanned from -5 to 5 with 0.001 resolution of **dx**. Lastly, by employing integration (conducted with an aid of `np.sum` function) we verify that:

$$\int_{-\infty}^{\infty} N(x; \mu, \sigma) dx = 1$$

where N is the pdf of the Normal distribution,

```
Integral pdf(x; mu_fit, sig_fit) dx = 1.00
```

The **cumulative density function** (cdf) can found by:

```
.cdf      cdf = norm.cdf(x, mu_fit, sig_fit)
```

and plotted, if necessary.

What appears to be a simple academic exercise in statistics, in Python is at the same level of simplicity, namely, a transformation of a data sample from (assumed by model; confirmed by fitting) Normal distribution to the **standardised Normal distribution**. Continuing [Code 3.9](#), we add:

```
# --standardised Normal distribution
z = (ret - mu_fit)/sig_fit # transformation
mu_fit2, sig_fit2 = norm.fit(z) # estimation
print("\nmu_fit2, sig_fit2 = %.4f, %.4f" % (mu_fit, sig_fit))
stdpdf = norm.pdf(x, mu_fit2, sig_fit2) # N(0,1) pdf
print("Integral pdf(x; mu_fit2, sig_fit2) dx = %.2f" %
      (np.sum(stdpdf * dx)))
```

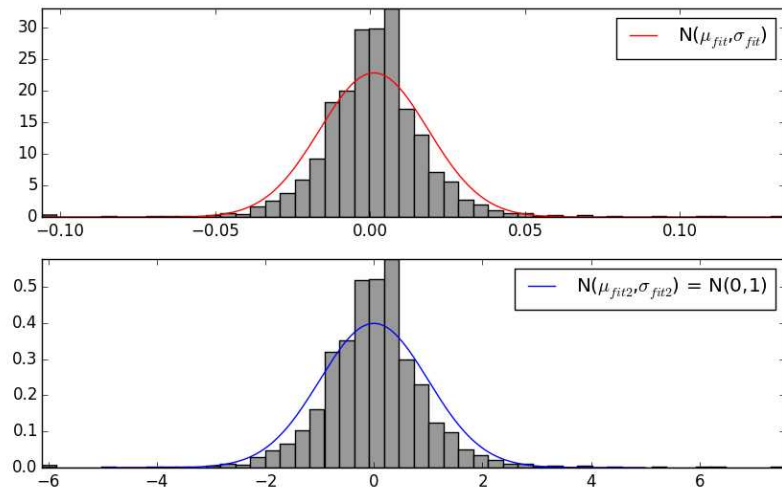
where we use:

$$Z = \frac{X - \mu}{\sigma}$$

transformation of the normal random variables to $Z \sim N(0, 1)$. Also in this case:

```
Integral pdf(x; mu_fit2, sig_fit2) dx = 1.00
```

and



```
mu_fit2, sig_fit2 = -0.0000, 1.0000
```

as expected. As far as the numbers are lovely, the picture tells a whole story. Extending further 3.9:

```
# --Plotting PDFs
fig, ax1 = plt.subplots(figsize=(10, 6))
#
plt.subplot(211)
plt.hist(ret, bins=50, normed=True, color=(.6, .6, .6))
plt.hold(True)
plt.axis("tight")
plt.plot(x, pdf, 'r', label="N($\mu_{fit}$,$\sigma_{fit}$)")
plt.legend()
#
plt.subplot(212)
plt.hist(z, bins=50, normed=True, color=(.6, .6, .6))
plt.axis("tight")
plt.hold(True)
plt.plot(x, stdpdf, 'b', label="N($\mu_{fit2}$,$\sigma_{fit2}$) =
                                     N(0,1)")

plt.legend()
plt.show()
```

we combine both pdf's with the original (upper) and transformed (bottom) MA daily returns distributions (see the chart above).

3.5.3. Finding Quantiles. Value-at-Risk.

The starting point is trivial. For any given distribution, k is called α -quantile such:

$$Pr(X < k) = \alpha$$

It means that in case of the Normal distribution, we have to perform the integration in the interval:

$$\int_{-\infty}^k N(x; \mu, \sigma) dx = \alpha$$

To 3.9, we add:

```
# --Finding k such that Pr(X<k)=alpha given alpha
alpha = 0.05
k = norm.ppf(alpha, mu_fit, sig_fit)
print("\nk = %.5f" % k)
```

where `.ppf` function stands for the inverse of the cumulative density function or **percent point function** (ppf). The task can be reversed and formulated as follows: given k find α . In Python we solve it by:

```
# --Finding Pr(X<k) given k
pr = norm.cdf(k, mu_fit, sig_fit)
print("Pr(X<k) = %.5f" % pr)
```

what delivers the output:

```
k = -0.02753
Pr(X<k) = 0.05000
```

if the same value of k is used for the latter computation. This result, derived here based on MA stock data, simply communicates that there is 5% of chances that on the next day, MA may lose 2.75% or more. Saying so, in fact, we define a quantitative measure of risk—VaR or **Value-at-Risk** described more formally as:

$$\Pr(L \leq -\text{VaR}_{1-\alpha}) = \alpha$$

where by L we denote a loss (in percent) that an asset can experience on the next day. Having that, there are at least two methods of finding VaR for the sample data: analytical and empirical.

The **analytical** way is based on (1) fitting the distribution with the model, e.g. the Normal distribution, and (2) finding α -quantile or, saying in terms of finance, $(1-\alpha)\text{VaR}$, based on the model integration as have performed above. The **empirical** method is through a "manual" integration of the distribution. It requires a design of a special custom function:

```
def findvar(ret, alpha=0.05, nbins=200):
    # Function computes the empirical Value-at-Risk (VaR) for
    # return-series
    # (ret) defined as NumPy 1D array, given alpha
    #
    # compute a normalised histogram (\int H(x)dx = 1)
    # nbins: number of bins used (recommended nbins>50)
    hist, bins = np.histogram(ret, bins=nbins, density=True)
    wd = np.diff(bins) # resolution of H(x)
    # cumulative sum from -inf to +inf
    cumsum = np.cumsum(hist * wd)
    # find an area of H(x) for computing VaR
    crit = cumsum[cumsum <= alpha]
    n = len(crit)
    # (1-alpha)VaR
    VaR = bins[n]
    return VaR
```

We employ the `np.histogram` function for creating the histogram with `nbins` in resolution. Next, the application of the `np.cumsum` function generates a 1D array with a running cumulative sum for all histogram bars. Using it, we screen for the region less than α . Now it is easy to find VaR since it is the last value of the `bins` vector returned by the `np.histogram` function.

Lastly, we compare both approaches finalising [Code 3.9](#) with:

```
# --Compare Value-at-Risk numbers
cl = 100*(1 - alpha) # confidence level
print("\n%% VaR (analytical) = %.5f" % (cl, -k))
VaR = findvar(ret, alpha=0.05)
print("%% VaR (empirical) = %.5f" % (cl, -VaR))
```

what delivers:

```
95% VaR (analytical) = 0.02753
95% VaR (empirical) = 0.02555
```

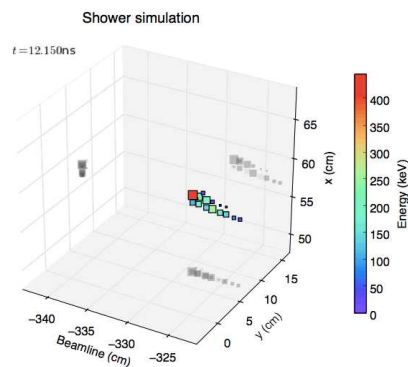
and points at the very good agreement between both methods we have tested. Although for a single investor, the difference of 0.198% in VaR is negligible, for a bank it may be a huge headache. ☺

3.6. 3D, 4D Arrays, and N -dimensional Space

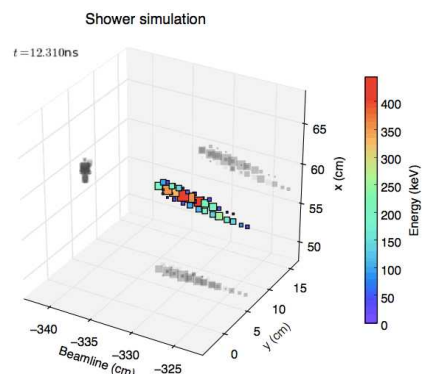
Moving in **3D** (three dimensional) space is natural to us and is intuitively described by length, width, and height. The latter sets a system of coordinates from day one when we are born. We learn it, we assimilate it. Our body is, in its first approximation, defined as a 3D object and there is no question about it. When we move across space, we traverse in time. Therefore, the time introduces **4D** (the fourth dimension) to our description. Are we able to imagine the fifth dimension using the same example? I believe so. If you look at your body moving along the invertible axis of time, the changes of your skin colour or skin texture in time could be regarded as information defining the fifth dimension or **5D** (hyper-)space.

Can we visualise 5D data collection? The answer is *yes* and *no*. *No* because we cannot plot a static image showing full 5D information in "one shot". *Yes* because we can create an animation where the slices of 5D data are captured and displayed on the screen.

One of the greatest examples of the Python code (but too complicated to be explained within this book) is *Animator5D* written by Ben Bartlett which you can reach and download from <https://github.com/bencbartlett/Animator5D> GitHub repository. *Animator5D* uses a simple framework for rendering 5-dimensional animations (x , y , z , time, some colour value) as an animated GIF. Originally written to reconstruct electromagnetic (EM) scattering (a shower) in the CMS detector (<http://cms.web.cern.ch/news/cms-detector-design>) over time, the program was designed to visualise the energy deposited per layer as the *fifth* quantity. What follows are two slices of information in such defined 5D space:



Output of Animator5D
(c) 2015 by Ben Bartlett



3.5.1. Life in 3D

In Python's NumPy, a **manual construction** of 3D array out of 2D arrays takes the following steps, e.g.:

```
>>> x = np.array([[1,2], [4,5], [7,8]])
>>> x
array([[1, 2],
       [4, 5],
       [7, 8]])
>>> y = -1 * x.copy()
>>> y
array([[ -1, -2],
       [ -4, -5],
       [ -7, -8]])
>>> x.ndim; y.ndim # confirm we deal with 2D arrays
2
2
>>> z = np.array([x, y]) # 3D array
>>> z
array([[[ 1,  2],
        [ 4,  5],
        [ 7,  8]],
       [[-1, -2],
        [-4, -5],
        [-7, -8]]])
>>> z.ndim
3
>>> z.shape
(2, 3, 2)
```

i.e. we end up with 2x(3x2) 3D array. Pay a closer attention of how the use of the inner square brackets is made. We also can confirm:

```
>>> (z[0] == x).all(); (z[1] == y).all()
True
True
```

therefore by indexing:

```
z[1]
```

we get an access to the 2nd slice of 3D "cube". As expected, the addition:

```
>>> z[0] + z[1]
array([[0, 0],
       [0, 0],
       [0, 0]])
```

flattens 3D structure via element-wise projection onto 2D plane. That operation is different than:

```
>>> f = z.flatten(); f
array([ 1,  2,  4,  5,  7,  8, -1, -2, -4, -5, -7, -8])
```

that creates a 1D array containing all 3D elements as a plain "list". In some cases, it is convenient to work with all data regardless of the number of dimensions.

A **reversal** is possible, i.e. going from 1D to 3D shape:

```
>>> f.reshape(2, 3, 2)
array([[[ 1,  2],
        [ 4,  5],
        [ 7,  8]],

       [[-1, -2],
        [-4, -5],
        [-7, -8]]])
```

The element-wise **sum** can be achieved with a help of the `np.sum` function where the direction (axis) along which the summation takes place appears in the function calling as an additional parameter:

```
>>> np.sum(z, axis=0)
array([[0, 0],
       [0, 0],
       [0, 0]])
>>> np.sum(z, axis=1)
array([[ 12,  15],
       [-12, -15]])
>>> np.sum(z, axis=2)
array([[ 3,  9, 15],
       [-3, -9, -15]])
```

The **extension** of 2x3x2 3D array of **z** by inclusion of an additional 3x2 2D array of ones

```
>>> z
array([[[ 1,  2],
        [ 4,  5],
        [ 7,  8]],

       [[-1, -2],
        [-4, -5],
        [-7, -8]]])
>>> o = np.ones((3, 2))
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

(depending on the desired position of **o** slice) is accessible by:

```
>>> z1 = np.vstack([z, o[np.newaxis, ...]])
array([[[ 1,  2],
        [ 4,  5],
        [ 7,  8]],

       [[-1, -2],
        [-4, -5],
        [-7, -8]],

       [[ 1.,  1.],
        [ 1.,  1.],
        [ 1.,  1.]])
```

or

```
>>> z2 = np.vstack([o[np.newaxis,...], z])
```

and

```
>>> z1.ndim; z1.shape # z2.ndim; z2.shape
3
(3, 3, 2)
```

3.5.2. Embedding 2D Arrays in 4D, 5D, 6D

As we have discussed, if we consider K 2D arrays ($N \times M$) stacked "slice-by-slice" as a 3D array then its dimensions will be $K \times N \times M$. The best way to image that case is an index card drawer. A card index is described by length and width, $N \times M \times 0$ cm, and we place K of them stacked together inside. Therefore, K cards can be described by $K \times N \times M$ 3D-cuboid. The operation of stacking K cards up can be thought of as **embedding** 2D objects in 3D space.

Can we take it higher and talk about embedding 2D objects in 4D or even higher dimensions? Yes. Let this journey to begin...

Imagine the following Monte-Carlo **simulation**. The design of this simulation is to show you the evolution of 2D array in time (in 3D) in K discrete time steps. In the beginning with have a 3x3 array filled with zeros, **a**. Next, we create a temporary 2D array of zeros, **tmp**, and randomly select two indexes, i and j and assign some random value, **val**, to be between 1 and 10 such that:

```
tmp[i, j] = val
```

In time step #2, we take the last slice of **a** and add (element-wise addition) matrix **tmp** to it. The resultant matrix is added to **a** therefore the dimensions of **a** array change from:

```
(1, 3, 3)
```

to

```
(2, 3, 3)
```

and so on. Let's assume K to be equal 69 (a number of simulated time steps). Using the knowledge from the precedent Section, we write:

Code 3.10

```
import numpy as np

np.random.seed(2)

a = np.zeros((1, 3, 3))

for k in range(1, 70):
    i = np.random.random_integers(0, 2)
    j = np.random.random_integers(0, 2)
    val = np.random.random_integers(1, 10)
    tmp = np.zeros((3, 3))
    tmp[i, j] = val
    b = a[-1].copy() + tmp
    a = np.vstack([a, b[np.newaxis, ...]])
```

For **k** = 1, the arrays of **tmp**, **b**, and **a** are:

```
[[ 0.  9.  0.]    # tmp
 [ 0.  0.  0.]
 [ 0.  0.  0.]]

[[ 0.  9.  0.]    # b
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

```

[[[ 0.  0.  0.]
   [ 0.  0.  0.]
   [ 0.  0.  0.]]

 [[ 0.  9.  0.]
   [ 0.  0.  0.]
   [ 0.  0.  0.]]]
# a

```

For $k = 2$ we may end up with **a** given as:

```

[[[ 0.  0.  0.]
   [ 0.  0.  0.]
   [ 0.  0.  0.]]

 [[ 0.  9.  0.]
   [ 0.  0.  0.]
   [ 0.  0.  0.]]

 [[ 0.  9.  0.]
   [ 0.  0.  0.]
   [ 0.  0.  9.]]]

```

After $k = 4$ we can see:

```

[[[ 0.  0.  0.]
   [ 0.  0.  0.]
   [ 0.  0.  0.]]

 [[ 0.  9.  0.]
   [ 0.  0.  0.]
   [ 0.  0.  0.]]

 [[ 0.  9.  0.]
   [ 0.  0.  0.]
   [ 0.  0.  9.]]

 [[ 0.  9.  0.]
   [ 0.  0.  0.]
   [ 0.  6.  9.]]

 [[ 0.  9.  0.]
   [ 0.  0.  0.]
   [ 5.  6.  9.]]]

```

By adding to [Code 3.10](#) three more lines:

```

print(a.ndim)
print(a.shape)
print(a[-1]) # display the last slice from the pile

```

after all 1+69 time-steps we print the final outcomes:

```

3
(70, 3, 3)
[[ 29.  28.  42.]
 [ 40.  62.  60.]
 [ 46.  24.  20.]]

```

i.e, our 3D array of **a** stores 70 two-dimensional slices and the last "frame" stores the end result of the simulation.

For the need of reproduction of the results, we make use of the `np.random.seed(2)` function at the beginning of [Code 3.10](#).

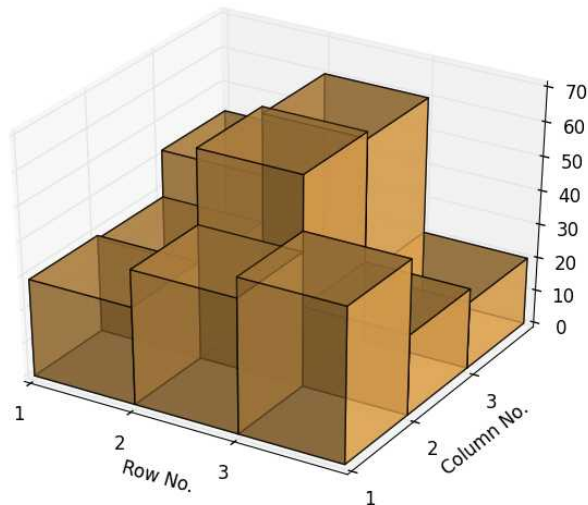
If you look closer at what we obtained, i.e.

```
[[ 29.  28.  42.]
 [ 40.  62.  60.]
 [ 46.  24.  20.]]
```

then it is obvious that we can enumerate both rows and columns with indexes equal [1, 2, 3]. If so, what would you say to plot that final result in a form of a **3D bar plot**?! The values of `a[-1]` become the heights of the bars. Therefore, by extending [3.10](#):

```
# Plot 3D bar plot for 3D array of a[-1]
xpos = [1, 1, 1, 2, 2, 2, 3, 3, 3]
ypos = [1, 2, 3, 1, 2, 3, 1, 2, 3]
zpos = np.zeros(9)
dx = np.ones(9)
dy = np.ones(9)
dz = a[-1].flatten()
#
ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color='#f2b158', alpha=0.7)
#
ax.view_init(ax.elev, ax.azim+0)
plt.xticks([1, 2, 3])
plt.yticks([1, 2, 3])
plt.xlabel("Row No.")
plt.ylabel("Column No.")
plt.show()
```

the visualisation of `a[-1]` emerges:



Within our exercise we built a simulation of 2D matrix in time by adding into a random cell a random value chosen between 1 and 10, and every next step we assigned the sum over all previous cells. As one might expect, the end result is purely random since we have not included any analytical model in the background nor we did not multiply any of `a[i]` slices by some fixed 3x3 probability matrix (i.e. assigning 9 weights to all 9 cells in `a[i]` array).

You are more than welcome to modify the code and experiment a bit with it.

Now, let's study the fundamentals of **4D space** in NumPy. It will act on your imagination and (hopefully) inspire you on the ways of how to manipulate L different experiments based on K simulations of the time-evolution of $N \times M$ 2D arrays.

The simplest way to create 4D array in NumPy would be, e.g.:

```
>>> x4 = np.array([[[[1, 2, 3]]]])
>>> x4.ndim
4
>>> x4.shape
(1, 1, 1, 3)
```

but, somehow, it does **not** look **sexy**. Not at all. It is flat and there is not too much to touch, to embrace, including the view... ☹

So, how about that one:

```
>>> x4 = np.array([[ [1]], [2]], [[3]] ])
>>> x4.ndim
4
>>> x4.shape
(1, 3, 1, 1)
>>> x4
array([[[[1]],
          [[2]],
          [[3]]]])
```

A quick check reveals that:

```
>>> x4[0][1]
```

```
array([[2]])
```

is 1x1 2D array. Note double inner square brackets.

All inner arrays are 1x1. They are located in 3D space, therefore the dimensions are 3x(1x1). It is tempting to substitute those 1x1 arrays, for example, with 3x3 2D arrays which we choose as follows:

```
>>> a = np.array([[1,1,1], [1,2,1], [1,1,1]])
>>> b = np.array([[1,-1,1], [-1,1,-1], [1,-1,1]])
>>> c = np.array([[-2,1,-2], [1,1,1], [-2,1,-2]])
```

then

```
>>> x4 = np.array([[ a, b, c ]])
>>> x4.ndim
4
>>> x4.shape
(1, 3, 3, 3)
>>> x4
array([[[[ 1,  1,  1],
          [ 1,  2,  1],
          [ 1,  1,  1]],
        [[ 1, -1,  1],
          [-1,  1, -1],
          [ 1, -1,  1]],
        [[-2,  1, -2],
          [ 1,  1,  1],
          [-2,  1, -2]]]])
```

The 4D array of **x4** can be interpreted as the storage of the end-results of $K=3$ independent simulations (e.g. conducted with a help of [Code 3.10](#)).

The first dimension of **x4** points at the first "experiment" ($L=1$),

```
>>> x4.shape
(1, 3, 3, 3)
```

Now, imagine that by repeating the whole experiment once again ($L=2$) we have obtained exactly the same results but due to some background noise each end-2D matrix (3x9 elements) has been altered (contaminated) at the same level. The noise matrix was:

```
>>> np.random.seed(9)
>>> n = np.random.random((3, 3))/10
>>> n
array([[ 0.00103742,  0.05018746,  0.04957733],
       [ 0.01338295,  0.01421111,  0.02185587],
       [ 0.04185082,  0.02481012,  0.00840597]])
```

such that the noise is additive to **a**, **b**, and **c**:

```
>>> x4 = np.array([[ a, b, c ], [a+n, b+n, c+n]])
>>> x4.ndim; x4.shape
4
(2, 3, 3, 3)
```

therefore, **x4** becomes:

```
>>> x4
array([[[[ 1.          ,  1.          ,  1.          ],
         [ 1.          ,  2.          ,  1.          ],
         [ 1.          ,  1.          ,  1.          ]],

       [[ 1.          , -1.          ,  1.          ],
         [-1.          ,  1.          , -1.          ],
         [ 1.          , -1.          ,  1.          ]],

       [[-2.          ,  1.          , -2.          ],
         [ 1.          ,  1.          ,  1.          ],
         [-2.          ,  1.          , -2.          ]]],

      [[[ 1.00103742,  1.05018746,  1.04957733],
         [ 1.01338295,  2.01421111,  1.02185587],
         [ 1.04185082,  1.02481012,  1.00840597]],

       [[ 1.00103742, -0.94981254,  1.04957733],
         [-0.98661705,  1.01421111, -0.97814413],
         [ 1.04185082, -0.97518988,  1.00840597]],

       [[-1.99896258,  1.05018746, -1.95042267],
         [ 1.01338295,  1.01421111,  1.02185587],
         [-1.95814918,  1.02481012, -1.99159403]]]])
```

By repeating the experiment 100 times, we may **simulate** that all 3x(3x3) end-arrays return the same integer values (as given by **a**, **b**, and **c** arrays) affected by the noise level of the same magnitude.

The following code addresses this scenario:

Code 3.11

```
import numpy as np

a = np.array([[1,1,1], [1,2,1], [1,1,1]])
b = np.array([[1,-1,1], [-1,1,-1], [1,-1,1]])
c = np.array([[-2,1,-2], [1,1,1], [-2,1,-2]])
```



```

x4 = np.array([[a, b, c]]) # template
x4_model = x4.copy()

for nsim in range(100-1):
    n = np.random.random((3,3))/10
    noisy = np.array([a+n, b+n, c+n])
    x4 = np.vstack([x4, noisy[np.newaxis, ...]])

print(x4.ndim)
print(x4.shape)

```

We start with a pure template of 4D array **x4** (1x3x3x3) and in each simulation we add a random noise. Conducting 100 experiments we end up with:

```

4
(100, 3, 3, 3)

```

4D array of **x4**. Someone may wish to inspect the results. One of the best methods is to average 3 different outcomes (3x3) over 100 experiments. Extending [Code 3.11](#) by adding:

```

np.mean(x, axis=0)    avg = np.mean(x4, axis=0)
print()              print()
print(avg)            print(avg)

```

we display the averaged results:

```

[[[ 1.04871494  1.05082341  1.05504118]
   [ 1.0490031  2.04724781  1.04314526]
   [ 1.05434021  1.05296299  1.05146649]]

  [[ 1.04871494 -0.94917659  1.05504118]
   [-0.9509969  1.04724781 -0.95685474]
   [ 1.05434021 -0.94703701  1.05146649]]

  [[-1.95128506  1.05082341 -1.94495882]
   [ 1.0490031  1.04724781  1.04314526]
   [-1.94565979  1.05296299 -1.94853351]]]

```

Since the model "values" have been saved in **x4_model** object, we can find the **residuals** for every 3x3 "averaged" array by:

```
res = avg - x4_model
```

what accomplishes our interesting investigation of L experiments of $K=3$ different simulations.

Ready to move into **5D space**? Nah... Looks like a mounting headache. Have a break. However, let me just denote that one can combine both [Codes 3.10](#) and [3.11](#) in order to describe a time-evolution of $N \times M$ 2D arrays in K steps, repeated L times (number of independent experiments), conducted on P different days.

Having that, adding **6D** would be painless, i.e. (...) in R different laboratories dotted around the world. Averaging the end-results over world locations would be as easy as `avg6 = np.mean(x6, axis=0)`. ☺

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 **ufuncs** 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))))
```

```

# method 1 (loop and list)
t = time()
f1 = []
for i in range(1, 10000001):
    x = 0.00001*i
    f1.append(fun(x))
t1 = time()-t

# method 2 (list comprehension)
t = time()
x = [0.00001*i for i in range(1, 10000001)] # len(x) = 10,000,000
f2 = [fun(a) for a in x]
t2 = time()-t

# method 3 (NumPy's ufuncs)
t = time()
x = np.arange(1, 10000001) * 0.00001
f3 = np.sqrt(np.abs(np.sin(x-np.pi)*np.power(np.cos(x+np.pi), 2)) *
              (1+np.exp(-x/np.sqrt(2*pi)))))
t3 = time()-t

print(np.array(f1 == f2).all())
print(np.array(f2 == f3).all())
print("t1, t2, t3 = %.2f sec, %.2f sec, %.2f sec" % (t1, t2, t3))
print("speedups t2/t3, t1/t3 = %.2fx, %.2fx" % (t2/t3, t1/t3))

plt.figure(figsize=(8, 5))
plt.plot(x, f3, color='b')
plt.xlabel("x")
plt.axis("tight")
plt.ylabel("f(x)")
plt.show()

```

Having a pretty complex function, $f(x)$, it is convenient to define it as a custom function, `fun`. The method #1 performing the computations in the loop is a classical way the most of programmers would think of to apply in the first place. It's plain and logical as all steps are easily separated. However, it's not the best solution. In method #2, the use of a list comprehension is more "Pythonic" way of calling the function of `fun` and accumulating all results directly inside the list. The grid is huge and fine and covers 10 million points.

Method #3 beats the competitors delivering the same results over 10 to 12 times faster, respectively:

```

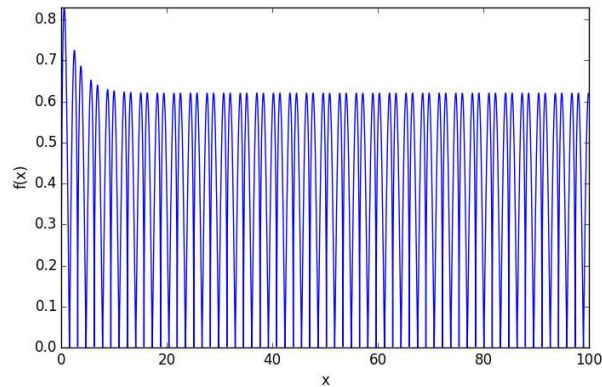
True
True
t1, t2, t3 = 16.11 sec, 13.08 sec, 1.28 sec
speedups t2/t3, t1/t3 = 10.24x, 12.62x

```

To make sure that all three methods return the same values, the boolean check with a help of the `.all()` function has been applied.

Having a choice of performing the same task faster allows us to look at the mathematical functions—differently. In [Code 3.12](#) we defined a vector of arguments, `x`, as since the values of $f(x)$ are sought after, the vectorised operations can be applied, element-wise.

Plotting of the resultant NumPy array of `f3` takes only a few line of code and delivers:



The hunt for speed in Python continues. The projects like PyPy (<http://pypy.org>), Cython (<http://cython.org>), Continuum Analytics' Numba (<http://numba.pydata.org>) or NumbaPro (<http://docs.continuum.io/numbapro/index>) address independent solutions related to the code manipulation, compilation, and accelerated execution. You are more than welcome to explore them all. However, the first starting point you should get familiarised with is—the NumPy's ufuncs.

3.7.2. Mathematics of ufuncs

Intuitively, the operations on matrixes should follow the rules of matrix algebra. Therefore, the basic **arithmetics** applied to two row vectors returns:

```
>>> a = np.array([-1, -2, -3, -4, 5])
>>> b = np.array([5, 4, 3, 2, 1])

>>> a + 1
array([ 0, -1, -2, -3, -4])

>>> a - b
array([-6, -6, -6, -6, -6])

>>> b*2
array([10,  8,  6,  4,  2])

>>> b/a
array([-5. , -2. , -1. , -0.5, -0.2])

>>> b**3 - a/2.1
array([ 125.47619048,  64.95238095,  28.42857143,  9.9047619 ,
        3.38095238])

>>> (a//4) % 3
array([2, 2, 2, 2, 1])
```

As we can see, all operations are element-wise and work in the same fashion if 2D arrays are considered. What you do not see here is so-called **broadcasting**, i.e. the method how certain operations are conducted. We will describe it in *Volume II*.

The application of **ufuncs** is somehow natural as we have witnessed it in the introductory example. Focus and analyse what follows:

```
>>> a = np.array([[3, 4, 5], [6, 7, 8]])
>>> b = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]])
>>> a
array([[3, 4, 5],
       [6, 7, 8]])
>>> b
array([[ 0.1,  0.2,  0.3],
       [ 0.4,  0.5,  0.6]])
```

next:

```
>>> a + b
array([[ 3.1,  4.2,  5.3],
       [ 6.4,  7.5,  8.6]])
>>> a - b
array([[ 2.9,  3.8,  4.7],
       [ 5.6,  6.5,  7.4]])
```

and

```
>>> np.abs(a - b)
array([[ 2.9,  3.8,  4.7],
       [ 5.6,  6.5,  7.4]])

>>> np.exp2(a)
array([[ 8.,  16.,  32.],
       [ 64., 128., 256.]])
>>> (np.exp2(a) == np.power(2, a)).all()
True

>>> np.log2(b/2+a)
array([[ 1.60880924,  2.03562391,  2.36457243],
       [ 2.63226822,  2.857981  ,  3.05311134]])
```

A bit of trigonometry required?

```
>>> x = np.arange(-np.pi, np.pi+np.pi/4, np.pi/4)
array([-3.14159265, -2.35619449, -1.57079633, -0.78539816,  0.
        0.78539816,  1.57079633,  2.35619449,  3.14159265])

>>> np.degrees(x)
array([-180., -135., -90., -45.,  0.,  45.,  90., 135.,
        180.])
>>> np.radians(np.degrees(x))
array([-3.14159265, -2.35619449, -1.57079633, -0.78539816,  0.
        0.78539816,  1.57079633,  2.35619449,  3.14159265])

>>> np.cos(x)**2 + np.sin(x)**2
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])

>>> np.log1p(1+np.cos(x)**3)
array([ 0.
        0.4986194 ,  0.69314718,  0.85592627,  1.09861229,
        0.85592627,  0.69314718,  0.4986194 ,  0.
        ])

>>> np.arccos(np.cos(b))
array([[ 0.1,  0.2,  0.3],
       [ 0.4,  0.5,  0.6]])
```

The following table summarises the most important mathematical ufuncs and operations available to you in NumPy 1.10.1:

NumPy's Universal Functions	Description (element-wise ops)
<code>abs(x)</code> <code>absolute(x)</code>	an absolute value of x an absolute value of x
<code>power(x, y)</code> <code>sqrt(x)</code> <code>reciprocal(x)</code> <code>exp(x)</code> <code>exp2(x)</code> <code>expm1(x)</code> <code>log(x)</code> <code>log2(x)</code> <code>log10(x)</code> <code>log1p(x)</code> <code>sign(x)</code>	the power $x^{**}y$ a square root of x a reciprocal of x $e^{**}x$ $2^{**}x$ $e^{**}x-1$ a natural logarithm a logarithm with base of 2 a logarithm with base of 10 $\log(1+x)$ an array of signs (0 remains 0)
<code>mod(x, y)</code> <code>remainder(x, y)</code>	x modulo y a remainder from division x by y
<code>sin(x)</code> <code>cos(x)</code> <code>tan(x)</code> <code>arcsin(x)</code> <code>arccos(x)</code> <code>arctan(x)</code> <code>arctan2(x, y)</code> <code>sinh(x)</code> <code>cosh(x)</code> <code>tanh(x)</code> <code>arcsinh(x)</code> <code>arccosh(x)</code> <code>arctanh(x)</code> <code>deg2rad(x)</code> <code>rad2deg(x)</code>	a sine of x a cosine of x a tangent of x an arcsine of x an arccosine of x an arctangent of x an arctangent of x/y a hyperbolic sine of x a hyperbolic cosine of x a hyperbolic tangent of x an inverse hyperbolic sine an inverse hyperbolic cosine an inverse hyperbolic tangent $\text{np.radians}(x) == \text{np.deg2rad}(x)$ $\text{np.degrees}(x) == \text{np.rad2deg}(x)$

A full list of universal functions is available at <http://docs.scipy.org/doc/numpy-1.10.1/reference/ufuncs.html#available-ufuncs>.

That page also lists some useful **floating functions** that, very often, one can apply to obtain additional information on the arrays, e.g.:

```
>>> b = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]])
>>> np.iscomplex(b).all()
False
>>> b != 0
>>> np.isinf(b).all()
True
```

or

Floating Functions	Description (element-wise ops)
<code>isreal(x)</code> <code>iscomplex(x)</code> <code>isfinite(x)</code> <code>isinf(x)</code> <code>floor(x)</code> <code>ceil(x)</code> <code>trunc(x)</code>	do you deal with real numbers? do you have complex numbers in x? any infs inside? if so, say True the floor of x the ceiling of x neglect the rest <pre>>>> np.trunc(np.array([np.pi])) array([3.])</pre>
<code>copysign(y, x)</code>	copy the signs from y to x <pre>>>> np.copysign(np.array([-1, 4]), np.array([-1, -1])) array([-1., -4.])</pre>

3.7.3. Algebraic Operations

Algebra. A gateway to the Master degree in a quantitative field. At some point in time, you start to understand its importance. From the basic 1D and 2D array processing to the complex operations on data flowing through multiple channels. You cannot skip it. It's somehow superbly essential if you think about a serious number crunching.

Let's fly over the key concepts every quant will find most useful.

Matrix Transpositions, Addition, Subtraction

Say,

```
>>> X = np.array([[1, 3, 3], [2, -1, 9], [4, 5, 0]])
>>> X
array([[ 1,  3,  3],
       [ 2, -1,  9],
       [ 4,  5,  0]])
>>> Y = -X[2:].T.copy() + 1
>>> Y
array([[ -3],
       [ -4],
       [  1]])
```

then the operations of **addition/subtraction**, matrix **transposition**, **Hermitian transposition** (i.e., with complex conjugate) would look like:

```
>>> X + 2*X                                # alternatively X - 2*X
array([[ 3,  9,  9],
       [ 6, -3, 27],
       [12, 15,  0]])

>>> Y.T
array([[ -3, -4,  1]])

# create a random complex 2D array
>>> complex_X = x + 1j*np.random.randint(-1, 2, size=(3, 3))
array([[ 1.-1.j,  3.-1.j,  3.+1.j],
       [ 2.+0.j, -1.-1.j,  9.+1.j],
       [ 4.-1.j,  5.-1.j,  0.+0.j]])
>>> np.array(np.matrix(complex_X).getH()) # Hermitian transpose
array([[ 1.+1.j,  2.-0.j,  4.+1.j],
       [ 3.+1.j, -1.+1.j,  5.+1.j],
       [ 3.-1.j,  9.-1.j,  0.-0.j]])
```

where to get the Hermitian transposition we had to convert a NumPy array into a NumPy *matrix object* (omitted in the book), to apply a function responsible for transposition, and to convert the result back to the NumPy array.

Please also note on the quickest method (used in this example) of creating a **random complex 2D array** being, in fact, a mixture of real and complex numbers, all in one.

Matrix Multiplications

This one is tricky. And if you have studied algebra in the past, you know what I mean! That is why let's consider the following. As previously, we have two arrays:

```
>>> x = np.array([[1, 3, 3], [2, -1, 9], [4, 5, 0]]); x
array([[ 1,  3,  3],
       [ 2, -1,  9],
       [ 4,  5,  0]])
>>> y = -x[2:].T.copy() + 1; y
array([[ -3],
       [ -4],
       [  1]])
```

then an **element-wise** matrix multiplication returns:

```
>>> x*x
array([[ 1,  9,  9],
       [ 4,  1, 81],
       [16, 25,  0]])
>>> np.abs(y*y*y)
array([[27],
       [64],
       [ 1]])
```

The **inner product** we obtain with a help of:

```
np.inner >>> np.inner(x, -x)
array([[ -19, -26, -19],
       [-26, -86,  -3],
       [-19,  -3, -41]])
```

and it is not the same what

```
>>> x*(-x)
array([[ -1,  -9,  -9],
       [ -4,  -1, -81],
       [-16, -25,  0]])
```

The most **classical** algebraic 2D **matrix multiplication** is the one where the requirement for specific dimensions must be met: the number of columns in a first array must be equal the number of rows in a second array:

```
np.dot >>> np.dot(x, y)
array([[ -12],
       [  7],
       [-32]])
```

where an additional NumPy function of:

```
np.vdot >>> np.vdot(y, 2*y)
52
```

returns a dot product of two vectors. Please also note of how the wrong input generates an error:

```
>>> np.dot(x, y.T)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (3,3) and (1,3) not aligned: 3 (dim 1) != 1 (dim 0)
```

@ Operator, Matrix Inverse, Multiple Linear Regression

One of the newest feature of Python 3.5 is the acceptance of the PEP 465 proposal related to the introduction of a dedicated infix operator for matrix multiplication (source: <https://www.python.org/dev/peps/pep-0465/>). The author of the proposal, Mr. Nathaniel J. Smith, underlined the fact that there were two important operations which competed for use of Python's `*` operator: the element-wise multiplication, and matrix multiplication.

Most numerical Python codes use the operator for element-wise multiplication, and function/method syntax for matrix multiplication. Nathaniel pointed that it usually led to ugly and unreadable code and misinterpretation.

In February 2014 he proposed that the minimal change to Python syntax which would be sufficient to resolve a lot of problems was a brand new infix operator for **matrix multiplication**: a `@` operator.

Recalling the previous subsection:

```
>>> np.dot(x, y)
array([[ -12],
       [  7],
       [-32]])
```

is now possible by

`@` infix operator

```
>>> x @ y
array([[ -12],
       [  7],
       [-32]])
```

The use of `@` operator is restricted to arrays (matrixes) therefore, a multiplication by scalar will return an error:

```
>>> x @ 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Scalar operands are not allowed, use '*' instead
```

though

```
>>> y @ (4*y.T)
array([[ 36,  48, -12],
       [ 48,  64, -16],
       [-12, -16,  4]])
```

works.

The main benefit of a new operator is to **reduce the complexity** of notation. Let's study the following case study in order to compare the simplicity of performing matrix dot products, now with `@` when both the length of notation extends itself and the clarity of Python code would be a nice thing to have.

Based on the input data, find the solution for a multi-factor model (an equation of a hyperplane of regression) for Y based on examination of X_1, X_2, X_3 .

Within this challenge we may think of the **multi-factor model** that tries to estimate the return of a security, R , based on 3 factors we measured independently (1 = the factor had an influence, 0 = it did not), expressed as:

$$Y = \beta_1 X_1 + \beta_2 X_2 + \beta_3 f_3 + \beta_M + \epsilon$$

where the last term is the error term standing before *beta* for the market. The above equation represents a subset of the general formulation of the problem in the form:

$$Y = a_1 X_1 + a_2 X_2 + \dots + a_k X_k + a_{k+1} + \epsilon$$

All X 's are assumed to be random variables and all a_k are known as the regression coefficients. The latter, in the framework of a **multiple linear regression**, one estimates using the least square methods. Employing the matrix notation, we can express our model as:

$$\mathbf{Y} = \mathbf{X}\mathbf{a} + \epsilon$$

where there is a broadly accepted approach to find the unbiased estimator of \mathbf{a} based on \mathbf{X} sample, namely:

$$\mathbf{A} = \begin{pmatrix} A_1 \\ \vdots \\ A_{k+1} \end{pmatrix} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y}$$

Therefore, we go for A_k ($k = 1, \dots, k+1$) coefficients. Notice the complexity of matrix operations required for finding \mathbf{A} . This is where the beauty of a new @ operator enters the area in a full spotlight. Let's start with a random sample of \mathbf{X} observations and we head for \mathbf{A} .

Let's say we observe:

i	1	2	3	4	5	6	7	8
X_1	0	1	0	0	1	1	0	1
X_2	0	0	1	0	1	0	1	1
X_3	0	0	0	1	0	1	1	1
Y	1	2	6	2	7	3	3	2

then all intermediate steps can be derived:

```
>>> X = np.array([[0, 0, 0, 1], [1, 0, 0, 1], [0, 1, 0, 1],
                  [0, 0, 1, 1], [1, 1, 0, 1], [1, 0, 1, 1],
                  [0, 1, 1, 1], [1, 1, 1, 1]]); X
array([[0, 0, 0, 1],
       [1, 0, 0, 1],
       [0, 1, 0, 1],
       [0, 0, 1, 1],
       [1, 1, 0, 1],
       [1, 0, 1, 1],
       [0, 1, 1, 1],
       [1, 1, 1, 1]])

>>> Y = np.array([1,2,6,2,7,3,3,2]).T; Y
array([1, 2, 6, 2, 7, 3, 3, 2])

>>> np.dot(X.T, X)
array([[4, 2, 2, 4],
       [2, 4, 2, 4],
       [2, 2, 4, 4],
       [4, 4, 4, 8]])
```

where the last operation is:

$$\mathbf{X}^T \mathbf{X}$$

Now, in order to be sure we can invert the matrix we have to ensure that its determinant:

$$\det(\mathbf{X}^T \mathbf{X}) \neq 0$$

therefore:

```
np.linalg.det      >>> np.linalg.det(np.dot(X.T, X))
64
```

and we are good to go for the **matrix inverse**:

```
np.linalg.inv      >>> np.linalg.inv(np.dot(X.T, X))
array([[ 5.00000000e-01,  0.00000000e+00,  0.00000000e+00,
        -2.50000000e-01],
       [ 0.00000000e+00,  5.00000000e-01,  0.00000000e+00,
        -2.50000000e-01],
       [ 0.00000000e+00,  2.08166817e-17,  5.00000000e-01,
        -2.50000000e-01],
       [-2.50000000e-01, -2.50000000e-01, -2.50000000e-01,
         5.00000000e-01]])
```

Next, we derive:

$$\mathbf{X}^T \mathbf{Y}$$

as follows:

```
>>> np.dot(X.T, Y)
array([14, 18, 10, 26])
```

Eventually, we may apply:

$$\mathbf{A} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y}$$

to find that:

```
>>> A = np.dot(np.linalg.inv(np.dot(X.T, X)), np.dot(X.T, Y))
array([ 0.5,  2.5, -1.5,  2.5])
```

what solves our problem and delivers the regression line:

$$y = 0.5x_1 + 2.5x_2 - 1.5x_3 + 2.5$$

terminating all calculations.

Could you look again at that creature:

```
A = np.dot(np.linalg.inv(np.dot(X.T, X)), np.dot(X.T, Y))
```

Does it look sexy? Nah... Can we shorten it? Of course! Let's call our new friend, the infix `@` operator in Python 3.5, and rewrite **A** in a more **compact** way:

```
>>> from numpy.linalg import inv
>>> A = inv(X.T @ X) @ X.T @ Y
>>> A
array([ 0.5,  2.5, -1.5,  2.5])
```

Now we are talking... ☺

Linear Equations

The NumPy's `linalg` (linear algebra) submodule offers, as one might expect, a ready-to-use function for solving a linear matrix equation or system of linear scalar equations. By splitting all coefficients of:

$$-5x_1 + 3x_2 + 4x_3 = -18$$

$$3x_1 - 2x_2 + x_3 = -7$$

$$6x_1 + 3x_2 + 6x_3 = 27$$

into two separate matrixes of **a** and **b** (see the Code 3.13 below), we derive:

Code 3.13

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

a = np.array([[ -5,  3,  4],
               [  3, -2,  1],
               [  6,  3,  6]])
b = np.array([-18, -7, 27])
```

`np.linalg.solve`

```
x = np.linalg.solve(a, b)

print("det(a) = %.2f" % np.linalg.det(a))
for i, e in enumerate(x):
    print("x%g\t= %.2f" % (i+1, e))
```

id est,

```

det(a) = 123.00
x1      = 4.95
x2      = 8.56
x3      = -4.73

```

In this case, the determinant is non-zero therefore the solution exists and one can interpret it as: for the planes (defined by equations above) there is one common point in 3D space (x_1, x_2, x_3) where they cross each other.

An attempt in visualisation of all three planes one can achieve by finding the normal (p_1, p_2, p_3) such that: if a is not 0 a point on the plane is $(-d/a, 0, 0)$; if b is not 0 a point on the plane is $(0, -d/b, 0)$; if c is not 0 a point on the plane is $(0, 0, -d/c)$; where

$$ax_1 + bx_2 + cx_3 + d = 0$$

denotes a general form of the plane equations given in the linear system above. For each of them we find the normal vector and the point on the plane.

In Python, we execute it by extending [Code 3.13](#):

```

n1 = a[0, :]
n2 = a[1, :]
n3 = a[2, :]

try:
    point1 = np.array([-b[0]/n1[0], 0, 0])
except:
    point1 = np.array([0, 0, 0])
try:
    point2 = np.array([0, -b[1]/n2[1], 0])
except:
    point2 = np.array([0, 0, 0])
try:
    point3 = np.array([0, 0, -b[2]/n3[2]])
except:
    point3 = np.array([0, 0, 0])

d1 = -1 * np.sum(point1*n1)
d2 = -1 * np.sum(point2*n2)
d3 = -1 * np.sum(point3*n3)

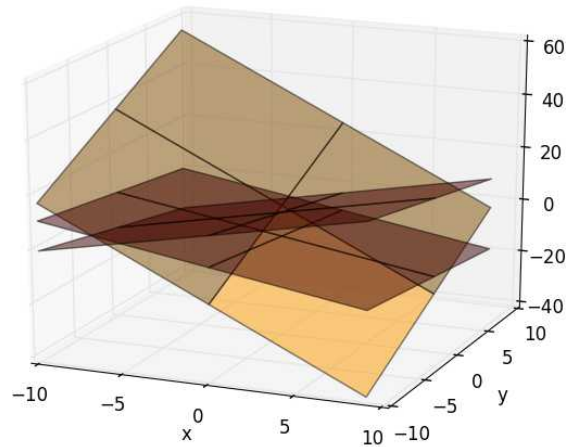
# create a mesh grid
xx, yy = np.meshgrid(range(-10, 10), range(-10, 10))

z1 = (-n1[0]*xx - n1[1]*yy - d1)*1./n1[2]
z2 = (-n2[0]*xx - n2[1]*yy - d2)*1./n2[2]
z3 = (-n3[0]*xx - n3[1]*yy - d3)*1./n3[2]

# plot the surfaces
plt3d = plt.figure().gca(projection='3d')
cmap = plt.get_cmap('jet')
plt3d.plot_surface(xx, yy, z1, alpha=0.5, color=cmap(1200))
plt3d.plot_surface(xx, yy, z2, alpha=0.5, color=cmap(190))
plt3d.plot_surface(xx, yy, z3, alpha=0.5, color=cmap(550))
plt.xlabel("x"); plt.ylabel("y")
plt.show()

```

what brings us to:



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 ($N \times L$). 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: $\mathbf{B}\mathbf{X} = \mathbf{X}$ 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 $\mathbf{P}\mathbf{X} = \mathbf{X}$. 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} .

The goal of PCA is to find such \mathbf{P} where $\mathbf{Y} = \mathbf{P}\mathbf{X}$ such that:

$$\text{cov}(\mathbf{Y}) = (N - 1)^{-1} \mathbf{X}\mathbf{X}^T$$

is diagonalised, and $\text{cov}(\mathbf{Y})$ denotes the covariance matrix. Finding principal components of \mathbf{X} is as simple as a computation of eigenvectors of $\text{cov}(\mathbf{Y})$ matrix where the row vectors' values of \mathbf{Y} ought to be $\sim N(0, 1)$.

For a full explanation of the Principal Component Analysis' algebra I encourage you to study <http://www.quantatrisk.com/2013/10/16/anxiety-detection-model-for-stock-traders-based-on-principal-component-analysis/>. In the following case study, we will show how to find PCs for N -Asset Portfolio.

Code 3.14 PCA for N -Asset Portfolio

In this exercise we won't use real-market data. Instead **we will learn** how to use NumPy arrays to **simulate** close price time-series for N assets (e.g. stocks) spanned over $L+1$ days based on a given $(N \times L)$ matrix of daily returns. In order to help you out with the visualisation of this process we start from a naive position of:

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(2014)

# define portfolio
N = 5 # a number of assets
L = 4 # a number of days

# asset close prices
p = np.random.randint(10, 30, size=(N, 1)) + \
    np.random.randn(N, 1) # a mixture of uniform and N(0,1) rvs

print(p)
print(p.shape)
print()
```

where we specify a number of assets in portfolio and a number of days. $L = 4$ has been selected for the clarity of printing of the outcomes below, however, feel free to increase that number (or both) anytime you rerun this code.

Next, we create a matrix $(N \times 1)$ with a starting random prices for all N assets to be between \$10 and \$30 (random integer) supplemented by $(0, 1)$ fractional part. Printing \mathbf{p} returns:

```
[[ 25.86301396]
 [ 19.82072772]
 [ 22.33569347]
 [ 21.38584671]
 [ 24.56983489]]
(5, 1)
```

Now, let's generate a matrix of random daily returns over next 4 days for all 5 assets:


```

r = np.random.randn(N, L)/50 # ~ N(0,1)
print(r)
print(r.shape)

```

delivering:

```

[[ 0.01680965 -0.00620443 -0.02876535 -0.03946471]
 [-0.00467748 -0.0013034  0.02112921  0.01095789]
 [-0.01868982 -0.01764086  0.01275301  0.00858922]
 [ 0.01287237 -0.00137129 -0.0135271  0.0080953 ]
 [-0.00615219 -0.03538243  0.01031361  0.00642684]]
(5, 4)

```

Having that, our wish is, for each asset, take its first close-price value from **p** array and using information on daily returns stored row-by-row (i.e. asset per asset) in **r** array, reconstruct close-price asset time-series:

```

for i in range(r.shape[0]):
    tmp = []
    for j in range(r.shape[1]):
        if(j == 0):
            tmp.append(p[i][0].tolist())
            y = p[i] * (1 + r[i][j])
        else:
            y = y * (1 + r[i][j])
            tmp.append(y.tolist()[0])

    if(i == 0):
        P = np.array(tmp)
    else:
        P = np.vstack([P, np.array(tmp)])

print(P)
print(P.shape)

```

That returns:

```

[[ 25.86301396  26.29776209  26.13459959  25.38282873  24.38110263]
 [ 19.82072772  19.72801677  19.70230331  20.11859739  20.33905475]
 [ 22.33569347  21.91824338  21.53158666  21.8061791  21.99347727]
 [ 21.38584671  21.66113324  21.63142965  21.33881915  21.51156325]
 [ 24.56983489  24.41867671  23.55468454  23.79761839  23.95056194]]
(5, 5)

```

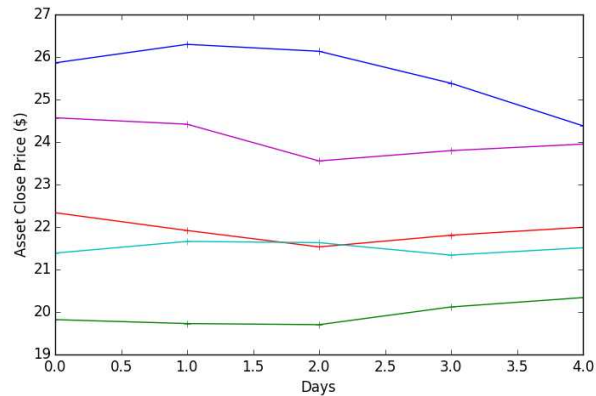
The operations applied within the loops may look a bit complex but they are based on what we have learnt so far in this book. ☺

Thus, we have two loops: the outer one over rows/assets (index i) and inner one over columns/days (index j). For $j = 0$ we copy the price of the asset from **p** as a "starting close price", e.g. on the first day. Concurrently, using the first information from **r** matrix we compute a change in price on the next day. In **tmp** list we store (per asset) the history of close price changes over all $L+1$ days. These operations are based on a simple list processing. Finally, having a complete information on i -th asset and its price changes after $r.shape[1] + 1$ days, we build a new array of **P** with an aid of `np.vstack` function (see Section 3.3.4). Therefore **P** stores the simulated close-price time-series for N assets.

We can display them by adding to 3.14:

```
plt.figure(num=1, figsize=(8, 5))
plt.plot(P.T, '+-')
plt.xlabel("Days")
plt.ylabel("Asset Close Price (\$)")
plt.show()
```

what reveals:



where the transposition of **P** for plotting has been applied to deliver asset-by-asset price-series (try to plot the array without it and see what happens and understand why it is so).

Recalling that **P** has been found as:

```
[ [ 25.86301396  26.29776209  26.13459959  25.38282873  24.38110263 ]
  [ 19.82072772  19.72801677  19.70230331  20.11859739  20.33905475 ]
  [ 22.33569347  21.91824338  21.53158666  21.8061791  21.99347727 ]
  [ 21.38584671  21.66113324  21.63142965  21.33881915  21.51156325 ]
  [ 24.56983489  24.41867671  23.55468454  23.79761839  23.95056194 ]
(5, 5)]
```

the computation of the mean values, row-by-row, takes:

```
m = np.mean(P, axis=1)

[ 25.6118614,  19.94173999,  21.91703598,  21.5057584,  24.0582753 ]
```

what should be understood more like this (row-wise):

```
25.6118614
19.94173999
21.91703598
21.5057584
24.0582753
```

The same applies to the standard deviation row-wise:

```
s = np.std(P, axis=1) # number of d.o.f = 0
```

In next step, we aim at subtraction of the mean value (as derived per asset) from **P** and its normalisation by the corresponding standard deviations. This is a requirement for PCA. Therefore, by adding:

```
m = np.mean(P, axis=1) + np.zeros((N, L+1))
m = m.T
s = np.std(P, axis=1) + np.zeros((N, L+1))
s = s.T
```

```

print(m)
print()
print(s)
print()

# normalised P array
normP = (P-m)/s
print(normP)

```

to 3.14, we come up with a desired output:

```

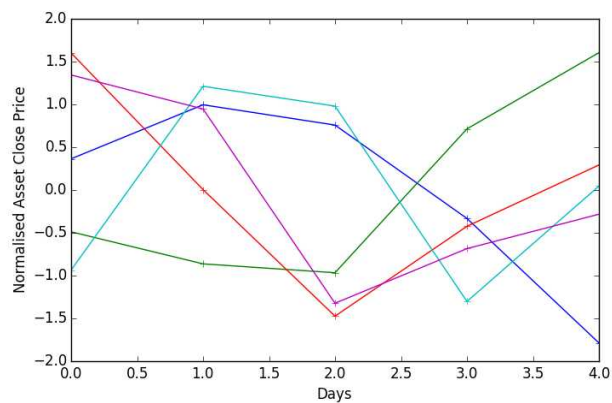
# m
[[ 25.6118614  25.6118614  25.6118614  25.6118614  25.6118614 ]
 [ 19.94173999 19.94173999 19.94173999 19.94173999 19.94173999]
 [ 21.91703598 21.91703598 21.91703598 21.91703598 21.91703598]
 [ 21.5057584  21.5057584  21.5057584  21.5057584  21.5057584 ]
 [ 24.0582753  24.0582753  24.0582753  24.0582753  24.0582753 ]]

# s
[[ 0.6890596  0.6890596  0.6890596  0.6890596  0.6890596 ]
 [ 0.24770509 0.24770509 0.24770509 0.24770509 0.24770509]
 [ 0.261526  0.261526  0.261526  0.261526  0.261526 ]
 [ 0.12823091 0.12823091 0.12823091 0.12823091 0.12823091]
 [ 0.38071781 0.38071781 0.38071781 0.38071781 0.38071781]]

# normP
[[ 0.36448598  0.99541561  0.7586255  -0.33238441 -1.78614268]
 [-0.48853361 -0.86281319 -0.96661993  0.71398373  1.603983 ]
 [ 1.60082551  0.00461675 -1.473847  -0.42388471  0.29228945]
 [-0.93512318  1.21168011  0.98003868 -1.30186438  0.04526877]
 [ 1.34367132  0.94663662 -1.32274021 -0.68464595 -0.28292178]]

```

Displaying the normalised close-price time-series,



we get thanks to:

```

# normalised P array
plt.figure(num=2, figsize=(8, 5))
plt.plot(normP.T, '+-')
plt.xlabel("Days")
plt.ylabel("Normalised Asset Close Price")
plt.show()

```

From this point, the computation of the **covariance** matrix for our normalised N -asset price-series set requires:

```

np.cov      c = np.cov(normP)
print(c)

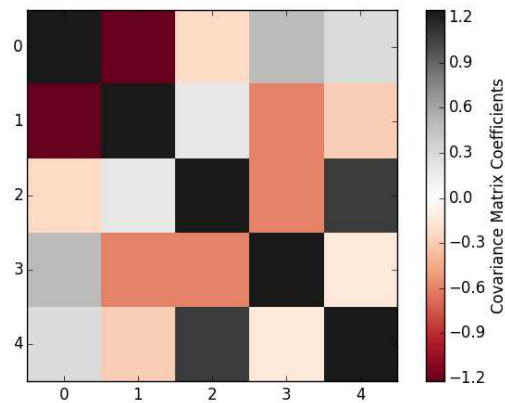
```

```
plt.figure(num=3)
plt.imshow(c, cmap="RdGy", interpolation="nearest")
cb = plt.colorbar()
cb.set_label('Covariance Matrix Coefficients')
```

which displays both the covariance matrix coefficients,

```
[[ 1.25      -1.21812086 -0.22780047  0.49015782  0.29037152]
 [-1.21812086  1.25      0.20119748 -0.59820976 -0.28430998]
 [-0.22780047  0.20119748  1.25      -0.59268253  1.07809658]
 [ 0.49015782 -0.59820976 -0.59268253  1.25      -0.13182634]
 [ 0.29037152 -0.28430998  1.07809658 -0.13182634  1.25      ]]
```

and their visualisation,



where the diagonal elements represent the variances.

A derivation of longly awaited **eigenvalues** and **eigenvectors** for the covariance matrix takes:

```
np.linalg.eig      w, v = np.linalg.eig(c)

print(w); print()
print(v)
print(v.shape)
```

where the `np.linalg.eig` function returns a tuple

```
[ 3.01079265e+00  2.43631310e+00  7.40910779e-01  6.57724704e-17
  6.19834710e-02]
```

of eigenvalues (**w** array) and

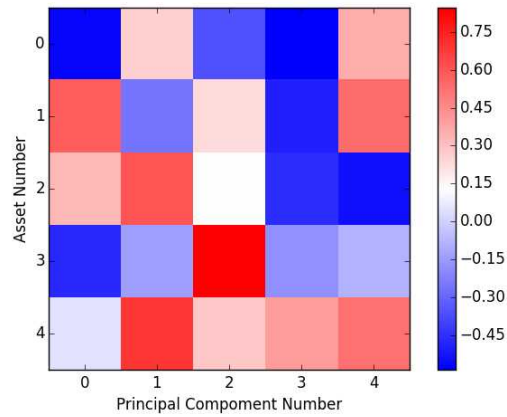
```
[[ -0.57032118  0.25997416 -0.3637151  -0.58784   0.35958678]
 [ 0.58459138 -0.25856444  0.23234816 -0.49389736  0.54173525]
 [ 0.32889202  0.60571277  0.12841265 -0.46259987 -0.54263582]
 [ -0.47168761 -0.14169051  0.84636404 -0.18259553 -0.08809874]
 [ 0.0482442   0.69180467  0.28443588  0.40394278  0.52440943]]
(5, 5)
```

normalised (unit "length") eigenvectors (**v** array); the latter possible to be visualised by:

```
plt.figure(num=4, figsize=(8, 5))
plt.imshow(v, cmap="bwr", interpolation="nearest")
cb = plt.colorbar()
plt.ylabel("Asset Number")
```

```
plt.xlabel("Principal Component Number")
plt.show()
```

in the form of:



Summarising, we computed PCA for five principal components in order to illustrate the process. If all simulated assets time-series behaved in the same fashion (move up or down over all 5 days in a similar direction), we would expect to notice (at least) the first PC to display the same (or close) values. This is not the case for our randomly generated asset price-series. Interestingly, the forth PC is more coherent but, again, its interpretation remains meaningless due to randomness applied.

3.8. Element-wise Analysis

Working with the data processing utilising raw NumPy arrays would not be so appealing without flexible and smartly designed **boolean operations**. As we have seen so far, the ability to scan, search, and filter the array for desired information is a natural need at some stage of research. The boolean array is generated and all operations are element-wise. NumPy's capabilities do not end here. Going one step further, we have an ability to apply a mask to extract only a portion of information out of a whole array.

Within this Section we will try to summarise the essentials of all boolean (logical) operations available in NumPy by approaching the subject from the most practical point of view.

Imagine that the following (random) 3x5 matrix represents your data set. You may look at it as an image where the numbers represents the pixel's residual values, or as a collection of 3 time-series 5 points long:

```
import numpy as np

np.random.seed(28)

x = np.random.randint(0, 12, size=(3, 5)) * \
    np.random.randint(-1, 2, size=(3, 5))

print(x)

[[-1 -9  0  6  4]
 [ 0 -3  7  0 -3]
 [-7  0  8 -8 -2]]

y = x[2, :] + 1
print(y)

[-6  1  9 -7 -1]
```

where **y** is a 1D row vector (e.g., an individual time-series).

3.8.1. Searching

The most intuitive operation of finding out what elements of the array meet a specified condition can be done by, e.g.

```
print(x > 4)  # greater than 4

[[False False False  True False]
 [False False  True False False]
 [False False  True False False]]

print(x == 0)  # equal 0

[[False False  True False False]
 [ True False False  True False]
 [False  True False False False]]
```

```
print(y != 0) # not equal
[ True  True  True  True  True]
```

What appears to be intuitive here, in fact, for NumPy is a ufunc (universal function) performing an element-wise boolean operation. Therefore,

```
print(y >= 0)
print(np.greater_equal(y, 0))
```

are two equivalent methods in checking which element of an array is greater or equal zero. The use of a `>=` operator is made for a compact and understandable notation. Similarly:

ufunc	Operator
<code>np.equal</code>	<code>==</code>
<code>np.not_equal</code>	<code>!=</code>
<code>np.greater</code>	<code>></code>
<code>np.greater_equal</code>	<code>>=</code>
<code>np.less</code>	<code><</code>
<code>np.less_equal</code>	<code><=</code>

In order to construct a bit more complex condition, we have to use a special case of the boolean operators: `|` (or) and `&` (and). Analyse:

```
print((x < 4) & (x >=0)) # both conditions must be met

[[False False  True False False]
 [ True False False  True False]
 [False  True False False False]]

print((x < -1) | (x == 0))

[[False  True  True False False]
 [ True  True False  True  True]
 [ True  True False  True  True]]

print( (y < 0) & ((y != -6) | (y != -7)))

[ True False False  True  True]
```

where a **negation** of the last expression we achieve thanks to the `~` operator applied as follows:

```
print( ~(y < 0) & ((y != -6) | (y != -7)))

[False  True  True False False]
```

It is also possible to combine two (or more) arrays and formulate your query such that it performs an element-wise comparison among those arrays. Say, we define:

```
z = np.ones((3, 5))
print(z)

[[ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
```


then

```
print( (x > 0) | (z > 0) )

[[ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]]

print( (x > 0) & (z > 0) )

[[False False False  True  True]
 [False False  True False False]
 [False False  True False False]]
```

what makes a perfect sense.

3.8.2. Searching, Replacing, Filtering

Recalling what we have already learnt on searching in NumPy, we may apply similar constructions inside the `np.where` function. However, the output we will see will be in a form of the array(s) storing the indexes corresponding to our boolean condition, if satisfied. Compare:

```
print(y)
print(y >= -1)
print(np.where(y >= -1))

[-6  1  9 -7 -1]
[False  True  True False  True]
(array([1, 2, 4]),)
```

and

```
Also try:
>>> i, j = np.where(x == 0)
>>> print(i)
[0 1 1 2]
>>> print(j)
[2 0 3 1]
>>> print(type(i))
<class 'numpy.ndarray'>

print(x)
print(~(x != 0)) # another way to say that something is equal zero
print(np.where(~(x != 0)))

[[-1 -9  0  6  4]
 [ 0 -3  7  0 -3]
 [-7  0  8 -8 -2]]

[[False False  True False False]
 [ True False False  True False]
 [False  True False False False]]

(array([0, 1, 1, 2]), array([2, 0, 3, 1]))
```

where the `np.where` function applied to `x` array returns 2D index-coordinates,

```
0, 2 # row, column
1, 0
1, 3
2, 1
```

of the array's cells for which the values are equal zero. Having the coordinates, we may pick all or only a few elements and change their values manually. In some cases this method may be very handy.

Alternatively, as we already know, `np.where` function defined in the following way:

```
print(x)
print(np.where((x > 0), 0, x))

[[-1 -9  0  6  4]
 [ 0 -3  7  0 -3]
 [-7  0  8 -8 -2]]

[[-1 -9  0  0  0]
 [ 0 -3  0  0 -3]
 [-7  0  0 -8 -2]]
```

should be read: if any element in **x** is greater than 0, **replace** it with 0 otherwise leave it as it is. It is possible to change the position of arguments, i.e:

```
print(np.where((x > 0), x, 0))
print(np.where((x < 0), 0, x))

[[0 0 0 6 4]
 [0 0 7 0 0]
 [0 0 8 0 0]]

[[0 0 0 6 4]
 [0 0 7 0 0]
 [0 0 8 0 0]]
```

where both callings of the function of `np.where` are equivalent.

An advanced replacement across two arrays works well too:

```
z[1, 1] = -5
print(x)
print(z)
print(np.where((z < 0) & (x < 0), x, 0))

[[-1 -9  0  6  4]
 [ 0 -3  7  0 -3]
 [-7  0  8 -8 -2]]

[[ 1.  1.  1.  1.  1.]
 [ 1. -5.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]]

[[ 0  0  0  0  0]
 [ 0 -3  0  0  0]
 [ 0  0  0  0  0]]
```

In this case, first, we alter **z** array a bit, next we ask a question: if any element in **z** is *less than 0* and any element in **x** is *less than 0* then print the corresponding elements of **x** otherwise zeros. Therefore, we have applied a sort of **filter** based on information (element-wise) taken from two independent arrays of the same shape. Similarly,

```
print(np.where((z < 0) & (x < 0), 0, 7)) # otherwise print 7

[[7 7 7 7 7]
 [7 0 7 7 7]
 [7 7 7 7 7]]
```

produces another output. Reasonably easy, don't you think?

3.8.3. Masking

The boolean result can be directly applied in order to extract (i.e. to apply a **mask**) those elements that meet a given condition, e.g.:

```
print(y)
print( y[y <= -2] )

[ -6  1  9 -7 -1]
[ -6 -7]
```

where we get an array with elements in **y** that are *less than or equal* -2. Also,

```
print(x)
print( x[x > 1] )
print( x[(x > 1) | (x < -1)] )

[[-1 -9  0  6  4]
 [ 0 -3  7  0 -3]
 [-7  0  8 -8 -2]]

[6 4 7 8]

[-9  6  4 -3  7 -3 -7  8 -8 -2]
```

and, if required to be **sorted**, then:

```
np.sort print(np.sort(x[(x > 1) | (x < -1)])) # ascending order

[-9 -8 -7 -3 -3 -2  4  6  7  8]
```

or

```
print(-np.sort(-x[(x > 1) | (x < -1)])) # descending order

[ 8  7  6  4 -2 -3 -3 -7 -8 -9]
```

where the function of `np.sort` returns a sorted copy of an array. More on sorting in *Volume II*.

3.8.4. Any, if Any, How Many, or All?

It is often the case when we are interested in counting **non-zero** or **zero elements** in an array. One can do it with a help of the `np.where` and `len` functions. However, NumPy gives us a hand in performing this task in a quicker way:

```
np.count_nonzero print(x)
print(np.count_nonzero(x)) # a number of non-zero elements
print(x.size - np.count_nonzero(x)) # a number of zeros

[[-1 -9  0  6  4]
 [ 0 -3  7  0 -3]
 [-7  0  8 -8 -2]]

11
4
```

Counting can be performed for a particular interval. For example, if we generate an array with a large number of random variables drawn from a Normal distribution, $rvs \sim N(0, 1)$, we may verify that between -1.96 and 1.96 the total number of points constitutes 95% of a mass of the distribution:

```
N = 100000000
rvs = np.random.randn(N)

print(np.sum((rvs > -1.96) & (rvs < 1.96))/N)

0.9500016
```

By analysing this example you may grasp the idea of counting for NumPy arrays. Are you sure? Have a closer look at the following two cases. Coming back to our **x** matrix,

```
print(x)

print(x[(x < 10) & (x > 4)])
print(np.sum((x < 10) & (x > 4)))

[[-1 -9  0  6  4]
 [ 0 -3  7  0 -3]
 [-7  0  8 -8 -2]]

[6 7 8]
3
```

while

```
print(x[(x < 10) & (x > 4)])
print(np.sum(x[(x < 10) & (x > 4)]))

[6 7 8]
21
```

In the first case, the use of the `np.where` function returns a total number of elements meeting $(x < 10) \& (x > 4)$ condition whereas in the second case, it returns a sum of all elements. A gentle difference in notation, a huge difference in results. Keep that in mind!

Based on a specified logical condition, if only a general information is required to be obtained, one can use `np.any` and `np.all` functions:

```
np.any    print(x)
          print(np.any((x > -2) | (x < -4)))

np.all    print(y)
          print(np.all(y > 0))

[[-1 -9  0  6  4]
 [ 0 -3  7  0 -3]
 [-7  0  8 -8 -2]]
True

[-6  1  9 -7 -1]
False
```

3.8.5. Measures of Central Tendency

If you come to NumPy from Excel, your brain (still) works in 1D or 2D frame for which the fundamental calculations of the mean, variance, standard deviation, etc. (i.e. the measures of central tendency) seem to be straightforward. Below, let me guide you on how to recalibrate your thinking and adopt your brain cells to the Python's version of the same side of the world we live in.

For **1D arrays**, say, we have:

```
a = x[(x > 0) & (x < 10)]
print(a)

print(a.size)           # number of elements
print(np.sum(a))        # sum
print(np.mean(a))       # mean
print(np.var(a, ddof=1)) # sample variance
print(np.std(a, ddof=1)) # sample standard deviation
print(np.median(a))     # median

from scipy import stats
print(stats.mode(a))    # mode
print(stats.mstats.mode(a)) # mode
```

A number of degrees of freedom `ddof=1` denotes a sample variance and sample standard deviations here. Use `ddof=0` for population measures.

```
[6 4 7 8]

4
25
6.25
2.916666666667
1.70782512766
6.5

ModeResult(mode=array([4]), count=array([1]))
ModeResult(mode=array([6]), count=array([1]))
```

where last two functions return an array of the modal (i.e., most common) value in the passed array (mode), and an array of counts for each mode. There is a subtle difference in results for derived modes and, in our case, is caused by the implementation plus the uncertainty in finding the most common value for a set of four numbers where each of them occurs only once. Make sure to double check your results working with a larger data set.

For **2D arrays**, we use:

```
r = np.arange(15).reshape(3, 5)
print(r)

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

print(r.size)           # number of elements
15

print(np.sum(r, axis=0)) # sums for columns
print(np.sum(r, axis=1)) # sums for rows

[15 18 21 24 27]
[10 35 60]
```

```

print(np.mean(r, axis=0)) # mean values for columns
print(np.mean(r, axis=1)) # mean values for rows

[ 5.  6.  7.  8.  9.]
[ 2.  7. 12.]

print(np.var(r, axis=0, ddof=1)) # sample variances for columns
print(np.var(r, axis=1, ddof=1)) # sample variances for rows

[ 25.  25.  25.  25.  25.]
[ 2.5  2.5  2.5]

print(np.std(r, axis=0, ddof=1)) # sample std devs for columns
print(np.std(r, axis=1, ddof=1)) # sample std devs for row

[ 5.  5.  5.  5.  5.]
[ 1.58113883  1.58113883  1.58113883]

from scipy import stats
print(stats.mode(r, axis=0)) # modes for columns
print(stats.mode(r, axis=1)) # modes for rows

ModeResult(mode=array([[0,1,2,3,4]]), count=array([[1,1,1,1,1]]))
ModeResult(mode=array([[ 0],
                        [ 5],
                        [10]]), count=array([[1],
                        [1],
                        [1]]))

```

where, as previously, the values of modes are inaccurate.

Having that all said, we are now equipped with a powerful tool to perform fundamental operations on 1D and 2D arrays. The cornerstone of **NumPy for quants** has been laid.

In *Volume II* we will discover more advanced concepts of NumPy and explore a greater number of useful examples with numerous fruitful applications in quantitative finance.

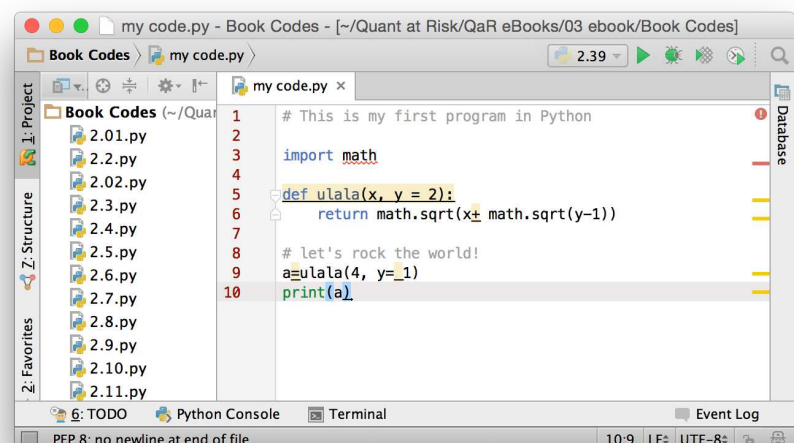
Appendix

A. Recommended Style of Coding in Python

The best practices for writing a readable code in Python has been officially specified within so called **PEP8** document available at: <https://www.python.org/dev/peps/pep-0008>. It is a part of the Python Developer's Guide which lists all accepted, and those that wait to be accepted, proposals. The main goal of PEP8 is to describe all recommended practices while completing your code in order to make it widely (worldwide) and easily readable.

If you begin your journey with Python, I strongly encourage you to read PEP8 throughout. Most probably, you will not understand a lot at the very beginning. Therefore, don't worry about that.

I could give you tonnes of examples here to guide you through however, what I have discovered to work really well for me, was the use of one of good IDE software solutions, e.g. PyCharm described in Section 1.4.3. Your IDE will make of the effort to underline and remind you all mistakes you have committed while typing the entire Python code inside the editor. Have a look at the following screenshot where I tried (in PyCharm) to write a piece of a code as a programmer with no knowledge of PEP8:



The inner module responsible for checking the PEP8 rules while typing marked in the text (with a yellowish hue of that colour) as well as in the right margin its remarks on violation of the PEP8 good practices.

Firstly, it points at the definition of the function to be written rather as:

```
def ulala(x, y=2):
```

Next, it suggests to add an extra space in the following line, i.e.:

```
    return math.sqrt(x + math.sqrt(y-1))
```

while the calling of that function in the main body of your program should be more warmly accepted if you type it as:

```
a = ulala(4, y=1)
```

Adding an additional line after the line #10 makes PEP8 to celebrate your maturity in that domain. Do not forget to separate every function with double empty lines before and after.

Trust me, once you become a good Python developer, the world will thank you for your commitment to a precisely followed trajectory of the PEP8 guide. ☺

B. Date and Time

Python 3.5 offers the abundant set of solutions for *date* and *time* formatting. It is a part of the Standard Library. It is sufficient for you to explore and familiarise yourself with a `datetime` module (<https://docs.python.org/3.5/library/datetime.html>).

The measurements of time in Python 3.5 you may study based on materials given at <https://docs.python.org/3.5/library/time.html>.

C. Replace VBA with Python in Excel

Quants love to use VBA. It is an embedded part of the Microsoft Excel software. It is easy to learn (you do not need more than one day to be fluent in VBA just by studying the on-line random tutorials and examples). The main pain related to VBA is in its speed. A bunch of sophisticated Monte-Carlo simulations may consume a lot of time before the results appear in the spreadsheet. Is there any cure for that?

And you might guess what I am going to tell you right now. *Of course!* Personally, I worked for a client who asked me to deliver an Excel-based tool to perform the analysis of his on-line business. The project required the establishment of a firm connection with

Google Analytics for the most recent data update and their further processing. Moreover, the client wanted a clear separation of all data processed in a form of a multi-sheet Excel workbook with some fancy plots included. Since his working environment was based on Microsoft Windows, I decided to create a whole workbook powered by computations in Python making use of **DataNitro** (see: <https://datanitro.com>).

Already described in *Python for Finance* book written by my dear colleague Dr. Yves Hilpisch (O'Reilly Media, 2014), **DataNitro** is a perfect solution you can install in MS Windows' Excel for U\$99 per a single licence (one month free trial available). All its on-site documentation and guidance will help you to understand how to connect Python (e.g. Python's NumPy-based computations) with Excel itself.

That's not the all.

If you operate not only in Windows, but in Linux or Mac OS X, you should consider making use of the goodies of another great tool for your all-day/night-long **Python-Excel intercourse**—the **xlwings** module (<http://xlwings.org>) delivered to you by my friend, Dr. Felix Zumstein of Zoomer Analytics, LLC. His introductory videos and few examples allows you for an instant integration of Python with Excel. Adding the plots from Python's `matplotlib` to the Excel's spreadsheet is no longer a problem.

I have tried it and I love it! Just organise an extra hour of your spare time and dive into it. For sure, **xlwings** will add wings to your Excel-based numerical solutions! Highly recommend!!

D. Your Plan to Master Python in Six Months

When I was at primary school, I had four years of French. My teacher, Mrs. Halina Kowalska, gave a bullet-point list on how one should learn a foreign language. Let me use the best out of her advice for your course of actions when it comes to studying the Python language over next 6 months:

1. Write Python codes every day for at least an hour. Forget about mistakes you do. There is no game without the pain.
2. Study this book and try to embrace all examples I designed for you. They are individually crafted to make you aware what is going on inside the code. To teach you the way how you should inspect the Python code. From the very beginning.
3. Stop and modify any Python code you come across. Try to wonder on the alternative solutions.
4. Dedicate yourself to be fluent in Python in six months from now. Why? Because only a clarity of your goal pulls all required energy towards you. The more you devote time and effort to become the best in Python, the sooner is the day when you exceed your own expectations.

5. Don't throw the towel in so quickly. Don't give up just because something does not work. Be over the level of the problems you detect. Inspect, instead. Be the first one who spends a whole night in order to find a correct solution.
6. Support your journey with good resources. Buy books, scan multiple solutions for a similar problem on the Web. Trust me, the question you want to ask (in) Google, someone somewhere has already asked!
7. Subscribe to the daily updates on **StackOverflow** website. Open your own account at <http://stackoverflow.com> and choose the keywords such as *numpy*, *scipy*, *python*, etc. to be included in your morning emails. This is the greatest source of solutions for Python programmer I can thought of right now!
8. Subscribe to all possible accounts on **Twitter** that deliver news on Python. To kick off, visit my Twitter account (<https://twitter.com/quantatrisk>) and follow all other users I follow who publish on Python daily.
9. Subscribe to **PythonWeekly** (pythonweekly.com) for updates. You may learn a lot from reading the stuff they send out.
10. Join all possible Python meet-ups around you. Make an effort to challenge your coding every week to be better. Never give up. Become unstoppable so one day you will know Python so well that no one will even think of not hiring you in their company! You deserve it. Fight for what is best for You. Good luck!

